Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

# Conflict Visualization for Evolving UML Models

Petra Brosch[a] Martina Seidl[b] Manuel Wimmer[c]
Gerti Kappel[a]

a. Business Informatics Group, Vienna University of Technology, Austria, http://www.big.tuwien.ac.at

b. Inst. of Formal Models and Verification, Johannes Kepler University, Austria, http://fmv.jku.at

c. Software Engineering Group, Universidad de Málaga, Spain, http://www.lcc.uma.es

Abstract The urgent demand for supporting teamwork and continuous evolution of software models triggered intensive research on optimistic version control systems for models. State-of-the-art model versioning approaches primarily focus on detecting changes and conflicts between concurrently evolved versions of a model. However, techniques for conflict visualization have been hardly investigated yet.

In this paper, we propose to support the visualization of conflicts in the concrete syntax of UML models. For this purpose, we present an approach to tentatively merge concurrently evolved versions of one model featuring all performed changes, yet keeping conformance to the UML metamodel. Changes and conflicts are visualized in this tentatively merged model without requiring any editor extensions. Instead, we employ the powerful profile mechanism of UML to enable modelers to resolve conflicts within their favorite UML editor.

Keywords model versioning; diagram merging; conflict visualization.

## 1 Introduction

With the application of model-driven engineering (MDE) techniques, modeling activities have matured from creating pretty pictures to producing artifacts translatable to executable code. Whereas models have originally been used for mere documentation purposes to communicate ideas, requirements, and designs, in MDE models are now lifted to first-class citizens taking an important role in the software development process. This upgrowth intrinsically demands tool support for managing evolution [FR07, SMB09]. Whenever functionalities are added and extended or when bugs are fixed, the underlying models have to be updated. Consequently, the same

Petra Brosch, Martina Seidl, Manuel Wimmer, Gerti Kappel. *Conflict Visualization for Evolving UML Models*. In Journal of Object Technology, vol. 11, no. 3, 2012, pages 2:1–30.
doi:10.5381/jot.2012.11.3.a2

kind of change management as successfully applied for textual code is required. Since models and code differ in many aspects, the techniques and tools available for textual code can be hardly reused.

Models are often rendered visually. Further, models provide a higher level of abstraction than textual code in order to deal with the complexity of modern software. These features of models explain the attractiveness of applying MDE. In general, modern software is so huge that it cannot be built by one single engineer, but a team or teams of engineers are necessary to satisfy the given time constraints [GJM02]. Consequently, for traditional software engineering, several paradigms like pessimistic and optimistic version control have been proposed for supporting the collaboration between multiple developers [BKL+12]. Especially, optimistic version control systems (VCS) gained high popularity by allowing engineers to work independently of other team members on their personal local copy and changes are merged at a later point in time. The benefits of working in parallel come at the price of incorporating the isolated changes of the modified artifact, which is a tedious and error-prone manual task when changes do not commute. For code, merging works satisfying well in practice. For models, the situation is different.

The first pragmatic attempts for realizing model evolution support as well as collaborative modeling support failed, where text-based versioning systems like Subversion[1] and CVS[2] were reused. These systems were successfully applied for code beforehand. However, it quickly turned out that the models' textual XMI serializations are neither an appropriate representation for machines to detect conflicts, nor an appropriate representation for humans to understand and resolve conflicts [ABK+09, BE09]. Consequently, dedicated model versioning systems emerged, operating on a graph-based representation of the model's abstract syntax [ASW09]. While these approaches advantage precise conflict detection, the expected boost for manual conflict resolution is still absent. The main reason is that conflicting changes are visualized in the abstract syntax of the models, while modelers are familiar with the concrete graphical syntax. For manual conflict resolution, all changes and the resulting conflicts must be well understood, which represents a huge challenge without the familiar view carrying the *mental map* [ELMS91], i.e., the personal view of the modeler on the model. This mental map is closely related to the arrangement and the layout of the model elements within diagrams representing the models. Although it seems natural to employ these diagrams as user interface for accessible conflict resolution, most model versioning systems totally neglect diagrams. Only a few graphical differencing approaches have been proposed [MGH05, OWK03] working on the diagram level. They generate a dedicated difference view by combining and highlighting changes using coloring techniques. However, all these approaches require for specific editor extensions, and thus, are hard-wired to the specific modeling environments.

We pursue the idea of presenting merge conflicts in the concrete syntax of the modeling language and present an approach for representing and visualizing merge conflicts for UML models based on UML profiles [Obj11c]. With this approach, standard UML modeling environments may be directly reused without adoptions of the modeling editor or heavy-weight extensions of the UML metamodel. Conflicts are visualized by the means of special annotations (i.e., stereotypes and tagged values) within the diagrams. If multiple model elements are involved in one conflict, we introduce UML collaborations to interlink these elements. To this end, the merge

[1] `http://subversion.tigris.org`
[2] `http://cvs.nongnu.org`

problems become directly visible to the modeler, because conflict visualization is performed within the concrete syntax of the modeling language.

In this paper, we first introduce a representative modeling scenario where several versioning conflicts occur and give an overview how state-of-the-art model versioning systems detect such conflicts. For the internal representation of conflicts, we use a conflict model which we shortly present in Section 3. In Section 4, we show how the information contained in this conflict model is expressed by the means of a UML profile. We then present a conflict aware merge strategy using this UML profile for the generation of a *Conflict Diagram* in Section 5. The concrete implementation of the approach is discussed in Section 6. Finally, we conclude with a review of the related work, a critical discussion of our approach as well as an outline to future research directions.

This article is an extension of [BKL+11b]. The conflict model and the corresponding UML profile have been significantly revised and extended. Furthermore, also the conflict diagram generation has been improved. The discussion about concrete implementation issues (cf. Section 6) is completely new.

## 2 Conflicts in Model Versioning

In optimistic model versioning, conflicts occur when the modifications of two or more models cannot be integrated in one unique and consistent model. The focus of this paper is merging and visualizing conflicting models, rather than detecting conflicts. However, as the notion of conflicts is a central prerequisite and conflicts act as input for the visualization approach presented in this paper, we briefly discuss the various kinds of conflicts with the help of the example shown in Figure 1 and aim at giving an intuition how state-of-the-art conflict detection components like [CRP08, KHvWH10, TELW10, BKL+10] detect and report such conflicts.

The reasons for conflicts are manifold as argued in [BKL+11a, TELW10, Men02]. According to [Men02], there are three main techniques to detect conflicts. (1) Generic conflict detection approaches like, e.g., [KHvWH10, TELW10, BKL+10], employ either *merge matrices* or the more general technique adopted from graph transformation theory [EEPT06] called critical pair analysis to determine applied modifications leading to non-commutative results. (2) Alternatively, to adapt conflicts to a certain domain and to allow for context-sensitive conflict detection, combinations of operations leading to a conflict when merged together may be specified as *conflict set*. Conflict detection is then performed by searching for those forbidden change patterns within all parallel applied operations, as, e.g., done by Cicchetti et al. [CRP08]. (3) Similarly, *semantic* conflict detection techniques go beyond generic conflict detection as domain specific constraints are evaluated. Semantic conflict detection may be applied to the merged model only and changes causing violations are determined retrospectively. A first promising approach for semantic conflict detection is presented in [Men99]. However, semantic conflict detection is expensive in terms of processing power and is therefore not directly supported by state-of-the-art model versioning systems.

For an overview of the conflicts regarded in this work, consider the following example. The modelers Harry and Sally work together in a project, where an event management system has to be developed. To support their collaboration, the artifacts under development are exchanged via the central repository of an optimistic model versioning system. One day, Harry checks out the model which is named Original Model in Figure 1 from the repository. This model contains a UML Class Diagram as

