An Introduction to Model Versioning*

Petra Brosch[1], Gerti Kappel[1], Philip Langer[1],
Martina Seidl[2], Konrad Wieland[3], and Manuel Wimmer[4]

[1] Vienna University of Technology, Austria
{brosch,gerti,langer}@big.tuwien.ac.at
[2] Johannes Kepler University Linz, Austria
martina.seidl@jku.at
[3] LieberLieber Software GmbH, Austria
konrad.wieland@lieberlieber.com
[4] Universidad de Málaga, Spain
mw@lcc.uma.es

Abstract. With the emergence of model-driven engineering (MDE), *software models* are considered as central artifacts in the software engineering process, going beyond their traditional use as sketches. In MDE, models rather act as the single source of information for automatically generating executable software. This shift poses several new research challenges. One of these challenges constitutes *model versioning*, which targets at enabling efficient team-based development of models. This compelling challenge induced a very active research community, who yielded remarkable methods and techniques ranging from model differencing to merging of models.

In this tutorial, we give an introduction to the foundations of model versioning, the underlying technologies for processing models and their evolution, as well as the state of the art in model versioning. Thereby, we aim at equipping students and researchers alike that are new to this domain with enough information for commencing to contribute to this challenging research area.

1 Introduction

Since the emergence of *software engineering* [72,94], researchers and practitioners have been struggling to cope with the ever growing *complexity* and *size* of the developed systems. One way of coping with the complexity of a system has been raising the level of abstraction in the languages used to specify a system. Besides dealing with the complexity of software systems under development, also managing the size of software systems constitutes a major challenge. As stated by Ghezzi et al., "software engineering deals with the building of software systems that are so large or so complex that they are built by teams of engineers" [37].

* 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.

M. Bernardo, V. Cortellessa, and A. Pierantonio (Eds.): SFM 2012, LNCS 7320, pp. 336–398, 2012.
© Springer-Verlag Berlin Heidelberg 2012

Orthogonal to the challenge entailed by the complexity and size of software systems, dealing with the demand to constantly *evolve* a system in order to meet ever changing requirements constitutes an additional major challenge. To summarize, Parnas defines software engineering as the "multi-person construction of multi-version software" [92].

Model-driven engineering (MDE) has been proposed as a new paradigm for raising the level of abstraction [7,38,98]. In MDE, softare models are considered as central artifacts in the software engineering process, going beyond their traditional use as sketches and blueprints. Models constitute the basis and the single source of information to specify and automatically generate an executable system. Thereby, developers may build models that are less bound to an underlying implementation technology and are much closer to the problem domain [103]. However, the emergence of this shift from code to models poses several new research challenges. One of these challenges is to cope with the ever growing *size* of systems being built in practice [34]. Developing a large system entails the need for a large number of developers who collaborate to succeed in creating a large system. Thus, adequate support for *team-based development of models* is a crucial prerequisite for the success of MDE. Therefore, as in traditional code-centric software engineering, *versioning systems* [18,66] are required, which allow for concurrent modification of the same model by several developers and which are capable of merging the operations applied by all developers to obtain ultimately one consolidated version of a model again.

In traditional code-centric software engineering, text-based versioning systems, such as Git[1], Subversion[2], and CVS[3], have been successfully deployed to allow for collaborative development of large software systems. To enable *collaborative modeling* among several team members, such text-based versioning systems have been reused for models. Unfortunately, it turned out quickly that applying text-based versioning is inadequate for models and leads to unsatisfactory results [2]. This is because such versioning systems consider only text lines in a text-based representation of a model as, for instance, the XMI serializations [81]. As a result, the information stemming from the model's graph-based structure is destroyed and associated syntactic information is lost. To overcome these drawbacks of text-based versioning systems used for models, dedicated *model versioning* approaches have been proposed recently. Such approaches do not operate on the textual representation; instead, they work directly on the model's graph-based structure.

Especially *optimistic versioning systems* gained remarkable popularity because they enable several developers to work concurrently on the same artifacts instead of pessimistically locking each artifact for the time it is changed by one developer. The price to pay for being able to work in parallel is that the concurrently applied operations of all developers have to be merged again. Therefore, the *versioning process* depicted in Fig. 1 is applied, which is referred to

[1] `http://git-scm.com`
[2] `http://subversion.tigris.org`
[3] `http://cvs.nongnu.org`

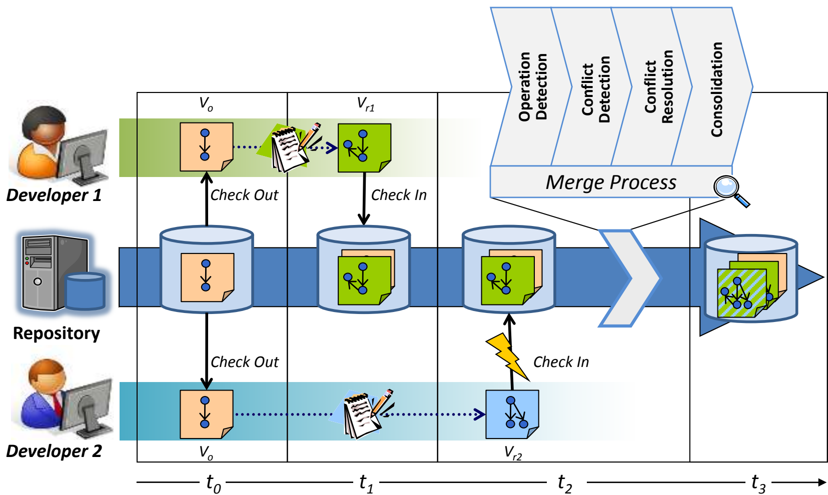


Fig. 1. Versioning Process

as *check-out/check-in protocol* [30]. Developers may concurrently check-out the latest version V_o of a model from a common repository at the time of t_0 (cf. Fig. 1). Thereby, a local working copy of V_o is created. Both developers may independently modify their working copies in parallel. As soon as one developer completes the work, assume this is developer 1, she performs a check-in at t_1. Because no other developer performed a check-in in the meanwhile, her working copy can be saved directly as a new revised version V_{r1} in the repository. Whenever developer 2 completes his task and performs the check-in, the versioning system recognizes that a new version has been created since the check-out. Therefore, the merge process is triggered at t_2 in order to merge the new version V_{r1} in the repository with the version V_{r2} by developer 2. Once the merge is carried out, the resulting merged version incorporating the operations of developer 1 and developer 2 is saved in the repository.

In this tutorial, we give an introduction to the underlying technologies for realizing this versioning process to allow for merging concurrently modified models. Therefore, we discuss the foundations of versioning in Section 2 and introduce the prerequisites for building a model versioning system in Section 3. Subsequently, we review the state of the art in model versioning in Section 4 and present our own model versioning system AMOR in Section 5. Finally, we conclude this tutorial with some challenging topics for future research in the domain of model versioning in Section 6.