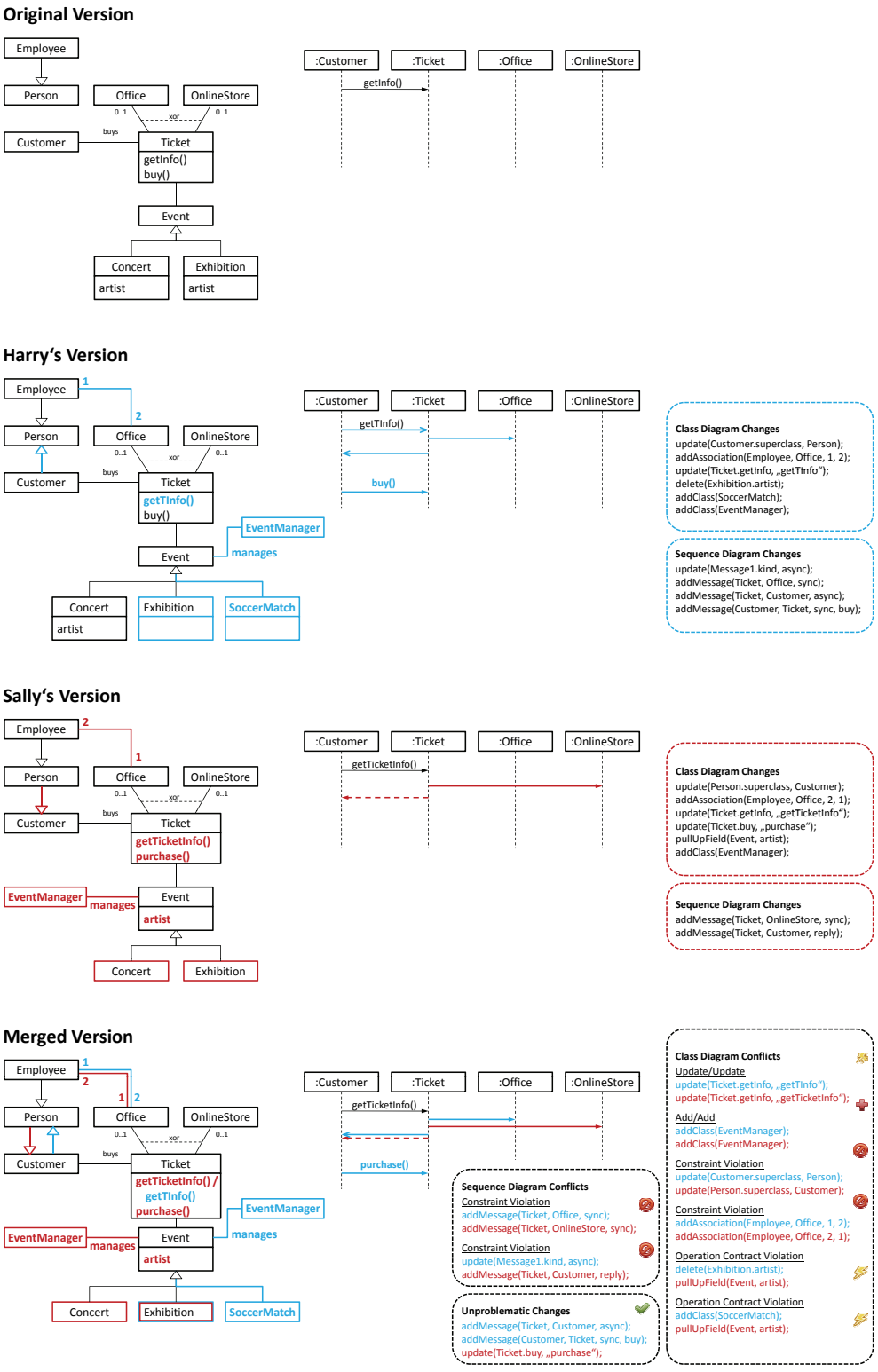


Figure 1 – Model Versioning Example

well as a UML Sequence Diagram which specifies some interactions between certain elements of the Class Diagram.

Harry performs the following modifications. In the Class Diagram, he introduces a generalization relationship between the classes Person and Customer, such that Customer becomes the subclass of Person. Then he adds an association between the classes Employee and Office with multiplicities 1 and 2, respectively. He further renames the operation getInfo() of class Ticket to getTInfo() and removes the attribute artist from the class Exhibition. Finally, he adds the two new classes SoccerMatch and EventManager. The class SoccerMatch gets subclass of Event. In the Sequence Diagram, Harry changes the message getTInfo() (recall that it was renamed in the Class Diagram) to an asynchronous message and adds a reply to this message which is also asynchronous. Between this request and reply messages he adds a message to an interaction partner of type Office. Furthermore, he introduces the call of operation buy from Customer to Ticket. He commits his modifications to the repository.

While Harry works on the model, Sally also checks out the Original Model. Unaware of Harry's modifications, she does the following. She relates the classes Customer and Person with a generalization. For her, the Customer is the superclass (everyone is a customer, also a company can be a customer) and Person is the subclass. She adds an association with multiplicities 2 and 1 between Employee and Office. She changes the name of the operation getInfo() to getTicketInfo() and she renames the operation buy() to purchase(). Then she performs the refactoring pullUpField on the classes Concert and Exhibition shifting the attribute artist to the class Event. She adds a class EventManager. In the Sequence Diagram, she adds a message from Ticket to OnlineStore after which she adds a reply message from Customer to Ticket.

When Sally tries to check in her changes, the model versioning system reports that an automatic merge is not possible, because her and Harry's modifications are partly incompatible. The following problems have to be solved.

1. *Contradicting Changes.* The probably most common conflict in model versioning is a conflict due to contradicting changes, where one modeler modifies a model element deleted by the other modeler or where both modelers modify the same model element in different ways. In our example, this conflict occurs for the renaming of the operation getInfo().

   Contradicting changes may be detected either by employing merge matrices or conflict sets. However, when contradicting changes are reported strongly depends on the granularity level of the conflict detection component, i.e., what kind of model element is considered to be atomic. For example if a class is considered as atomic unit of comparison, a conflict is reported when the names of two different attributes are modified. If the granularity level is set to consider every single feature of an element as atomic unit, then no conflict is reported in this case. Most model versioning systems support fine-grained conflict detection based on single features. In order to adapt the granularity of the conflict detection algorithms, [KHvWH10, CRP08] allow to customize the unit of comparison.

2. *Equivalent Changes.* Harry and Sally both introduced a class called EventManager. Such a situation may be handled in various ways by the conflict detection component, depending on the strategy how models are compared beforehand. (1) The two elements are considered as different elements, especially when the comparison relies on universally unique identifiers (UUID) of

elements. Then this element is inserted twice in the merged model, as, e.g., in [OWK03, MGH05, MCPW08]. (2) The two elements are considered as equal and they are merged, i.e., a model element is included in the merged model which contains all features of both. (3) A conflict is reported. Possibilities (2) and (3) depend on the unit of comparison and on the heuristics of the employed comparison algorithm. While EMF Compare[3] for example uses built-in heuristics regarding attribute values and references of model elements, the Epsilon Comparison Language [Kol09] allows to specify sophisticated comparison rules for different model elements, e.g., it may be specified, that classes are considered equal if they have the same name regardless of other attribute values.

3. *Constraint Violations.* Graphical models and their respective metamodels like, e.g., the Superstructure [Obj11c] for UML, are usually not precise enough to provide an unambiguous specification. Therefore, models and metamodels may be supplementary restricted by well-formedness rules specified, e.g., by OCL [Obj12] invariants. In the context of model versioning, even if the two modelers perform changes which do not impact the conformance of the model to its well-formedness rules, under certain circumstances the combination of the modifications of both modelers might result in an invalid model.

   (a) *Metamodel Constraint Violations.* For a metamodel constraint violation consider for example the introduction of the generalizations in Figure 1, i.e., the inheritance from Person to Customer for Harry, and the inheritance from Customer to Person in Sally's case leading to an inheritance cycle when applied in combination. Another metamodel constraint violation is shown in the Sequence Diagram. The message type of getInfo() is changed by Harry from synchronous to asynchronous. Furthermore, an asynchronous message is added to model the reply to the request. Sally, in contrast, added an explicit reply message. In the merged version, this reply message follows an asynchronous message, what is not allowed.

   (b) *Model Constraint Violation.* Besides specifying constraints for the usage of modeling languages, constraints may also be specified on the model level to restrict instances of a model or the model itself. Further, in modeling languages like UML, models consist of different diagrams. A diagram provides a specific view on a certain aspect of the model. For example, UML provides the Class Diagram for describing the structure of a system and the Sequence Diagram for describing the interactions happening between certain modeling elements. Within a diagram, constraints may be defined which hold for the diagram itself or constraints may be defined which restrict another diagram. We therefore distinguish *Intradiagram Constraints* and *Interdiagram Constraints.* A violation of an intradiagram constraint occurs in our example, as Harry and Sally both add an association between the classes Office and Employee, but with the opposite multiplicities. When analyzing the multiplicities, it becomes obvious that the model can never be instantiated, because one object of type Employee would need two objects of type Office and one object of type Office would need two objects of Employee. For a violation of interdiagram constraints consider the xor-constraint, which states that an object of type Ticket is either related

[3] http://www.eclipse.org/emf/compare/#compare

to an object of type Office or an object of type OnlineStore, but not to both. When we merge the two different versions of the Sequence Diagram, we have a ticket that communicates with both, i.e., the xor-constraint is violated.

Current model versioning systems do not directly provide built-in validation facilities, because such conflicts are often not evident until the merged version is constructed. They therefore either (1) refer to external validation solutions, such as EMF Validation[4] and [GE10, GDKR+11] or (2) incorporate language specific conflict sets in their conflict detection, such as Cicchetti et al. [CRP08].

4. *Operation Contract Violation.* Sally performed the refactoring pullUpField to shift the common attribute artist of the classes Concert and Exhibition to their superclass Event. When combined with Harry's changes, two problems arise. (1) The newly introduced class SoccerMatch, which is also a subclass of Event, would inherit the attribute artist in case the refactoring was applied. Usually few artists are involved in a soccer match, but this knowledge is seldom available to the conflict detection component. Still, this conflict may be detected and reported, if the conflict detection component is aware of composite operations [DMJN08]. A composite operation is a set of atomic changes which are applied in combination to perform a certain goal, like a refactoring. Composite operations are usually specified in terms of an operation contract [Mey92] asserting pre- and postconditions for their application. In our example, the added class SoccerMatch violates the precondition of the pullUpField refactoring, stating that all subclasses have to share the feature which shall be pulled to the superclass. (2) As Harry removes the attribute artist from class Exhibition, the refactoring is not applicable to Exhibition anymore, as again the precondition of the refactoring is not fulfilled for this class.

   Such conflicts are only detectable, if the conflict detection component is aware of composite operations. Therefore, Cicchetti et al. [CRP08] proposed a pattern language for the specification of conflict sets called conflict models in order to adapt their conflict detection mechanism to specific domains. Recently, a couple of model versioning systems are proposed, which are capable to reuse existing composite operations specified as model transformation. They either track the application of such model transformations directly in the editor [KHvWH10], or retrospectively analyze atomic differences to detect composite changes [KKT11, BKL+10]. In case composite operation specifications are available in terms of a declarative model transformation, e.g., a graph transformation, this specification may be reused by conflict detection components to perform a critical pair analysis [LEO06] and report conflicts in a fine-grained manner. Then, the conflicts occurred in our example are explicated as *Add/-Forbid* in case of the added class SoccerMatch and *Delete/Use* for the deleted attribute artist in class Exhibition.

The different kinds of conflicts discussed above represent the most common conflicts identified in the recent model versioning literature. The detection of these conflicts is either provided by state-of-the-art conflict detection components of versioning systems or may be achieved via external validation tools.

[4] http://www.eclipse.org/modeling/emf/?project=validation

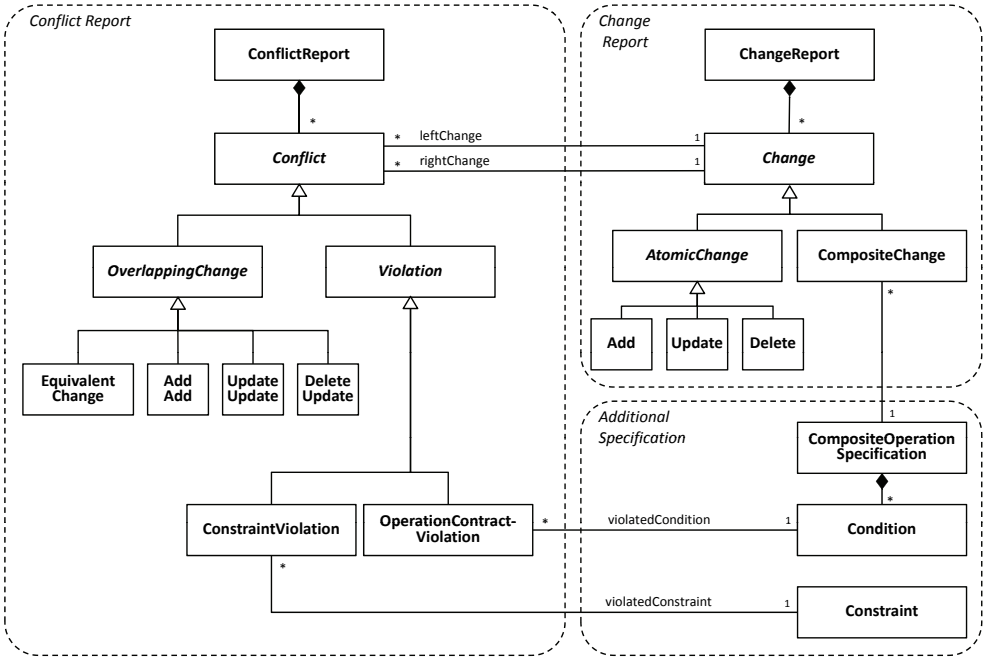


Figure 2 – Conflict Model

To sum up, in this paper we distinguish two groups of conflicts: *conflicts due to overlapping changes* and *violations*. The former is the result of syntactically overlapping changes which may be generically detected by analyzing the performed add, delete, and update operations, the latter require special knowledge on constraints of the employed modeling language and are only detectable either by searching for forbidden conflict sets or by validating the merged version.

## 3 Conflict Model

In order to process the conflicts discussed in the previous section, a dedicated data format for exchanging and representing information on detected conflicts is needed. To follow the model-driven engineering paradigm and to reuse existing techniques, our means of choice is a model-based representation of conflicts. The *Conflict Model* presented in the following serves as interface to decouple conflict detection and conflict visualization components. Obviously, the conflict model may be used as mediator to consolidate various conflict detection components. For example, the information on overlapping changes may be retrieved from a merge matrix based conflict detection component like [TELW10], while conflicts due to violated constraints may be collected from a conflict set based approach like [CRP08].

Thus, the conflict model is agnostic from any specific conflict detection technique and describes the output of conflict detection components in a structured manner, i.e., it precisely represents the context of occurred conflicts, not a conflict pattern. The context of a conflict is given by the involved model elements, the performed changes as well as the violated constraints. These constraints are either conformance rules defined in the model itself and in its metamodel, or pre- or postconditions of a com-

posite operation. In our conflict model depicted in Figure 2, we therefore access two sources of information for obtaining a *Conflict Report*, namely the *Change Report*, comprising the applied changes, and *Additional Specifications* made available to conflict detection components beforehand, to improve the quality on detected conflicts. Additional specifications describe language specific composite operations like refactorings as well as conformance rules. All these information are obtained from external conflict detection components like, e.g., [BKL+10, KHvWH10, TELW10, CRP08] and validation services [GE10, GDKR+11].

With this conflict model we may profoundly express the kinds of conflicts discussed in the previous section. Conflicts are either *Overlapping Changes* or *Violations*.

- *Overlapping Changes.* A conflict caused by overlapping changes always references two changes which either interfere with each other (Contradicting Change) or where one change makes the other change obsolete (Equivalent Change). In the example of Figure 1 the renaming of the operation getInfo() results in a conflict due to overlapping changes. The introduction of the class EventManager is considered equal and is thus automatically merged.

- *Violations.* For conflicts due to violations we distinguish the following subclasses:

  - *Operation Contract Violation.* A conflict due to the violation of an operation contract always involves at least one composite change like a refactoring. This change cannot be performed because another change violates a precondition, a postcondition, or an invariant specified by the operation contract. Composite changes may be specified with a tool like EMF Modeling Operations as proposed in [BLS+09], in terms of graph transformations as proposed in [KKT11, TELW10], or as change pattern which is part of a conflict pattern [CRP08]. We distinguish two cases: a composite change is either not applicable because a model element violating the change's operation contract has been added (e.g., class SoccerMatch in Figure 1) or an existing model element necessary for the execution has been changed or deleted (e.g., attribute artist of class Exhibition in Figure 1).
  - *Constraint Violation.* Conflicts may arise if the merged model violates metamodel constraints or constraints specified within the model. For example in Figure 1, the inheritance cycle or the violation of the xor-constraint defined in the Class Diagram are representatives of this category of conflicts.

Note that this conflict representation is not adopted for a certain modeling language and may be achieved by various conflict detection components (at least with the help of a simple adapter). As the OMG standard for diagram interchange [Obj06] proposes to consider the model's graphical visualization, i.e., its visual diagrams also as models, we apply the same conflict model for the representation of layout conflicts. For example, if two modelers move the shape of a model element to different positions, we have a conflict due to overlapping changes. In the following, we discuss the technical realization in more detail and explain how this conflict model serves as basis for the representation of conflicts within modeling environments.

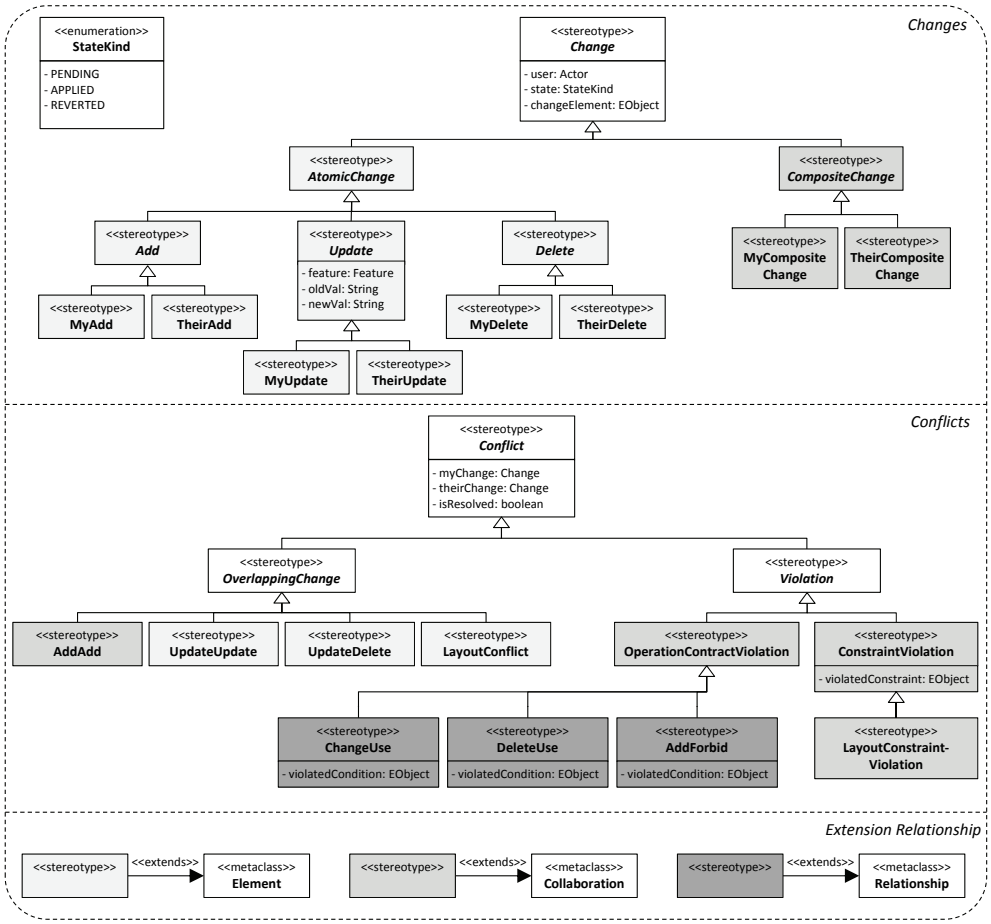


Figure 3 – Versioning Profile

# 4 A UML Profile for Model Versioning

In the previous sections, we have discussed how to represent the essence of the different kinds of merge conflicts in terms of a conflict model. However, when it comes to presenting the conflicts to the user in charge of merging two models, appropriate visualization techniques are essential. In this section, we elaborate on the supportive merge visualization of UML models. Our premise for visualizing conflicting UML models is that the user (1) should remain in her familiar UML tool, (2) should start with a tentative, automatically merged version comprising unproblematic changes, and (3) should be able to comprehend and reproduce changes and resulting conflicts. To this end, we present a dedicated *Versioning Profile* (cf. Figure 3). The versioning profile reflects the information of the *Conflict Report* and enables the visualization of a model's evolution, i.e., the performed changes, as well as the merge conflicts directly in the UML model. Our design rationale for using UML profiles is based on the following requirements

- *User-friendly visualization:* Information about performed changes, their respective users, and resulting merge conflicts shall be presented in the concrete syntax

of UML. This is an important requirement, because modelers know UML by its concrete syntax elements, but they are normally not experts on the abstract syntax of UML.

- *Integrated view:* All information necessary for the merge of one diagram shall be visualized within this single diagram to provide a complete overview of its evolution. This ensures that the mental maps of the modelers can be reused for model merging, especially, needed for the manual conflict resolution.
- *Standard-conform UML models:* The models incorporating the merge information shall conform to the UML metamodel. This ensures that the models can be opened by UML modeling editors and processed by UML model manipulation tools.
- *Model-based representation:* The merge information shall be explicitly represented as model elements. Fulfilling this requirements allows to facilitate model exchange between UML tools as well as postponing or delegating the resolution of certain conflicts by just saving the models with the conflict annotations.
- *Non-intrusive editor extensions:* The visualization of the merge information shall be possible without modifying the graphical editors of UML tools. This allows to be independent of a concrete UML modeling tool and avoids to provide proprietary tool extension which may be hard to maintain in the future.