2 Foundations of Versioning

The history of versioning in software engineering goes back to the early 1970s. Since then, software versioning was constantly an active research topic. As stated

by Estublier et al. in [30], the goal of software versioning systems is twofold. First, such systems are concerned with maintaining a historical archive of a set of artifacts as they undergo a series of operations and form the fundamental building block for the entire field of Source Configuration Management (SCM), which deals with controlling change in large and complex software systems. Second, versioning systems aim at managing the evolution of software artifacts performed by a distributed team of developers.

In that long history of research on software versioning, diverse formalisms and technologies emerged. To categorize this variety of different approaches, Conradi and Westfechtel [18] proposed *version models* describing the diverse characteristics of existing versioning approaches. A version model specifies the objects to be versioned, version identification and organization, as well as operations for retrieving existing versions and constructing new versions. Conradi and Westfechtel distinguish between the *product space* and the *version space* within version models. The product space describes the structure of a software product and its artifacts without taking versions into account. In contrast, the version space is agnostic of the artifacts' structure and copes with the artifacts' evolution by introducing versions and relationships between versions of an artifact, such as, for instance, their differences (deltas). Further, Conradi and Westfechtel distinguish between extensional and intentional versioning. *Extensional versioning* deals with the reconstruction of previously created versions and, therefore, concerns version identification, immutability, and efficient storage. All versions are explicit and have been checked in once before. *Intentional versioning* deals with flexible automatic construction of consistent versions from a version space. In other words, intentional versioning allows for annotating properties to specific versions and querying the version space for these properties in order to derive a new product consisting of a specific combination of different versions.

In this paper, we only consider extensional versioning in terms of having explicit versions, because this kind of versioning is predominantly applied in practice nowadays. Furthermore, we focus on the *merge phase* in the optimistic versioning process (cf. Fig. 1). In this section, we outline the fundamental design dimensions of versioning systems and elaborate on the consequences of different design decisions concerning the quality of the merge based on an example.

2.1 Fundamental Design Dimensions for Versioning Systems

Current approaches to merging two versions of one software artifact (software models or source code) can be categorized according to two basic dimensions. The first dimension concerns the product space, in particular, the *artifact representation*. This dimension denotes the representation of a software artifact, on which the merge approach operates. The used representation may either be *text-based* or *graph-based*. Some merge approaches operate on a tree-based representation. However, we consider a tree as a special kind of graph in this categorization. The second dimension is orthogonal to the first one and concerns how deltas are *identified*, *represented*, and *merged* in order to create a consolidated version. Existing merge approaches either operate on the *states*; that is, the versions of an

artifact, or on identified operations that have been applied between a common origin model (cf. V_o in Fig. 1) and the two successors (cf. V_{r1} and V_{r2} in Fig. 1).

When merging two concurrently modified versions of a software artifact, conflicts might inevitably occur. The most basic types of conflicts are *update-update* and *delete-update* conflicts. Update-update conflicts occur if two elements have been updated in both versions whereas delete-update conflicts are raised if an element has been updated in one version and deleted in the other (cf. [66] for more information on software merging in general).

Text-based merge approaches operate solely on the textual representation of a software artifact in terms of text files. Within a text file, the atomic unit of the versioned text file may either be a paragraph, a line, a word, or even an arbitrary set of characters. The major advantage of such approaches is their independence of the programming languages used in the versioned artifacts. Since a solely text-based approach does not require language-specific knowledge it may be adopted for all flat text files. This advantage is probably, besides simplicity and efficiency, the reason for the widespread adoption of pure text-based approaches in practice. However, when merging flat files—agnostic of the syntax and semantics of a programming language—both compile-time and run-time errors might be introduced during the merge. Therefore, graph-based approaches emerged, which take syntax and semantics into account.

Graph-based merge approaches operate on a graph-based representation of a software artifact for achieving more precise conflict detection and merging. Such approaches de-serialize or translate the versioned software artifact into a specific structure before merging. Mens [66] categorized these approaches in *syntactic* and *semantic merge approaches*. Syntactic merge approaches consider the syntax of a programming language by, for instance, translating the text file into the abstract syntax tree and, subsequently, performing the merge in a syntax-aware manner. Consequently, unimportant textual conflicts, which are, for instance, caused by reformatting the text file, may be avoided. Furthermore, such approaches may also avoid syntactically erroneous merge results. However, the textual formatting intended by the developers might be obfuscated by syntactic merging because only a graph-based representation of the syntax is merged and has to be translated back to text eventually. Westfechtel was among the first to propose a merging algorithm that operates on the abstract syntax tree of a software artifact [116]. Semantic merge approaches go one step further and consider also the static and/or dynamic semantics of a programming language. Therefore, these approaches may also detect issues, such as undeclared variables or even infinite loops by using complex formalisms like program dependency graphs and program slicing. Naturally, these advantages over flat textual merging have the disadvantage of the inherent language dependence (cf. [66]) and their increased computational complexity.

The second dimension for categorizing versioning systems is orthogonal to the first one and considers *how deltas are identified and merged* in order to create a consolidated version. This dimension is agnostic of the unit of versioning.

Therefore, a versioned element might be a line in a flat text file, a node in a graph, or whatsoever constitutes the representation used for merging.

State-based merging compares the states (i.e., versions) of a software artifact to identify the differences (deltas) between these versions and merge all differences that are not contradicting with each other. Such approaches may either be applied to two states (V_{r1} and V_{r2} in Fig. 1), called two-way merging, or to three states (including their common ancestor V_o in Fig. 1), called three-way merging. Two-way merging cannot identify deletions since the common original state is unknown. A state-based comparison requires a match function which determines whether two elements of the compared artifact correspond to each other. The easiest way to match two elements is to search for completely equivalent elements. However, the quality of the match function is crucial for the overall quality of the merge approach. Therefore, especially graph-based merge approaches often use more sophisticated matching techniques based on identifiers and heuristics (cf. [47] for an overview of matching techniques). Model matching, or more generally the graph isomorphism problem is NP-hard [46] and, therefore, very computation intensive. If the match function is capable of matching also partially different elements, a difference function is additionally required to determine the fine-grained differences between two corresponding elements. Having these two functions, two states of the same artifact may be merged by using the following process. For each element in the common origin version V_o of a software artifact, the corresponding elements from the two modified versions V_{r1} and V_{r2} are retrieved. If in both versions V_{r1} and V_{r2} a corresponding element is available, the algorithm checks whether the matching element has been modified in the versions V_{r1} and V_{r2}. If this is true in one and only one of the two versions V_{r1} and V_{r2}, the modified element is used for creating the merged version. If, however, the matching element is different in *both versions*, an update-update conflict is raised by the algorithm. If the matching element has not been modified at all, the original element can be left as it is in the merged version. In case there is no corresponding element in one of the two modified versions (i.e., it has been removed), it is checked whether it has been concurrently modified in the opposite revision and raises, in this case, a delete-update conflict. If the element has not been concurrently modified, it is removed from the merged version. The element is also removed, if there is no corresponding element in both modified versions (i.e., it has been deleted in both versions). Finally, the algorithm adds all elements from V_{r1} and V_{r2} that have no corresponding element in the original version V_o, as they have been added in V_{r1} or V_{r2}.