UML profiles define a lightweight extension to the UML metamodel and allow for customizing UML to a specific domain. UML profiles typically comprise *stereotypes*, *tagged values*, and additional *constraints* stating how profiled UML models shall be built. Stereotypes are used to introduce additional modeling concepts which extend standard UML metaclasses. Once a stereotype is specified for a metaclass, the stereotype may be applied to instances of the extended metaclass to provide further semantics. With tagged values, additional properties may be defined for stereotypes. These tagged values may then be set on the modeling level for applied stereotypes. Furthermore, syntactic sugar in terms of icons for defined stereotypes may be configured to improve the visualization of profiled UML models. The major benefit of UML profiles is reflected by the fact that profiled models are still conformant to UML, i.e., they are naturally handled by current UML tools.

As shown in Figure 2, the conflict report (1) assembles the change report by comprising all changes performed to the model and its respective diagrams, and (2), marks the changes which are overlapping or violating constraints. The versioning profile reflects this separation by introducing dedicated stereotypes for changes and conflicts. Like in the conflict report, the change stereotypes comprise the information about how a specific model element has evolved. Conflict stereotypes are introduced to annotate merge problems and link to the respective changes. The versioning profile is derived from the previously described conflict report but explicates additional information, which is only implicitly available beforehand.

**Changes.** The versioning profile provides stereotypes for each kind of change. To provide provenance information, each «*Change*» stereotype has tagged values to make the responsible user explicit. Additionally, as the stereotypes are not only used for mere visualization purposes, but also for supporting the merge process in terms of dedicated tooling (cf. Section 5), status information indicating whether the change has been already applied, is introduced. To complement tooling related information,

each change may be traced back to the corresponding change element in the change report. Changes are either atomic or composite. An «*AtomicChange*», i.e., add, delete, or update, may be applied to any concrete UML element, i.e., Class, Generalization, Property, etc., and thus, is defined to extend the UML metaclass Element. As updates are changes to existing elements, they have additionally tagged values for pointing to the affected feature of the changed element including its old and new value. Composite changes like refactorings, incorporate a set of indivisible atomic changes. To highlight this fact, a new UML collaboration is introduced in the merge process and annotated with the stereotype «*CompositeChange*». The collaboration connects all model elements concerned by the composite change via UML relationships. Each specific kind of change stereotype is finally defined in the form of «MyChange» and «TheirChange» to indicate which changes were originally performed by the user in charge of merging and which changes were applied by the other user.

**Conflicts.** The conflict part of the versioning profile defines stereotypes for the different conflict types depicted in Figure 2. Accordingly, a «*Conflict*» may be either an «*OverlappingChange*» or a «*Violation*». To differ conflicts in the model and in the visual diagram explicitly, the stereotypes «LayoutConflict» and «LayoutConstraintViolation» for annotating conflicts due to overlapping changes in the diagram and for the violation of special layout constraints, are introduced. «UpdateUpdate» and «UpdateDelete» stereotypes extend the UML metaclass Element, as these conflicts result from two atomic changes on the same model element. Even if a «LayoutConflict» occurs due to an overlapping change in the diagram, the stereotype is applied to the model and thus extends the metaclass Element. In this way, the visualization of layout conflicts is naturally handled by UML editors. In contrast, Add/Add conflicts and violations comprise different modeling elements. Thus, we again introduce UML collaborations to hint at the involved changes. «ConstraintViolation» and «LayoutConstraintViolation» further state violated constraints. In case of an «OperationContractViolation», the UML relationships interlinking the involved elements to the UML collaboration, are annotated with stereotypes (inspired from graph transformation theory [LEO06]) indicating how the contract is violated by the model element. The stereotypes «ChangeUse» and «DeleteUse» are applied on model elements already existing in the original model, which are involved in a composite operation and changed or deleted by the other user, respectively. «AddForbid» indicates the addition of a new model element which invalidates the precondition of a composite operation. Finally, all conflict stereotypes refer via tagged values to the underlying change stereotypes, what makes understanding and reproducing the conflicts possible.

## 5 Conflict Aware Merging of UML Models

Resolving conflicts by manually exploring the common ancestor model as well as the two changed models in combination with the change and conflict reports shows to be cumbersome and error-prone in practice. Thus, we generate a dedicated *Conflict Diagram* visualizing the merged model comprising all relevant changes and detected conflicts at a single glance (cf. Figure 4 for the running example). The conflict diagram provides a tentative, automatically merged version of the model and all associated diagrams. Thus, for our motivating example, two conflict diagrams are generated, i.e., one for the Class Diagram and one for the Sequence Diagram.

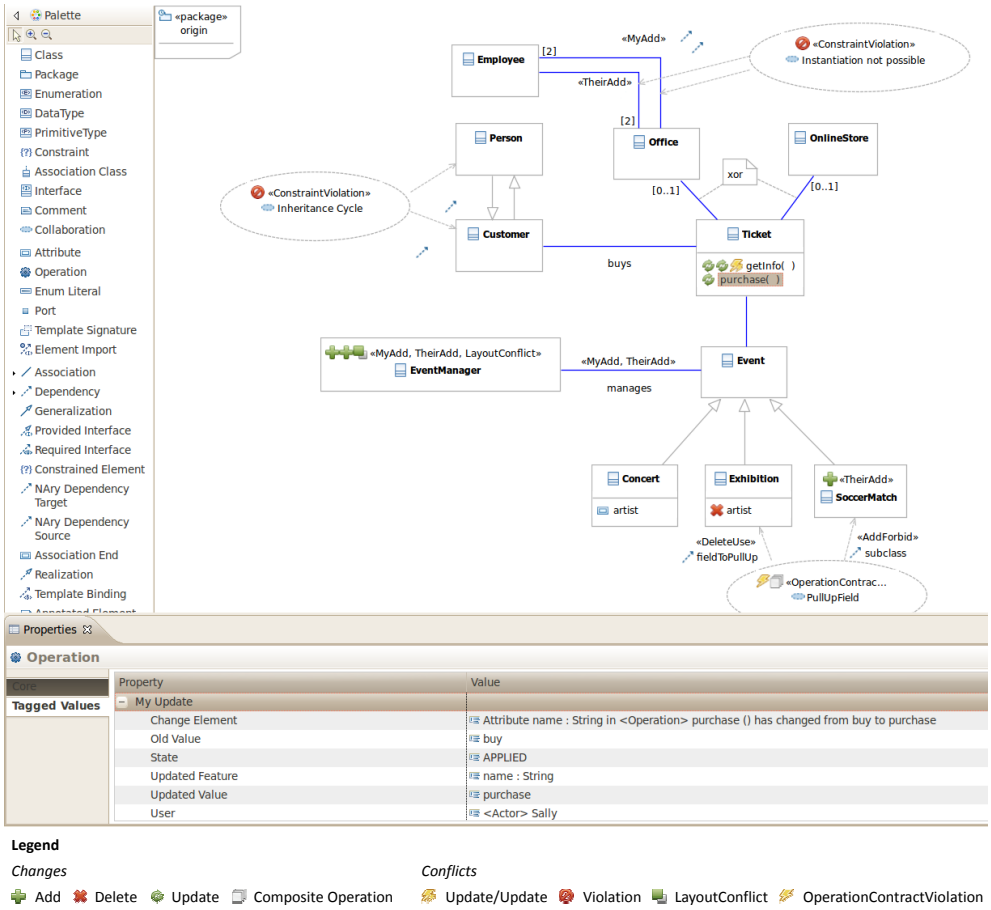


Figure 4 – Conflict Diagram for the Running Example

## Generating Conflict Diagrams

Although the conflict diagram is enriched with change and conflict information, it provides a familiar view, which the user in charge of merging, in our case Sally, can recognize as *her* diagram. It is obtained as follows:

1. All additions and non-overlapping atomic updates are applied to the common ancestor model. Deletions are skipped in order to allow annotating deleted elements with the respective stereotype (e.g., artist in class Exhibition in Figure 4). All composite changes are left out in this step since they are handled in Step 5. This allows for incorporating all atomic changes in the composite change (e.g., a refactoring) [DMJN08]. More precisely, if the class SoccerMatch added by Harry had an attribute artist, which was shifted to the superclass by Sally with the pullUpField refactoring, a re-execution of the refactoring would also include the new class and shift the attribute.

2. All changed elements are annotated with the corresponding change stereotypes defined by the type of change and the respective user. A link to the corresponding user and to the underlying Change of the change report are stored in tagged

values. The Change element is later needed to execute or undo the change in the manual merge phase. The tagged value state shows if a change is already applied, what is only the case for non-overlapping additions and updates. Deleted elements are just marked as deleted, but not yet deleted. This allows for visualizing and possibly preserving deleted elements. For updates, also the updated feature together with its old and new values are persisted. For an inserted element of Harry, cf. class SoccerMatch, annotated with «TheirAdd» as Sally did the later check-in and has now the burden of merging the models. Please note that the textual and graphical visualization of stereotypes varies for different modeling elements and editors. Thus the concrete peculiarity of «MyChange» and «TheirChange» may be visible not until visiting the property view, like in case of purchase in class Ticket in Figure 4.

3. Contradicting changes are annotated by applying the appropriate «Update-Update» and «DeleteUpdate» stereotypes to the concerned element. The actual changes are stored in tagged values by referencing to the respective change stereotypes.

4. Whenever equivalent changes occur, which cannot be reduced to one change, UML collaborations are added to connect the respective modeling elements. An «AddAdd» stereotype is attached to these collaborations and again, both corresponding changes are stored in tagged values. As the class EventManager inserted by Harry and Sally independently is deep equal, i.e., all features and references are equal, no conflict is reported by our conflict detection component. Thus, only one class is added to the model. However, two possibilities for placing the corresponding shape in the diagram exist. As the conflict diagram shall provide a familiar view to Sally, because she is merging the model, the position of her version is applied. However, a stereotype indicating a «LayoutConflict» is added to point her attention to Harry's different position for that shape.

5. As several modeling elements are involved in composite changes like in the pullUpField refactoring in Figure 4, we interlink them by introducing a UML collaboration for each distinct composite change. Before integrating composite changes, their preconditions are checked. If the preconditions are still valid, they are re-executed on the merged model. If the preconditions do not hold, an operation contract violation is at hand and the composite operation is not executed. Then the added collaboration is annotated with an «OperationContract-Violation» stereotype and relationships to the elements, which do not longer fulfill the precondition, are marked with dedicated stereotypes. In our example, the relationship to the attribute artist is annotated with a «DeleteUse» stereotype, as the deletion of the property violates the precondition of the pullUpField refactoring, i.e., every subclass must have the field to be pulled up. Similarly, the relationship to the added class SoccerMatch is marked as «AddForbid». This is also due to the missing property artist. Please note, as deleted elements like the attribute artist in class Exhibition are only marked as deleted but temporarily preserved, conflicts arising due to their non-existence may be visualized.

6. Finally, the merged model is validated. All applied additions and updates are incorporated for validation, as well as pending deletions. Violated constraints are, again, marked by adding UML collaboration elements, interlinking the involved

model elements and applying «ConstraintViolation» stereotypes. Their tagged value states the violated OCL constraint (e.g., Inheritance Cycle in Figure 4).

## Making Conflicts Accessible

The tentative, automatically merged conflict diagram provides support for manually merging the models. The algorithm described above generates a *Neutral View* of all non-overlapping changes. Based on this conflict diagram, two supplementary views are supported. *My View* automatically privileges changes of the user in charge of merging the model. Thus, instead of skipping overlapping updates, the value of my change is used. *Their View* provides the opposite view, to allow the person who performs the merge to immerse herself in the situation of the other modeler. Additionally to applying values of their changes, also the layout information of the other user's diagram is used. In our example, this would affect the class EventManager, which was introduced by both modelers, but treated as equal by the conflict detection component. In their view, the class is visualized on the right of class Event. Violations are handled in my and their view as in the neutral conflict diagram, because currently we do not trace back to the exact change causing the violation. Thus, atomic changes dominate composite operations, and in case of constraint violations, both changes are applied and marked with the help of the collaboration. The three conflict diagram views may be seamlessly transformed into each other and act as sandbox for checking various scenarios to better understand and resolve all conflicts.

The conflict diagram provides several benefits concerning the resolution of the conflicts. First of all, necessary information to resolve the occurred conflicts is provided at a single glance. Furthermore, different diagram filters may be employed on top of the stereotypes. With the help of these filters, specific kinds of stereotypes, i.e., conflicts, may be hidden enabling the user to focus on a specific conflict scenario. For example, a conflict resolution process can be supported such as firstly representing contradicting changes, subsequently, operation contract violations, and finally, constraint violation conflicts. Based on the user and state information of the stereotypes, the modeler responsible for the merge may switch between the two supplementary views my view and their view to analyze different scenarios. The stereotypes enable additional mechanisms for visualizing conflicts directly supported by state-of-the-art UML modeling tools. As depicted in Figure 4, special icons are used for stereotyped elements. However, in case of loads of changes, the icons may quickly overwhelm the diagram. For example, the operation getInfo in class Ticket of Figure 4 is decorated with three icons, i.e., the change of Harry, the change of Sally, and the resulting Update/Update conflict. In such cases, information hiding would be helpful, e.g., suppressing change icons, as the conflict stereotypes incorporate change information anyway.

Going beyond visualizing the conflict diagram, extensions to the UML editor in form of dedicated *Merge Actions* may be implemented to interact with the stereotypes. Then, pending changes may be applied or reverted to resolve conflicts. Conflict stereotypes are then deactivated by setting them to isResolved. However, this needs re-execution of conflict detection after each change. When checking in, elements marked for deletion are eventually deleted. Stereotypes for applied and reverted changes, as well as for resolved conflicts are removed. Pending changes and unresolved conflicts may be temporarily tolerated and checked in with the model for later processing, or may be handed over to another modeler as issue report.

## 6 Implementation

In the first part of this paper, we proposed to report merge conflicts in the concrete syntax of UML [Obj11c] by annotating the model elements with additional decorations and generating conflict diagrams for each corresponding diagram. Hence, it does not suffice to merge changes to the model only, but it is necessary to also merge diagrams. As diagrams are in accordance to OMG's UML diagram interchange standard [Obj06] and to the upcoming diagram definition standard [Obj11a] also models adhering to a dedicated diagram interchange metamodel, generic differencing and conflict detection facilities may be reused. Nevertheless, special care is necessary in the merge phase to keep model changes and diagram changes in sync. In this section, we therefore discuss some challenges we met while implementing the prototype of the presented merge algorithm and outline open issues.

We demonstrate our conflict aware merge strategy for the generation of conflict diagrams for evolving UML models, as UML is the predominant standard for visually modeling of software systems. However, OMG's diagram interchange and definition standards [Obj06, Obj11a] are not restricted to describe UML diagrams, but to describe diagrams for any MOF [Obj11b] based domain-specific modeling language. Further, alternative approaches lifting the required UML profile mechanism to the area of domain-specific modeling, are already proposed in [LWWC11, MD10, KRDM+10]. Thus the presented approach may be applied to arbitrary modeling languages, whereas we exemplary show how to use EMF Profiles [LWWC12] for this purpose later in this section.

We base our prototype on the Eclipse Modeling Framework (EMF)[5], more precisely on the EMF based reference implementation of the UML standard[6]. For the visualization, we employ UML2 Tools[7], which provide graphical editors for UML models based on the Graphical Modeling Framework (GMF)[8]. GMF in turn provides an implementation of a graphical notation metamodel close to the upcoming OMG standard for diagram definition [Obj11a], which interchange was successfully demonstrated in [EL11]. A prototypical implementation showcasing the conflict diagram generation for UML models is available at our project website[9].

**Considering Model and Diagram Changes together.** Whenever changes to the abstract syntax model are merged, the respective diagram changes have to be made to the conflict diagram. As there is usually not a one-to-one correspondence between the model elements of the abstract syntax and the elements shown in the concrete syntax, dependent changes have to be determined in order to treat them as composite unit.

Consider again the example shown in Figure 1. When switching from the concrete syntax to the abstract syntax of the diagram created with Eclipse's Class Diagram Editor (cf. Figure 5), we see that the diagram consists of several elements describing graphical elements, such as Shape, Connector, and Compartment. The concrete design rules how an element is rendered by the editor are not interchanged by the editors but persisted separately in a diagram definition model. This information is therefore only referenced via the type attribute of the diagram's elements. Further, primary view

[5] http://www.eclipse.org/modeling/emf/
[6] http://wiki.eclipse.org/MDT-UML2
[7] http://wiki.eclipse.org/MDT-UML2Tools
[8] http://www.eclipse.org/modeling/gmp/?project=gmf-notation
[9] http://modelevolution.org/prototypes/conflictvisualization

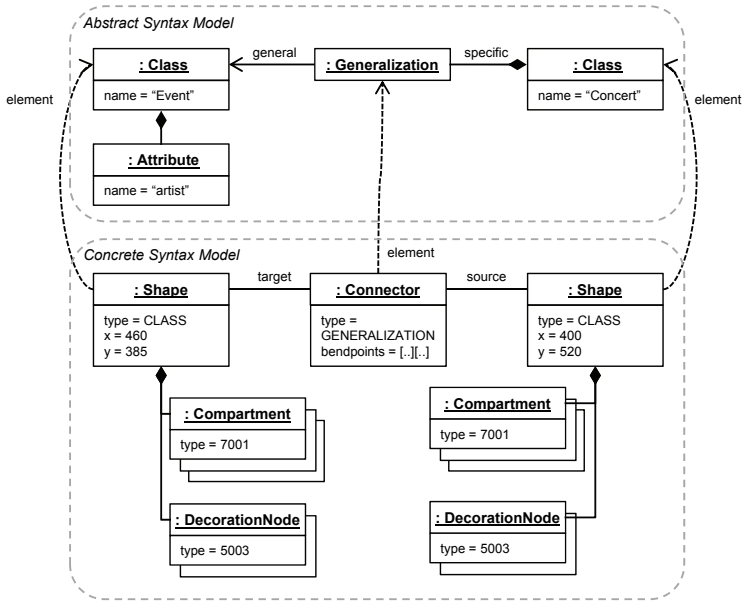


Figure 5 – Models Representing Abstract Syntax and Concrete Syntax

elements are mapped to elements of the abstract syntax model. Compartments and decoration nodes link only implicitly to the model elements and are part of a shape regardless of whether attributes or operations exist in the model or not. In contrast, decoration nodes for multiplicities of associations are optional and are only set by the editor for multiplicities different to 1, which is the default value. Thus, even if the concrete syntax model is described by a standard conform model, the outcome highly depends on the modeling editor rendering a *perfect* generic merge solution all but impossible. To overcome this limitation, our prototype implementation offers an adaptation point to plug in a custom dependency calculation for concrete syntax models of specific editors.

**Applying Stereotypes.** Even though UML allows to apply stereotypes to every model element, applying change and conflict stereotypes is not straight forward and additional editor related information is needed. For example, when the multiplicity of an association is changed, the stereotype indicating that change should not be applied on the association's end property, but on the association, because stereotypes on the property are not visualized in the UML2 Tools Class Diagram Editor. In contrast, the commercial UML tool Enterprise Architect[10] has no such restrictions.

**Placing Collaborations.** As several new UML collaboration elements and connectors are added to the conflict diagram, a fitting position has to be found in order to keep the diagram understandable. According to [WS06], the aesthetics of UML diagrams affected by the spatial layout of nodes is crucial for understanding a

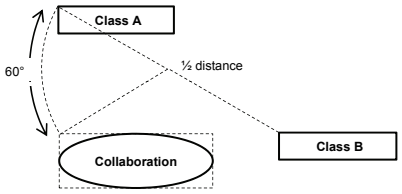


Figure 6 – Placing Collaborations

[10] `http://www.sparxsystems.eu`

model. To foster model comprehension, the collaborations should therefore be placed as near as possible by the connected elements, whereas connectors should have approximately the same length [WS06, CP96]. Thus we calculate the position of new collaborations by rotating the coordinates of the leftmost shape by 60 degrees, taking the center of the shapes to be connected as anchor, as depicted in Figure 6. We insert the collaboration on the calculated corner of the resulting triangle. In order to preserve the mental map, we do not change the layout of the existing elements. Currently, it is up to the editor to avoid crossing edges and element overlaps due to the merge and the new collaborations. However, a mental map preserving algorithm similar to the work of Jucknath et al. [JJGT06], where element overlaps are avoided by rearranging younger nodes and retaining the position of senior nodes, would be appreciated.