Operation-based merging does not operate on the states of an artifact. Instead, the operation sequences which have been concurrently applied to the original version are recorded and analyzed. Since the operations are directly recorded by the applied editor, operation-based approaches may support, besides recording atomic operations, also to record composite operations, such as refactorings (e.g., [48]). The knowledge on applied refactorings may significantly increase the quality of the merge as stated by Dig et al. [22]. The downside of operation recording is the strong dependency on the applied editor, since it has to record each performed operation

and it has to provide this operation sequence in a format which the merge approach is able to process. The directly recorded operation sequence might include obsolete operations, such as updates to an element which will be removed later on. Therefore, many operation-based approaches apply a cleansing algorithm to the recorded operation sequence for more efficient merging. The operations within the operation sequence might be interdependent because some of the operations cannot be applied until other operations have been applied. As soon as the operation sequences are available, operation-based approaches check parallel operation sequences (V_o to V_{r1} and V_o to V_{r2}) for commutativity to reveal conflicts [60]. Consequently, a decision procedure for commutativity is required. Such decision procedures are not necessarily trivial. In the simplest yet least efficient form, each pair of operations within the cross product of all atomic operations in both sequences are applied in both possible orders to the artifact and both results are checked for equality. If they are not equivalent, the operations are not commutative. After checking for commutativity, operation-based merge approaches apply all non-conflicting (commutative) operations of both sides to the common ancestor in order to obtain a merged model.

In comparison to state-based approaches, the recorded operation sequences are, in general, more precise and potentially allow for gathering more information (e.g., change order and refactorings), than state-based differencing; especially if the state-based approach does not rely on a precise matching technique. Moreover, state-based comparison approaches are—due to complex comparison algorithms—very expensive regarding their run-time in contrast to operation-based change recording. However, these advantages come at the price of strong editor-dependence. Nevertheless, operation-based approaches scale for large models from a conceptual point of view because their computational effort mainly depends on the length of the operation sequences and—in contrast to state-based approaches—not on the size of the models [48].

Anyhow, the border between state-based and operation-based merging is sometimes blurry. Indeed, we can clearly distinguish whether the operations are recorded or differences are derived from the states, nevertheless, some *state-based approaches* derive the *applied operations* from the states and use operation-based conflict detection techniques. However, this is only reasonable if a reliable matching function is available, for instance, using unique identifiers. On the contrary, some *operation-based approaches* derive the *states* from their operation sequences to check for potentially inconsistent states after merging. Such an inconsistent state might for instance be a violation of the syntactic rules of a language. Detecting such conflicts is often not possible by solely analyzing the operation sequences. Eventually, the conflict detection strategies conducted in state-based and operation-based approaches are very similar from a conceptual point of view. Both check for direct or indirect concurrent modifications to the same element and try to identify illegal states after merging, whether the modifications are explicitly given in terms of operations or whether they are implicitly derived from a match between two states.

2.2 Consequences of Design Decisions

To highlight the benefits and drawbacks of the four possible combinations of the versioning approaches (text-based vs. graph-based and state-based vs. operation-based), we present a small versioning example depicted in Fig. 2 and conceptually apply each approach for analyzing its quality in terms of the detected conflicts and derived merged version.

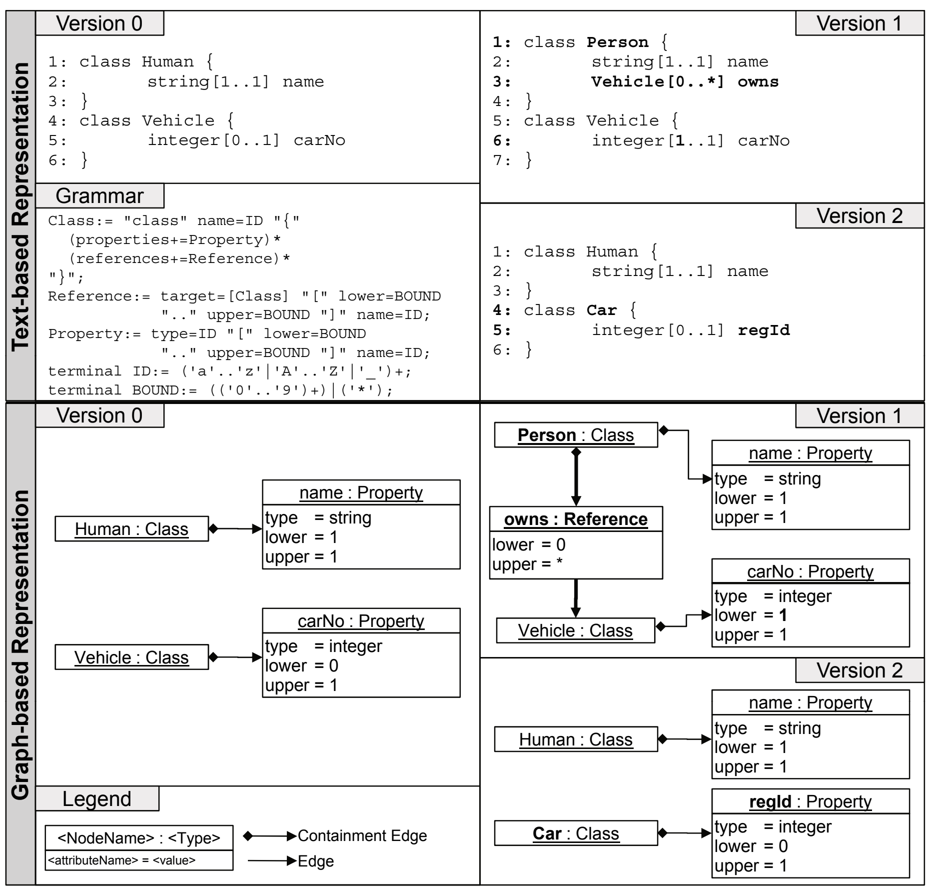
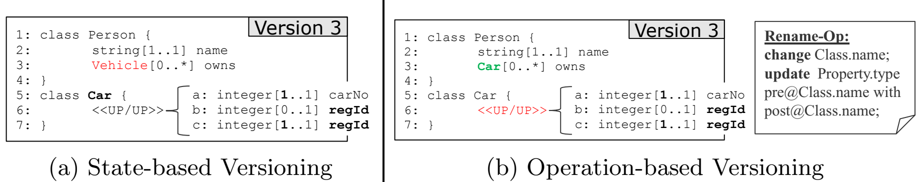


Fig. 2. Versioning Example

Consider a small language for specifying *classes*, its *properties*, and *references* linking two classes. The textual representation of this language is depicted in the upper left area of Fig. 2 and defined by the EBNF-like Xtext[4] grammar specified in the box labeled *Grammar*. The same language and the same examples are depicted in terms of graphs in the lower part of Fig. 2. In the initial version

[4] `http://www.eclipse.org/Xtext`



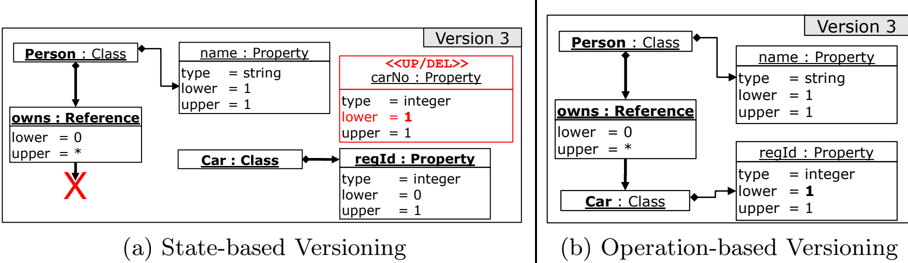
(a) State-based Versioning (b) Operation-based Versioning

Fig. 3. Text-based Versioning Example

(Version 0) of the example, there are two classes, namely Human and Vehicle. The class Human contains a property name and the class Vehicle contains a property named carNo. Now, two users concurrently modify Version 0 and create Version 1 and Version 2, respectively. All operations in Version 1 and Version 2 are highlighted with bold fonts or edges in Fig. 2. The first user changes the name of the class Human to Person, sets the lower bound of the property carNo to 1 (because every car must have exactly one number) and adds an explicit reference owns to Person. Concurrently, the second user renames the property carNo to regId and the class Vehicle to Car.

Text-based versioning. When merging this example with *text- and state-based* approaches (cf. Fig. 3a for the result) where the artifact's representation is a single line and the match function only matches completely equal lines (as with Subversion, CVS, Git, and bazaar), the first line is correctly merged since it has only been modified in Version 1 and remained untouched in Version 2. The same is true for the added reference in line 3 of Version 1 and the renamed class Car in line 4 of Version 2. However, the property carNo shown in line 5 in Version 0 has been changed in both Versions 1 (line 6) and Version 2 (line 5). Although different features of this property have been modified (lower and name), these modifications result in a concurrent change of the same line and, hence, a conflict is raised. Furthermore, the reference added in Version 1 refers to class *Vehicle*, which does not exist in the merged version anymore since it has been renamed in Version 2. We may summarize that text- and state-based merging approaches provide a reasonable support for versioning software artifacts. They are easy to apply and work for every kind of flat text file irrespectively of the used language. However, erroneous merge results may occur and several "unnecessary" conflicts might be raised. The overall quality strongly depends on the textual syntax. Merging textual languages with a strict syntactic structure (such as XML) might be more appropriate than merging languages which mix several properties of potentially independent concepts into one line. The latter might cause tedious manual conflict and error resolution.

One major problem in the merged example resulting from text-based and state-based approaches is the wrong reference target (line 3 in Version 1) caused by the concurrent rename of Vehicle. *Operation-based approaches* (such as MolhadoRef) solve such an issue by incorporating knowledge on applied refactorings in the merge. Since a *rename* is a refactoring, MolhadoRef would be aware of



(a) State-based Versioning (b) Operation-based Versioning

Fig. 4. Graph-based Versioning Example

the rename and resolve the issue by re-applying the rename after a traditional merge is done. The result of this merge is shown in Fig. 3b.

Graph-based versioning. Applying the merge on top of the *graph-based representation* depicted in Fig. 2 may also significantly improve the merge result because the representation used for merging is a node in a graph which more precisely represents the versioned software artifact. However, as already mentioned, this advantage comes at the price of language dependence because merging operates either on the language specific graph-based representation or a translation of a language to a generic graph-based structure must be available. *Graph- and state-based approaches* additionally require a match function for finding corresponding nodes and a difference function for explicating the differences between matched nodes. The preciseness of the match function significantly influences the quality of the overall merge. Assume matching is based on name and structure heuristics for the example in Fig. 2. Given this assumption, the class Human may be matched since it contains an unchanged property name. Therefore, renaming the class Human to Person can be merged without user intervention. However, heuristically matching the class Vehicle might be more challenging because both the class and its contained property have been renamed. If the match does not identify the correspondence between Vehicle and Car, Vehicle and its contained property carNo is considered to be removed and Car is assumed to be added in Version 2. Consequently, a delete-update conflict is reported for the change of the lower bound of the property carNo in Version 1. Also the added reference owns refers to a removed class which might be reported as conflict. This type of conflict is referred to as *delete-use* or *delete-reference* in literature [110,117]. If, in contrast, the match relies on unique identifiers, the nodes can soundly be matched. Based on this precise match, the state-based merge component can resolve this issue and the added reference owns correctly refers to the renamed class Car in the merged version. However, the concurrent modification of the property carNo (name and lower) might still be a problem because purely state-based approaches usually take either the entire element from either the left or the right version to construct the merged version. Some state-based approaches solve this issue by conducting a more fine-grained difference function to identify the detailed

differences between two elements. If these differences are not overlapping—as in our example—they can both be applied to the merged element. The result of a graph-based and state-based merge without taking identifiers into account is visualized in Fig. 4a.

Purely *graph- and operation-based approaches* are capable of automatically merging the presented example (cf. Fig. 4b). Between Version 0 and Version 1, three operations have been recorded, namely the rename of Human, the addition of the reference owns and the update concerning the lower bound of carNo. To get Version 2 from Version 0, class Vehicle and property carNo have been renamed. All these atomic operations do not interfere and therefore, they all can be re-applied to Version 0 to obtain a correctly merged version.

In summary, a lot of research activity during the last decades in the domain of traditional source code versioning has lead to significant results. Approaches for merging *software models* draw a lot of inspiration from previous works in the area of *source code* merging. Especially graph-based approaches for source code merging form the foundation for model versioning. However, one major challenge still remains an open problem. The same trade-off as in traditional source code merging has to be made regarding editor- and language-independence versus preciseness and completeness. Model matching, comparison and merging, as discussed above, can significantly be improved by incorporating knowledge on the used modeling language, as well as language-specific composite operations, such as refactorings. On the other hand, model versioning approaches are also forced to support several languages at the same time, because even in small MDE projects several modeling languages are usually combined.

3 Five Steps towards Model Versioning

In this section, we survey the fundamental techniques for stepwise establishing versioning support for software models. Therefore, we introduce the basics of model-driven engineering, model transformations, model differencing, conflicts, and merging in the following.