**Handling Layout Conflicts.** Like on the abstract syntax of the model, also on the concrete syntax of the model may occur conflicts. For example, if both modelers move a class in different directions, a conflict on the diagram occurs, even without any change to the model itself. Such kind of layout conflicts are currently automatically resolved, as the conflict diagram is generated using the layout of *MyUser* by default. Additionally, a «LayoutConflict» stereotype is applied. Similar to violations of the model, additions or moves of different shapes may also result in (partly) hidden shapes or crossing edges in the merged model. Layout constraints may be stated like metamodel constraints in terms of OCL and detected by external validation components. Violated constraints may again be indicated by introduced collaborations annotated with «LayoutConstraintViolation» stereotypes.

**Supporting other technologies and languages.** Until now, we have shown the implementation of our approach for the UML2 Tools which represent the current UML reference implementation. To show the applicability of our approach for other tools outside Eclipse, we have additionally implemented the versioning profile for the commercial UML tool Enterprise Architect. Important lessens learned are: (1) stereotype applications are shown for every model element type in the diagram, (2) powerful filtering techniques are possible and easy to configure by just stating the stereotypes which should be used as filtering criteria, (3) visualization rules for model elements may be defined again based on which stereotype is applied, e.g., the stereotype application can effect the background color of the model element.

Another implementation of the versioning profile has been achieved for EMF models in general. By using the concept of meta-profiles supported by EMF Profiles, we are able to define the versioning profile independent from the metamodels, compute the change and conflict annotations for any EMF-based model, and store them as profile applications. The annotations may be either shown in the abstract syntax in case no GMF-based modeling editor is available or in the concrete graphical syntax if a GMF-based modeling editor is available for the modeling language.

## 7 Related Work

Regarding our goal of computing conflict diagrams for evolving UML models managed by optimistic model versioning systems, we identify four, partly orthogonal threads of related work. First, we analyze strategies for integrating isolated changes in general. Second, we consider how changes and conflicts are detected and presented to the user

in model versioning approaches and review specific model merging approaches. Third, we elaborate on approaches for ensuring the mental map [ELMS91] across different diagram versions. Fourth, we present more widely related work, namely collaborative ontology engineering, where similar issues arise as in collaborative modeling.

**General Merge Strategies.** In order to deal with conflicts when merging artifacts, several strategies are conceivable. Versioning systems for code like Subversion and CVS typically employ a manual merge strategy when it comes to integrating conflicting changes. Then, the two parallel evolved versions of an artifact are shown to the user side by side, conflicting changes are highlighted. The user has to analyze the evolution of the artifact and to decide which changes shall be integrated into the merged version. For sequential artifacts like text files, merging works satisfactory well in practice. However, applied to the textual serialization of graph-based artifacts like models, this approach fails [BKL+11a].

As manual conflict resolution is error-prone and cumbersome, it seems naturally, that avoiding conflicts is a preferable goal. Munson and Dewan present a flexible framework for merging arbitrary objects, which may be configured in terms of merge policies [MD94]. Merge policies may be tailored by users to their specific needs and include rules for conflict detection and rules for automatic conflict resolution. Actions for automatic conflict resolution are defined in merge matrices and incorporate the kinds of changes made to the object and the users who performed those changes. Thus, it may be configured, e.g., that changes of specific users always dominate changes of others, or that updates outpace deletions.

In contrast, nearly as long as collaborative systems exist, several works have been published, arguing that inconsistencies are not always a negative result of collaborative development. They propose to tolerate inconsistencies at least temporarily for several reasons [NER01]. Inconsistencies may identify areas of a system, where the developers' common understanding has broken down, and where further analysis is necessary. Another reason for tolerable inconsistencies arise when changes to the system are so large, that not all dependent changes can be performed at once. Further, fixing inconsistencies may be more expensive than their impact and risk costs. Tolerating inconsistencies requires the knowledge of their existence and careful management. Undetected inconsistencies in contrast, should be avoided as they cause problems. Schwanke and Kaiser [SK88] proposed as one of the first an adapted programming environment for identifying, tracking, tolerating, and periodically resolving inconsistencies. Similarly, Balzer [Bal91] allows to tolerate inconsistencies by relaxing consistency constraints and annotating inconsistent parts with so called *pollution markers*.

Even if several merge strategies for textual artifacts are capable of automatically resolving or tolerating conflicts to a certain extent, carrying those ideas to merging models is challenging. Merged models have to obey the rules of their graph-based structure at any point in time. Otherwise, the merged model cannot be opened in modeling editors for further processing.

**Merging in Model Versioning.** In the following we discuss model versioning systems with special emphasis on the merge phase, i.e., how changes and conflicts are detected and presented to the user.

According to Mens [Men02], *state-based* approaches and *change-based* approaches may be distinguished. While state-based approaches, e.g., [MCPW08, BP08, CRP08,

XS05, LGJ07] compute the changes between two model versions by matching and difference algorithms, change-based approaches, e.g., [SZN04, KHvWH10, OS05] record the changes directly in the modeling editor as they are applied. Concerning the visualization of changes and merge conflicts, state-based approaches present the common ancestor model, the two revised models, the difference reports as well as the conflict report. In contrast, change-based approaches show the initial model, the two sets of changes from both modelers, respectively, and an optional conflict report stating dependencies between changes. Another possibility for change-based approaches is to show the initial model with applied non-conflicting changes and only the conflicting changes are outstanding. However, the aforementioned approaches neither consider to represent only one integrated model and tolerate inconsistencies during the merge phase, nor enrich the integrated model with annotations for showing the changes and conflicts explicitly to the user. Only two approaches allow to compute a single, conflict-free merged model. Ehrig et al. [EET11] produce a pre-merged version by giving updates a higher priority as deletions. By this, a first version of the merged model is generated and subsequently the user has to reason about if deletion should have actually a higher priority. Cicchetti et al. [CRP08] adopt an automatic merging approach for models by defining dedicated reconciliation strategies specifying the applicable set of change operations within their conflict patterns. In the context of a co-evolution scenario where the metamodel of a modeling language evolves, Cicchetti et al. [CRP09] present an approach where they are able to analyze the dependencies between changes and support the resolution of certain dependencies. This allows for a scheduling of the modifications. An approach for fixing inconsistencies between different diagrams is proposed in [ELF08] which uses then information on the impact of the change. All aforementioned versioning approaches neglect the concrete syntax of models, thus, no attempts are made to show changes and conflicts in the concrete syntax of models.

Finally, dedicated approaches for visualizing differences between diagram versions have been proposed by Mehra et al. [MGH05] and Ohst et al. [OWK03] by using different coloring and highlighting techniques for changed model elements shown in so called *unified diagrams* incorporating all changes of both users. The implementation of the approach of Ohst et al. has been presented in [Nie04], but solely considers two-way merges. Three-way merges are only mentioned as subject to future work. Thus, only Update/Update conflicts for attribute values and element moves are marked explicitly in the unified diagrams. Furthermore, tool-specific extensions have to be implemented for modeling editors in order to use this approach. In contrast to Ohst et al., Mehra et al. consider tree-way merges. Although conflicting changes are detected by their differentiation algorithm, no attempt is made to indicate to the user that accepting one change may invalidate another, as explicitly stated in their paper. Concerning the concrete syntax changes, a diagram with many overlapping highlighted model elements is generated in cases where a large number of changes occurred. This is because, for each movement, the origin as well as the new place of each element with a line as connector is shown. The approach has been implemented for the meta-CASE tool Pounamu [ZGH+07] for providing generic visualization support for modeling languages defined in Pounamu, but for UML modeling environments there is no support available.

The presented approach of this paper is built on top of existing model versioning systems and aims at unifying ideas from the fields of tolerating inconsistencies and the visualization of model differences. However, our approach is also related to works

regarding preserving the mental map and collaborative development in other modeling domains.

**Preserving the Mental Map between Diagrams.** There is a number of works focusing on preserving the mental map across sequences of diagrams. In general, the sequences of diagrams are created by transforming the underlying model and adjusting the diagram to the evolved model. When the mental map shall be preserved, the goal is to keep changes of the layout at a minimum such that the modeler needs not to spend much effort in realignment. This aspect is the particular focus of previous work by Jucknath-John et al. [JJGT06], Pilgrim [vP07], Johannes and Gaul [JG09], and Grimm et al. [GBPV07].

Jucknath-John et al. [JJGT06] aim at layout graphs that are transformed by a sequence of endogenous graph transformations. Their goals are: (1) achieve an optimal quality for each single graph layout, (2) retain the mental map of a graph layout, and (3) allow to identify of the changes between two succeeding graph layouts by visually emphasizing the differences. To achieve these goals, the authors introduce the concept of node aging and protection of the layout of senior nodes, i.e., nodes that have been introduced earlier than others are less likely to be repositioned by the algorithm than younger nodes.

The focus of Pilgrim [vP07] is to retain the mental map in exogenous model transformations. The proposed algorithm takes the transformed input model, the input diagram layout, the output model, and the transformation trace as input to create a new diagram layout for the generated output model. Nodes representing elements in the output model are placed according to the position of nodes representing input model elements linked by the transformation trace in order to retain the mental map. The output diagram layout is optimized by scaling and adjusting the nodes to avoid overlaps.

Johannes and Gaul [JG09] considered the diagram layout when composing domain-specific models. In their approach, the layout composition information is delivered through a graphical model composition script, which specifies how models should be composed. After the composed model is created, the diagrams of the composed model are merged into a new composed diagram according to the positions in the graphical model composition script. Finally, Johannes and Gaul also apply some algorithms to adjust the final layout to remove overlaps.

Grimm et al. [GBPV07] presented an approach for tackling the challenge of preserving the mental map when UML class diagrams have to be merged. Their approach is based on using one of the concurrently edited diagrams as so called *base diagram*, in which all modifications done for creating the other diagram, the so called *fitting diagram*, are included. For merging in a metal map preserving manner, the neighborhood of model elements in the fitting diagram has to be ensured also in the merged diagram as good as possible.

The first two mentioned approaches, Jucknath-John et al. and Pilgrim, particularly focus on retaining the mental map for transformation scenarios different from merging. Johannes and Gaul consider the composition of diagrams, but they only consider two-way merging as well as merging heterogeneous models, i.e., models which do not have the same origin model and therefore only small overlaps between the models exists. The most related approach is Grimm et al., however, they totally neglect abstract syntax conflicts and thus they do not represent conflicts in their merged diagrams. We preserve the mental map by prioritizing one view and use annotations for marking contradicting changes as well as violations of the concrete syntax within

the concrete syntax. However, we have to admit that some further means for preserving the mental map across diagram versions should be integrated in the future in our approach, e.g., for providing automatic resolutions to remove overlaps of diagram elements. Especially, the concept of node aging seems to be well-suited to be reused for merging diagrams. However, in this respect, we have to further explore if the modelers prefer the automatic resolution of concrete syntax conflicts by employing layout algorithms or if they want full control over the layout by resolving the conflicts manually.

**Collaborative Ontology Development.** Ontologies are represented by structural models which may be specified with UML class diagrams [GDD09]. Thus, we consider the collaborative development of ontologies as widely related work to model versioning. In the last years, the ontology engineering community reported needs for collaboratively developing ontologies [SNTM08]. In fact, ontologies are becoming so large that they cannot be built by a single person. Furthermore, ontologies have the requirement to be an accepted terminology and model for a particular community, so the community should be involved in the ontology development—so to speak to gain acceptance by participation. To tackle these issues, several approaches have been proposed. First, several Wiki-based environments supporting the collaborative development of ontologies have been proposed. For instance, LexWiki[11] supports to extend and refine terminologies by making comments and proposing change in a text-based manner by annotations. These annotations are later examined by curators which are editing the ontologies in standard ontology editors separated from LexWiki. A step further goes OntoWiki [ADR06] which allows to change and rate ontology definitions via a Web-based interface. However, OntoWiki does not support capabilities for representing conflicting changes explicitly. Finally, Collaborative Protégé [TNTM08] allows, as the name suggests, for the collaborative ontology development by using annotations similar as is done in the presented approach of this paper. In particular, Collaborative Protégé allows to annotate ontology changes, proposals, votings, as well as discussions. Although, in the papers of Collaborative Protégé, the need for synchronous and asynchronous development is mentioned as one of the main requirements for ontology engineering, currently only synchronous development is supported. Thus the visualization of detected conflicts is not treated by these approaches in contrast to this paper. Furthermore, ontologies are developed directly in the abstract syntax using a tree editor, thus no concrete syntax conflicts are considered.

# 8 Conclusion

We presented an approach for the visualization of merge conflicts in the context of optimistic model versioning. In contrast to most state-of-the-art model versioning systems, we do not report conflicts on the abstract syntax of a model, but in the concrete syntax as used by the modelers. By using the powerful profile mechanism of UML, we presented a solution which does not require any adoptions of the modeling environment. This allows the modelers to stick to their familiar notation when they are forced to resolve merge conflicts. Then the modeler in charge of conflict resolution may focus on the integration of the changes by interpreting only the changes of the others. In the abstract syntax representation, it would first be necessary to identify

[11] `http://biomedgt.org`

the own changes because the mental map, i.e., the personal view of the modeler on the model, is destroyed.

We comprehensively discussed the technical realization of the proposed approach by introducing a UML profile for representing conflicts and we showed how the information about conflicts may be directly incorporated into any UML diagram. In fact, (semi-)automatic conflict resolution may be offered. Furthermore, we demonstrated that our approach is suitable for reporting various kinds of conflicts.

Our hypothesis is that with this approach the model merging needs less time and is less error prone than model merging on the abstract syntax. The framework presented in this paper provides the technical basis for verifying our hypothesis. We plan to conduct extensive user studies where we will compare the user behavior in different merge scenarios where the conflicts are presented in various styles and the conflict resolution is supported at different levels.

Although we think that our support will result in substantial relaxation of the merge process, we see that there is the potential danger of overloading the model with too much additional information. Therefore, we intend to develop methods in future work for information filtering in order to avoid confusion and disorientation. Additionally, we want to consider dependencies between conflicts in order to realize an assistant for the conflict resolution process which offers precise suggestions how to resolve the conflicts. By this means, we intend to make conflict resolution easier and safer.

## References


[ABK+09] Kerstin Altmanninger, Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. Why Model Versioning Research is Needed!? An Experience Report. In *Proceedings of the Joint MoDSE-MCCM 2009 Workshop @ MoDELS'09*, 2009.

[ADR06] Sören Auer, Sebastian Dietzold, and Thomas Riechert. OntoWiki - A Tool for Social, Semantic Collaboration. In *Proceedings of the 5th International Semantic Web Conference (ISWC'06)*, volume 4273 of *LNCS*, pages 736–749. Springer, 2006. doi:10.1007/11926078_53.

[ASW09] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. A Survey on Model Versioning Approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009. doi:10.1108/17440080910983556.

[Bal91] Robert Balzer. Tolerating Inconsistency. In *Proceedings of the 13th International Conference on Software Engineering (ICSE'91)*, pages 158–165. IEEE, 1991. doi:10.1109/ICSE.1991.130638.

[BE09] Lars Bendix and Pär Emanuelsson. Collaborative Work with Software Models–Industrial Experience and Requirements. In *Proceedings of the 2nd International Conference on Model Based Systems Engineering (MBSE'09)*, pages 2–6, 2009. doi:10.1109/MBSE.2009.5031721.

[BKL+10] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Horst Kargl. Adaptable Model Versioning in Action. In *Proceedings of Modellierung 2010*, volume 161 of *LNI*, pages 221–236. GI, 2010.

[BKL+11a] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. The Past, Present, and Future of Model Versioning. In *Emerging Technologies for the Evolution and Maintenance of Software Models*, chapter 15, pages 410–443. IGI Global, 2011. doi:10.4018/978-1-61350-438-3.ch015.

[BKL+11b] Petra Brosch, Horst Kargl, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Gerti Kappel. Conflicts as First-Class Entities: A UML Profile for Model Versioning. In *Models in Software Engineering - Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers*, volume 6627 of *LNCS*, pages 184–193. Springer, 2011. doi:10.1007/978-3-642-21210-9.

[BKL+12] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. An Introduction to Model Versioning. In *Advanced Lectures of 12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems - Formal Methods for Model-Driven Engineering (SFM'12)*, volume 7320 of *LNCS*, pages 336–398. Springer, 2012. doi:10.1007/978-3-642-30982-3_10.

[BLS+09] 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. doi:10.1007/978-3-642-04425-0_20.

[BP08] Cédric Brun and Alfonso Pierantonio. Model Differences in the Eclipse Modeling Framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.

[CP96] Michael K. Coleman and D. Stott Parker. Aesthetics-based Graph Layout for Human Consumption. *Software: Practice and Experience*, 26(12):1415–1438, 1996. doi:10.1002/(SICI)1097-024X(199612)26:12<1415::AID-SPE69>3.0.CO;2-P.

[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)*, volume 5301 of *LNCS*, pages 311–325. Springer, 2008. doi:10.1007/978-3-540-87875-9_23.

[CRP09] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. Managing dependent changes in coupled evolution. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT'09)*, volume 5563 of *LNCS*, pages 35–51. Springer, 2009. doi:10.1007/978-3-642-02408-5_4.

[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. doi:10.1109/TSE.2008.29.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006. doi:10.1007/3-540-31188-2.

[EET11] Hartmut Ehrig, Claudia Ermel, and Gabriele Taentzer. A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications. In *Proceedings of the 14th International Conference on Fundamental Approaches to Software Engineering (FASE'11)*, volume 6603 of *LNCS*, pages 202–216. Springer, 2011. doi:10.1007/978-3-642-19811-3_15.

[EL11] Maged Elaasar and Yvan Labiche. Diagram Definition: A Case Study with the UML Class Diagram. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS'11)*, volume 6981 of *LNCS*, pages 364–378. Springer, 2011. doi:10.1007/978-3-642-24485-8_26.

[ELF08] Alexander Egyed, Emmanuel Letier, and Anthony Finkelstein. Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE'08)*, pages 99–108. IEEE, 2008. doi:10.1109/ASE.2008.20.

[ELMS91] Peter Eades, Wei Lai, Kazuo Misue, and Kozo Sugiyama. Preserving the Mental Map of a Diagram. In *Proceedings of the 1st International Conference on Computational Graphics and Visualization Techniques*, pages 34–43. ACM, 1991.

[FR07] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Proceedings of the Workshop on Future of Software Engineering @ ICSE'07*, pages 37–54. IEEE, 2007. doi:10.1109/FOSE.2007.14.

[GBPV07] Frank Grimm, Georg Beier, Keith Phalp, and Jonathan Vincent. Towards Semi-Automatic, Mental Map Preserving Visual Merging of UML Class Models. In *Proceedings of the International Conference on Applied Computing*, pages 655–660, 2007.

[GDD09] Dragan Gasevic, Dragan Djuric, and Vladan Devedzic. *Model Driven Engineering and Ontology Development (2. ed.)*. Springer, 2009.

[GDKR+11] Antonio García-Domínguez, Dimitrios Kolovos, Louis Rose, Richard Paige, and Inmaculada Medina-Bulo. EUnit: A Unit Testing Framework for Model Management Tasks. In *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS'11)*, volume 6981 of *LNCS*, pages 395–409. Springer, 2011. doi:10.1007/978-3-642-24485-8_29.

[GE10] Iris Groher and Alexander Egyed. Selective and Consistent Undoing of Model Changes. In *Proceedings of the 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6395 of *LNCS*, pages 123–137. Springer, 2010. doi:10.1007/978-3-642-16129-2_10.