3.1 Model-Driven Engineering

The idea of MDE is to automate the repetitive task of translating model based blueprints to code and enable developers to concentrate on creative and non-trivial tasks which computers cannot do, i.e., creating those blueprints [10]. Several techniques are indispensable for putting MDE into practice. In the following, we introduce a common framework for creating domain-specific modeling languages and models called *metamodeling*.

An early attempt towards MDE was made in the 1980s with computer-aided software engineering (CASE) tools, already following the goal to directly generate executable systems based on graphical domain-specific models [98]. As CASE tools were (1) costly to develop, and (2) only appropriate to certain domains, it was soon realized that the development of domain-specific environments was

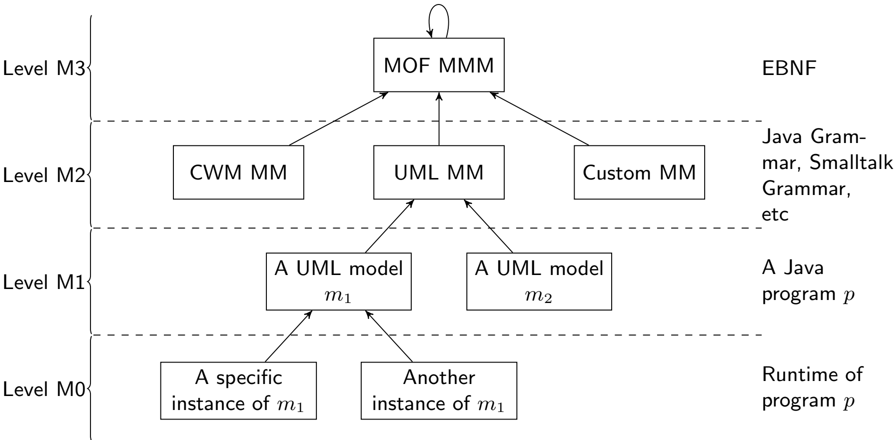


Fig. 5. Metamodeling Layers; adapted from [9]

itself a domain and metamodeling environments to create domain-specific environments were established [58,106,71]. In fact, the term *meta* denotes that an operation is applied on itself, e.g., a discussion about conducting a discussion is called meta-discussion [53]. In a similar vein, metamodeling is referred to modeling modeling languages.

In an endeavor to establish a commonly accepted set of key concepts and to preserve interoperability between the rapidly growing number of domain-specific development environments, the Object Management Group (OMG) released the specification for *Model Driven Architecture* (MDA) [79], standardizing the definition and usage of (meta-)metamodels as driving factor of software development. To this end, the OMG proposes a layered organization of the metamodeling stack similar to the architecture of formal programming languages [9], as depicted in Fig. 5. The meta-metamodel level M3 manifests the role of the *Meta-Object Facility* (MOF) [77] as the unique and self-defined metamodel for building metamodels, i.e., the meta-metamodel ensuring interoperability of any metamodel defined therewith. Every metamodel defined according OMG's proposed MDA standard share the same metatypes and may be reflectively analyzed. MOF may be compared to the Extended Backus-Naur Form (EBNF) [42], the metagrammar for expressing programming languages. The metamodel level M2 contains any metamodel defined with MOF, including the Unified Modeling Language (UML) [86], the Common Warehouse Metamodel (CWM) [76] also specified by the OMG, and any custom domain-specific metamodel. A metamodel at this level conforms to a definition of a programming language with EBNF, such as the Smalltalk grammar or the Java grammar. A metamodel defines the abstract syntax of the modeling language and is usually supplemented with one or more concrete syntactics. Though textual concrete syntactics get more and more popular, graphical concrete syntactics are more common. To leverage MOF's interoperability power also for the graphical visualization of models, a graphical concrete syntax is again defined as metamodel at level M2. The metamodel for

the concrete syntax defines graphical elements such as shapes and edges by extending a standardized diagram interchange metamodel [75,80] and associates those elements with corresponding elements of the abstract syntax metamodel. The model level M1 contains any model built with a metamodel of level M2, e.g., a UML model. An equivalent for a model is a specific program written in any programming language defined in EBNF. Finally, the concrete level M0 reflects any model based representation of a real situation. This representation is an instance of a model defined in level M1. We may again draw the parallel to the formal programming language architecture. Level M0 corresponds to all dynamic execution traces of a program of level M1.

The major benefit of MDA is to decouple system specifications from the underlying platform [14]. In this way, the specification is much closer to the problem domain and not bound to a specific implementation technique. This benefit is maximized when domain-specific modeling languages are employed. Thus, the MDA specification [79] differentiates at level M2 languages for *Computation Independent Model* (CIM), *Platform Independent Model* (PIM), and *Platform Specific Model* (PSM) to quantify the abstraction quality of a model. While a CIM provides a fully computation independent viewpoint close to the domain in question, a PIM approximates the system description in a technology neutral manner. A PSM eventually unifies the PIM with the specifics of the underlying platform to be used as specification for code generation.

To bridge metamodeling and programming languages, and to justify MOF as interoperability standard, the OMG provides a standardized way for exchanging MOF based artifacts. OMG's standard for *XML Metadata Interchange* (XMI) [81] defines a mapping of any (meta)model expressed in MOF to the *Extensible Markup Language* (XML) [114]. The specification of MOF itself is divided into the kernel metamodel *Essential MOF* (EMOF) and the more expressive *Complete MOF* (CMOF) [77]. EMOF is closely related to object-oriented programming languages, such as Java, which allows a straightforward mapping, as implemented in [91,23]. Especially the Eclipse Modeling Framework (EMF) with its reference implementations for EMOF [23] and UML [24] fosters several adjacent subprojects for arbitrary MDA tasks, such as querying and comparing models, building textual and graphical modeling editors, etc. leading to increasing adoption in academia and in practice.

3.2 Model Transformation

In modern software engineering practice employing the MDE paradigm, a vast amount of interrelated models accrues. Those models are in the first place utilized to gain abstraction of the technical realization of a system, such that developers may completely concentrate on a specific matter of the problem domain and serve finally as construction plan for implementation. To free developers from the burden of repetitive and error-prone tasks such as translating models into source code and propagating changes throughout dependent models, a mechanism to transform and synchronize models is demanded. The field of *model transformation* accepts to play this central role and is thus noticed as the heart and soul

of MDE [104]. To embrace the diversity of domain-specific modeling languages the transformable models conform to, a model transformation language usually makes use of the flexible type-system gained by metamodeling and defines a fixed syntax for defining transformation rules only. The metamodels holding the actual types of elements to be transformed are bound not until a specific model transformation is specified [8]. Reflecting the plethora of application areas of models, model transformation tasks cover all sorts of activities, such as updating, synchronizing, translating models to other models, or translating models to code. To support these activities, a multitude of either general or specifically tailored model transformation languages and approaches emerged.