[GJM02] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, 2002.

[JG09] Jendrik Johannes and Karsten Gaul. Towards a Generic Layout Composition Framework for Domain Specific Models. In *Proceedings of 9th Workshop on Domain-Specific Modeling (DSM) @ OOPSLA'09*, 2009.

[JJGT06] Susanne Jucknath-John, Dennis Graf, and Gabriele Taentzer. Evolutionary Layout of Graph Transformation Sequences. *Electronic Communications of the EASST*, 1, 2006.

[KHvWH10] Maximilian Kögel, Markus Herrmannsdoerfer, Otto von Wesendonk, and Jonas Helming. Operation-based Conflict Detection. In *Proceedings of the 1st International Workshop on Model Comparison in Practice @ TOOLS'10*, pages 21–30. ACM, 2010. doi:10.1145/1826147.1826154.

[KKT11] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. A Rule-Based Approach to the Semantic Lifting of Model Differences in the Context of Model Versioning. In *Proceedings of the 26th International Conference on Automated Software Engineering (ASE'11)*, pages 163–172, 2011. doi:10.1109/ASE.2011.6100050.

[Kol09] Dimitrios Kolovos. Establishing Correspondences between Models with the Epsilon Comparison Language. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'09)*, volume 5562 of *LNCS*, pages 146–157. Springer, 2009. doi:10.1007/978-3-642-02674-4_11.

[KRDM+10] Dimitrios Kolovos, Louis Rose, Nikolaos Drivalos Matragkas, Richard Paige, Fiona Polack, and Kiran Fernandes. Constructing and Navigating Non-invasive Model Decorations. In *Proceedings of the 3rd International Conference on Theory and Practice of Model Transformations (ICMT'10)*, volume 6142 of *LNCS*, pages 138–152. Springer, 2010. doi:10.1007/978-3-642-13688-7_10.

[LEO06] Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Conflict Detection for Graph Transformation with Negative Application Conditions. In *Proceedings of the 3rd International Conference on Graph Transformations (ICGT'06)*, volume 4178 of *LNCS*, pages 61–76. Springer, 2006. doi:10.1016/j.tcs.2012.01.032.

[LGJ07] Y. Lin, J. Gray, and F. Jouault. DSMDiff: A Differentiation Tool for Domain-specific Models. *European Journal of Information Systems*, 16(4):349–361, 2007. doi:10.1057/palgrave.ejis.3000685.

[LWWC11] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. From UML Profiles to EMF Profiles and Beyond. In *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns (TOOLS'11)*, volume 6705 of *LNCS*, pages 52–67. Springer, 2011. doi:10.1007/978-3-642-21952-8_6.

[LWWC12] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. Emf profiles: A lightweight extension approach for emf models. *Journal of Object Technology*, 11(1):1–29, 2012. doi:10.5381/jot.2012.11.1.a8.

[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*, pages 25–30. ACM, 2008. doi:10.1145/1370152.1370159.

[MD94] Jonathan P. Munson and Prasun Dewan. A Flexible Object Merging Framework. In *Proceedings of the International Conference on Computer Supported Cooperative Work (CSCW'94)*, pages 231–242. ACM, 1994. doi:10.1145/192844.193016.

[MD10] Frédéric Madiot and Grégoire Dupé. EMF Facet: A Non-Intrusive Tooling to Extend Metemodels. http://www.eclipse.org/modeling/emft/facet/, 2010.

[Men99] Tom Mens. *A Formal Foundation for Object-Oriented Software Evolution*. PhD thesis, Vrije Universiteit Brussel, 1999.

[Men02] Tom Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002. doi:10.1109/TSE.2002.1000449.

[Mey92] Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992. doi:10.1109/2.161279.

[MGH05] Akhil Mehra, John Grundy, and John Hosking. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE'05)*, pages 204–213. ACM, 2005. doi:10.1145/1101908.1101940.

[NER01] Bashar Nuseibeh, Steve M. Easterbrook, and Alessandra Russo. Making Inconsistency Respectable in Software Development. *Journal of Systems and Software*, 58(2):171–180, 2001. doi:10.1016/S0164-1212(01)00036-X.

[Nie04] Jörg Niere. Visualizing Differences of UML Diagrams With Fujaba. In *Proceedings of the International Fujaba Days 2004*, 2004.

[Obj06] Object Management Group. UML Diagram Interchange, Version 1.0. http://www.omg.org/spec/UMLDI/1.0/, April 2006.

[Obj11a] Object Management Group. Diagram Definition (DD). http://www.omg.org/spec/DD/1.0/Beta2/, July 2011.

[Obj11b] Object Management Group. OMG Meta Object Facility (MOF) Core Specification V2.4.1. http://www.omg.org/spec/MOF/2.4.1/, August 2011.

[Obj11c] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure V2.4.1. http://www.omg.org/spec/UML/2.4.1/, August 2011.

[Obj12] Object Management Group. OMG Object Constraint Language (OCL). http://www.omg.org/spec/OCL/2.3.1/, January 2012.

[OS05] Takafumi Oda and Motoshi Saeki. Generative Technique of Version Control Systems for Software Diagrams. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 515–524. IEEE, 2005. doi:10.1109/ICSM.2005.49.

[OWK03] Dirk Ohst, Michael Welle, and Udo Kelter. Differences between Versions of UML Diagrams. In *Proceedings of the 9th European Software Engineering Conference (ESEC'03)*, pages 227–236. ACM, 2003. doi:10.1145/949952.940102.

[SK88] Robert W. Schwanke and Gail E. Kaiser. Living With Inconsistency in Large Systems. In *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 98–118. Teubner B.G. GmbH, 1988. doi:10.1109/32.310667.

[SMB09] Ragnhild Van Der Straeten, Tom Mens, and Stefan Van Baelen. Challenges in Model-Driven Software Engineering. In *Models in Software Engineering - Workshops and Symposia at MODELS 2009, Reports and Revised Selected Papers*, volume 6002 of *LNCS*, pages 35–47. Springer, 2009. doi:10.1007/978-3-642-01648-6_4.

[SNTM08] Abraham Sebastian, Natalya Fridman Noy, Tania Tudorache, and Mark A. Musen. A Generic Ontology for Collaborative Ontology-Development Workflows. In *Proceedings of the 16th International Conference on Knowledge Engineering and Knowledge Management (EKAW'08)*, volume 5268 of *LNCS*, pages 318–328. Springer, 2008. doi:10.1007/978-3-540-87696-0_28.

[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.

[TELW10] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. Conflict Detection for Model Versioning Based on Graph Modifications. In *Proceedings of the 5th International Conference on Graph Transformations (ICGT'10)*, volume 6372 of *LNCS*, pages 171–186. Springer, 2010. doi:10.1007/978-3-642-15928-2_12.

[TNTM08] Tania Tudorache, Natalya Fridman Noy, Samson W. Tu, and Mark A. Musen. Supporting Collaborative Ontology Development in Protégé. In *Proceedings of the 7th International Semantic Web Conference (ISWC'08)*, volume 5318 of *LNCS*, pages 17–32. Springer, 2008. doi:10.1007/978-3-540-88564-1_2.

[vP07] Jens von Pilgrim. Mental Map and Model Driven Development. *Electronic Communications of the EASST*, 7, 2007.

[WS06] Kenny Wong and Dabo Sun. On Evaluating the Layout of UML Diagrams for Program Comprehension. *Software Quality Control*, 14(3):233–259, 2006. doi:10.1109/WPC.2005.26.

[XS05] Zhenchang Xing and Eleni Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th International Conference on Automated Software Engineering (ASE'05)*, pages 54–65. ACM, 2005. doi:10.1145/1101908.1101919.

[ZGH+07] Nianping Zhu, John Grundy, John Hosking, Na Liu, Shuping Cao, and Akhil Mehra. Pounamu: A Meta-Tool for Exploratory Domain-Specific Visual Language Tool Development. *Journal of Systems and Software*, 80(8):1390–1407, 2007. doi:10.1016/j.jss.2006.10.028.

## About the authors

**Petra Brosch** is a post-doc researcher at the Business Informatics Group of the Vienna University of Technology. Her research interests include various topics in the area of model-driven engineering, especially model evolution, model versioning, and model transformation. In her PhD thesis, she worked on conflict resolution in model versioning with special emphasis on enabling merge support directly in the concrete syntax of models and recommending conflict resolution patterns.

**Martina Seidl** holds a PhD in computer science and works at the Business Informatics Group of the Vienna University of Technology and the Institute for Formal Models and Verification of the Johannes Kepler University Linz. Her research interests include various topics from the area of model evolution and model versioning, automated reasoning with special focus on the evaluation of quantified Boolean formulas as well as software verification.

**Manuel Wimmer** is working as a post-doc researcher at the Business Informatics Group of the Vienna University of Technology. His research interests comprise Web engineering and model engineering; in particular model transformations based on formal methods, generating transformations by-example as well as applying model transformations to deal with model (co-)evolution. Currently, he is on leave working as visiting researcher at the Software Engineering Group of the University of Málaga (Spain), where his research focuses on model-driven evolution support for Web applications.

**Gerti Kappel** is a full professor at the Institute of Software Technology and Interactive Systems at the Vienna University of Technology, heading the Business Informatics Group. Until 2001, she was a full professor of computer science and head of the Department of Information Systems at the Johannes Kepler University of Linz. She received the Ms and PhD degrees in computer science and business informatics from the University of Vienna and the Vienna University of Technology in 1984 and 1987, respectively. From 1987 to 1989 she was a visiting researcher at the Centre Universitaire d'Informatique, Geneva, Switzerland. Her current research interests include model engineering (model transformation, model versioning), Web engineering (ubiquitous Web technologies, model-driven Web engineering), as well as process engineering (business process modeling and transformation).

**Acknowledgments** This work has been funded by the Vienna Science and Technology Fund (WWTF) through project ICT10-018, by the Austrian Science Fund (FWF) under grant J 3159-N23, and by the fFORTE WIT Program of the Vienna University of Technology and the Austrian Federal Ministry of Science and Research.