In the following, we briefly present distinguishing characteristics of current model transformation approaches. However, those approaches may not only be adopted for one specific task. As shown in [27], exogenous model transformations may be also achieved with tools primarily built for endogenous model transformations. Conversely, tools for exogenous model transformations may perform endogenous model transformations by defining transformations with equal input metamodel and output metamodel. Even bidirectional model transformations may be achieved by defining one transformation for each direction. Although this interchange is possible, defining transformations is much easier and safer with the right tool.

Endogenous model transformation. Endogenous model transformation describes transformations where source and target model conform to the same metamodel. In case that the source and target models are one and the same artifact, i.e., the source model is directly refined, this kind of transformation is also called *in-place transformation*. If a new target model based on the source model's properties is created, the transformation is called *out-place*, even if both models conform to the same metamodel [67].

Outstanding approaches realizing endogenous model transformations are among others the graph transformation tool AGG [29], the model transformation by-demonstration approach EMF Modeling Operations (EMO) [13], and the reflective API of Eclipse's Modeling Framework EMF allowing direct programmatic manipulation of models as, e.g., employed in graphical modeling editors [23]. This kind of model transformation frames the basis for model versioning, as it describes the evolution of a model. More precisely, the original version is considered as input model for an endogenous model transformation—either for a predefined transformation or for a manual transformation performed in a modeling editor—and the output model shapes the revised version.

Exogenous model transformation. An exogenous model transformation denotes a model transformation between models conforming to different metamodels, i.e., it takes one or more models of any modeling language as input and generates one or more new models of another modeling language from scratch. A special case of exogenous model transformation is *bidirectional model transformation*, which provides a synchronous framework, i.e., the transformation definition may be executed in both directions. Based on this definition, bidirectional model transformation further enables incremental change propagation from one model to

another model, triggering an in-place transformation to recover a consistent state. The unidirectional exogenous model transformation approach is implemented in, e.g., the Atlas Transformation Language (ATL) [43]. One representatives for the bidirectional model transformation approach is OMG's standard for model transformation Query/View/Transformation (QVT) [78].

Model-to-text transformation. Model-to-text transformations address the generation of code or any other text representation (e.g., configuration files and reports) from an input model. Such transformation languages employ usually a template-based approach, where the expected output text is parameterized with model elements conforming the input metamodel. Acceleo[5] implementing OMG's MOF Model-to-Text Transformation Language standard (Mof2Text) [82], JET[6], and Xpand[7] are well-known representatives for this category.

3.3 Model Differencing

One major task of model versioning is to obtain the operations that have been applied between two versions of a model (e.g., between V_o and V_{r1} in Fig. 1). As already discussed in Section 2, obtaining the applied operations can be accomplished using two alternatives: *operation recording* [41,48,60,101], which is often referred to as operation-based versioning, and *model differencing* [1,14,45,54,59], which is also referred to as state-based versioning. As operation recording is largely straightforward and depends on the interfaces of the modeling editor heavily, we will focus on the state of the art in model differencing in the following.

Model differencing is usually realized as follows. First, a *match* is computed, which describes the correspondences between two versions of a model. Second, the actual *differences* are obtained by a fine-grained comparison of all corresponding model elements based on the beforehand computed match. After the differences have been obtained, they have to be represented in some way for their further usage, such as conflict detection and merging. In the following, we will elaborate on the state of the art of these three tasks, matching, differencing, and representing differences in more detail.

Model Matching. The problem of matching two model elements is to find the *identity* of the model elements to be matched. Once the identity is explicit, model elements with equal identities are considered as a match. Thereby, the identity is computed by a match function. The characteristics of a model element that are incorporated to compute the identity of a model element within the match function, however, varies among approaches, scenarios, and objectives of performing model matching. The predecessors of model matching approaches stem from schema matching in the data base research domain [95] and from ontology matching in the knowledge representation research domain [115]. Therefore, we first highlight remarkable categorizations of matching techniques from these two

[5] `http://www.eclipse.org/acceleo`

[6] `http://www.eclipse.org/modeling/m2t/?project=jet#jet`

[7] `http://wiki.eclipse.org/Xpand`

research domains and, subsequently, proceed with surveying recent approaches to model matching.

Schema matching and ontology matching. The problem of matching database schema gained much attention among researchers for addressing various research topics, such as schema integration, data extraction for data warehouses and e-commerce, as well as semantic query processing. To reconcile the structure and terminology used in the emerged approaches from these research topics, Rahm and Bernstein [95] proposed a remarkable classification of existing approaches. On the most upper layer, Rahm and Bernstein distinguish between *individual matcher approaches* and *combining matchers*. Individual matchers are further classified according to the following largely orthogonal criteria. First of all, they consider whether matching approaches also incorporate instance data (i.e., data contents) or only the schema for deriving correspondences among schema elements. Further, they distinguish between approaches that perform the match only on single schema elements (i.e., they operate on *element level*) or on combinations of multiple elements to also regard complex schema structures (i.e., *structure level*). Another distinction is made upon approaches that uses either *linguistic-based matching* (e.g., based on names or descriptions) or *constraint-based matching* (e.g., unique key properties or data types). Matching approaches may also be characterized according to the match cardinality; that is, whether they return one-to-one correspondences or also one-to-n or even n-to-m correspondences. Finally, there are approaches that not only take a schema as input, but also exploit auxiliary information (e.g., dictionaries, global schemata, previous matching decisions, or user input). On the other side, among combining matchers, Rahm and Bernstein identified *hybrid matchers* that directly combine several matching approaches to determine match candidates based on multiple criteria or information sources. They also identified *composite matchers*, which combine the results of several *independently executed* matchers. The composition of matchers is either done automatically or manually by the user.

With the rise of the semantic web [6], the problem of integrating, aligning, and synchronizing different *ontologies* into one reconciled knowledge representation induced an active research area. Therefore, several ontology matching approaches have been proposed (cf. [115] for a survey). As argued by Shvaiko and Euzenat [105], schema matching and ontology matching are largely the same problem because schemata and ontologies both provide a vocabulary of terms that describes a domain of interest and both constrain the meaning of terms used in the vocabulary [105].

Model matching. The aforementioned categorizations and terminologies also can be used for characterizing model matching approaches. However, the distinction between schema-only and instance-based approaches only applies to approaches specifically tailored to match *metamodels*, because models on the $M1$ level in the metamodeling stack (cf. Section 3.1) have no instances to be used for matching. Furthermore, in the context of model matching, the only constraint-based similarity measure that can be used across all meta levels is the type information

(i.e., the respective metaclass) of a model element. Besides applying the categorization coming from schema and ontology matching, Kolovos et al. [52] further proposed a categorization specifically dedicated to model matching approaches. In particular, they distinguish between static identity-based matching, signature-based matching, similarity-based matching, and custom language-specific matching. Static identity-based matching relies on immutable UUIDs attached to each model element, whereas signature-based matching compares model elements based on a computed combination of feature values (i.e., its signature) of the respective model elements. Which features should be incorporated for computing this signature strongly depends on the modeling language. Whereas approaches of these two categories, identity- and signature-based matching, treat the problem of model matching as a true/false identity (i.e., two model elements are either a match or not), similarity-based matching computes an aggregated similarity measure between two model elements based on their feature values. As not all feature values of a model element are always significant for matching, they often can be configured in terms of weights attached to the respective features. Finally, custom language-specific matching enables its users to specify dedicated match rules in order to also respect the underlying semantics of the respective modeling language for matching.

In the following, we discuss existing approaches in the domain of model matching. Many existing approaches in this domain are integrated in *model versioning* tools. In the following, however, we focus on their model matching capabilities only, and discuss the respective approaches concerning their model versioning support again in Section 4.

One of the first model matching approaches has been proposed alongside their model comparison algorithm by Alanen and Porres [93]. Although their approach only supports UML models and, thereby, they easily could have incorporated language-specific match rules, the proposed match function relies on static identifiers only. Also, specifically tailored for a specific modeling language is UMLDiff [119], which is, however, not based on static identifiers. Instead, UMLDiff computes similarity metrics based on a model element's name and structure. In terms of the aforementioned categorizations, UMLDiff applies string-based matching at the element level as well as graph-based matching at the structure level and internally combines the obtained similarity measures; thus, UMLDiff is a hybrid matching approach. The same is true for the approach by Nejati et al. [73], which is specifically tailored for matching UML state machines. Their matching approach uses static similarity measures, such as typographic, linguistic, and depth properties of model elements, but also behavioural similarity measures. Also specifically tailored to UML models is ADAMS [20], which uses a hybrid matcher that first applies a static identity-based matcher and matches all remaining (not matched) model elements using a simple static signature-based approach based on model element names. In contrast to language-specific matching approaches, also several generic approaches have been proposed such as DSMDiff [59] and EMF Compare [14]. DSMDiff first compares elements based on a computed signature (incorporating the element name and type) and,

subsequently, considers the relationship among model elements previously matched by signature. Largely similar to DSMDiff, EMF Compare computes four different metrics and combines them to obtain a final similarity measure. In particular, EMF Compare regards the name of an element, its content, its type and the relations to other elements. EMF Compare also offers a static identity-based comparison mode, which works similarly to the approach by Alanen and Porres [93]. However, EMF Compare only allows for either similarity-based or static-identity based matching; both strategies cannot be combined. The similarity-based matching approach applied in EMF Compare heavily exploits the tree-based containment structure when comparing models. Rivera and Vallecillo [97] argue that this leads to issues concerning the detection of, for instance, elements that have been moved to new container elements. Therefore, Rivera and Vallecillo [97] propose to compare model elements independently of their depth in the containment tree. Besides this difference, the exploited information on model elements for matching is largely similar to DSMDiff and EMF Compare. DSMDiff and EMF Compare aim at obtaining an optimal result, whereas no language-specific information or configuration is necessary; in contrast, the goal of SiDiff [99] is to provide an adaptable model comparison framework, which may be fine-tuned for specific modeling languages by configuring the actual characteristics of model elements to be considered in the comparison process and attaching weights to these characteristics. DSMDiff, EMF Compare, and SiDiff are hybrid matching approaches. On the contrary, Barret et al. recently presented Mirador [5], which is a composite matching approach. That is, several matching strategies are independently applied and presented in a consolidated view of all match results. Using this view, users may interactively refine the computed match by attaching weights and manually discarding or adding matches. Thereby, the goal is to offer a wide assortment of model comparison algorithms and matching strategies under control of the user. Yet another approach is taken by Kolovos with the Epsilon Comparison Language (ECL) [50]. Instead of providing a set of predefined and configurable matching strategies, ECL is a hybrid rule-based language, which enables users to implement comparison algorithms at a high level of abstraction and execute them for identifying matches. Although it indeed requires some dedicated knowledge to create language-specific match rules with ECL, it facilitates highly specialized matching algorithms, which may also incorporate external knowledge, such as lexicons and thesauri.

In summary, during the last years several notable yet diverse approaches for model matching have been proposed. The set of available matchers ranges from generic to language-specific and from hybrid to composite approaches, whereas some are adaptable and some are not. Nearly all operate on the structure level regarding the importance of a model element's context. In contrast to ontology matching approaches, the approaches for model matching are mainly syntactic and do not incorporate external knowledge. Only ECL explicitly enables matchers that take advantage of external knowledge or even formal semantics.

Computing and Representing Differences. The differences among models may be computed based on three orthogonal levels: the *abstract syntax*, the *concrete syntax*, and the *semantics* of the models. The abstract syntax describes a model in terms of a tree (or more generally, a graph), whereas nodes represent model elements and edges represent references among them. Each node may further be described by a set of features values (i.e., attribute values). Thus, abstract syntax differencing approaches are only capable of detecting differences in the *syntactic data* that is carried in the compared models. Differencing approaches that also take the concrete syntax of a model into account are further capable of detecting changes of the *diagramming layout* visualizing of a model (e.g., [20,65,87,88]). More recently, Maoz et al. [62] introduce *semantic model differencing*, which aims at comparing the meaning [39] of models rather than their syntactic representation. For instance, Maoz et al. propose an algorithm to compute the differences between two UML activity diagrams regarding their possible execution traces, as well as the differences concerning the instantiability of UML class diagrams [63,64]. In the context of model versioning, the comparison of models is largely based on the abstract syntax currently, which is why we focus on such differencing approaches in the remainder of this section.

Model differencing based on the abstract syntax. Existing work in the area of differencing based on the abstract syntax mainly differ regarding the used approach for matching model elements across two versions of a model, which has been discussed above already, and they vary concerning the detectable types of differences. Most of the existing model differencing approaches are only capable of detecting the applied *atomic operations* (i.e., add, delete, move, and update). The computation of such applied operations works largely similar in existing approaches. That is, the differencing algorithms perform a fine-grained comparison of two model elements that correspond to each other (as indicated by the applied match function). If two corresponding model elements differ in some way (i.e., an update has been applied), a description of the update is created and saved to the list of differences. If a model element has no corresponding model element on the opposite side, an element insertion or deletion is noted.

Besides such atomic operations, developers may also apply *composite operations*. A composite operation is a set of cohesive atomic operations that are applied within one transaction to achieve ultimately one common goal. The most prominent class of such composite operations are *refactorings* as introduced by Opdyke [90] and further elaborated by Fowler et al. [33]. The composite operations that have been applied between two versions of a model represents a valuable source of information for several model management tasks [68]. Furthermore, this information helps other developers significantly to better understand the evolution of a software artifact [49]. Three approaches have been proposed for detecting applied composite operations from two states of a model. First, Xing and Stroulia [120] presented an extension of UMLDiff for detecting refactorings. In their approach, refactorings are expressed in terms of change pattern queries that are used to query a set of atomic differences obtained from the UMLDiff model differencing algorithm. If a query returns a match, an application of a refactoring is reported.

A very similar approach has been proposed by Vermolen et al. [113] to allow for a higher automation in model migration. Both approaches are restricted to a specific modeling language and use hard-coded refactoring detection rules. In contrast to these approaches, Kehrer et al. [44] propose to derive the detection rules from graph transformation units realizing the composite operations. The derived detection rules may then be matched with generic difference models containing the atomic operations that have been applied between two versions of a model.

Representation of differences. For assessing different approaches for representing differences between two versions of a model, Cicchetti et al. [16] identified a number of properties a representation of operations should fulfill. Most importantly, they mention the properties indicating whether a representation is *model-based* (i.e., conforming to a dedicated difference metamodel), *transformative* (i.e., applicable to the compared models), and *metamodel independent* (i.e., agnostic of the metamodel the compared models conform to). Besides these properties, it is also important how explicit the detected operations are represented, or whether important information (such as the index at which a value has been added to an ordered feature) is hidden in the context of a detected operation's representation.

In several research papers addressing the topic of model differences, such as [5,20,65], it is not explicitly mentioned how the detected differences are represented. Many others at least define the types of differences they aim to detect. For instance, DSMDiff [59] marks model elements to be *added*, *deleted*, or *changed*. Alanen & Porres [93] explicitly represent, besides added and deleted model elements, *updates* of single-valued features, *insertions* and *deletions* of values in multi-valued features as well as *ordered* features. SiDiff [99] distinguishes among *structural differences*, *attribute differences*, *reference differences*, and *move differences*. Several language-specific approaches, in particular, Gerth et al. [35], UMLDiff [119], and Ohst et al. [88], introduce operations that are tailored to the specific modeling language they support; thus, they use a metamodel dependent representation of applied operations. For instance, Gerth et al. defines the operations, such as *move activity* and *delete fragment*, for state machines and UMLDiff presents a fine-grained definition of UML class diagram operations, such as *new inheritance relationship* (for UML classes).

All of the approaches mentioned above do not represent the detected differences in terms of a model that conforms to a dedicated difference metamodel; at least, it is not explicitly mentioned in their research papers. Nevertheless, the difference representations by Alanen & Porres and Gerth et al. are transformative; that is, detected differences can be applied to the compared models in order to create a merged version. To the best of our knowledge, the only approaches that use a model-based representation of differences are EMF Compare [14], Herrmannsdoerfer & Koegel [41], and Cicchetti et al. [16]. All of these approaches are designed to be independent from the metamodel. Whereas EMF Compare and Herrmannsdoerfer & Koegel use a generic metamodel, Cicchetti et al. *generate* a dedicated difference metamodel for specific modeling languages. Thereby, in the approach by Cicchetti et al., a dedicated metaclass for indicating insertions, deletions, and changes for every metaclass in the respective modeling language's metamodel is

generated. For instance, for UML class diagrams, difference metaclasses, such as AddedClass and ChangedAttribute are generated, whereas Class and Attribute are metaclasses in the modeling language's metamodel. In contrast, EMF Compare and Herrmannsdoerfer & Koegel make use of the reflective power of EMF and refer to the modeling language's metaclasses to indicate, for instance, a modification of a specific feature of a model element. EMF Compare refers to the affected model element by a generic reference to EObject, which is the abstract type of all objects within EMF. In contrast, Herrmannsdoerfer & Koegel foresee a more flexible model referencing technique to also enable, for instance, persistent ID-based model element references.

3.4 Conflicts in Versioning

Whenever an artifact is modified concurrently by two or more developers, conflicts may occur. In the following, we survey existing definitions of the term *conflict* and discuss proposed conflict categorizations.

The term conflict has been used in the area of versioning to refer to interfering operations in the parallel evolution of software artifacts. However, the term conflict is heavily overloaded and differently co-notated. Besides using the term conflict, also the terms *interference* and *inconsistency* have been applied synonymously in the literature as, for instance, in [32,112] and [66], respectively. The term conflict usually refers to directly contradicting operations; that is, two operations, which do not commute [60]. Nevertheless, there is a multitude of further problems that might occur, especially when taking syntax and semantics of the versioned artifact's language into account. Therefore, in order to better understand the notion of conflict, different categories have been created to group specific merge issues as surveyed in the following.

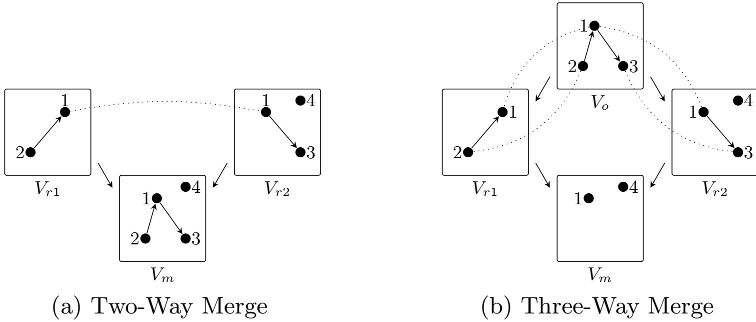
In the field of software merging, Mens [66] introduces *textual*, *syntactic*, *semantic*, and *structural* conflicts. Whereas *textual* conflicts concern contradicting operations applied to text lines as detected by a line-based comparison of a program's source code, *syntactic* conflicts denote issues concerning the contradicting modification of the parse tree or the abstract syntax graph; thus, syntactic merging takes the programming language's syntax into account and may also report operations that cause parse errors when merged (cf. line-based versus graph-based versioning in Section 2). *Semantic* merging goes one step further and also considers the semantic annotation of the parse tree, as done in the semantic analysis phase of a compiler. In this context, static semantic conflicts denote issues in the merged artifact such as undeclared variables or incompatible types. Besides static semantic conflicts, Mens also introduced the notion of *behavioral* conflicts, which denote unexpected behavior in the merged result. Such conflicts can only be detected by applying even more sophisticated semantic merge techniques that rely on the runtime semantics. Finally, Mens introduces the notion of *structural* conflicts, which arise when one of the applied operations to be merged is a "restructuring" (i.e., a refactoring) and the merge algorithm cannot uniquely decide in which way the merged result should be restructured.

Also the notion of conflict in the domain of graph transformation theory serves as a valuable source of knowledge in this matter. As defined by Heckel et al. [40], two direct graph transformations are in conflict, if they are *not parallel independent*. Two direct graph transformations are parallel independent, if they preserve all elements that are in the match of the other transformation; otherwise we encounter a *delete-use* conflict. Another manifestation of such a case is a *delete-delete* conflict. Although both transformations delete the same element anyway, this is still considered a conflict because one transformation deletes an element that is indeed in the match of the other transformation. If the graph transformations comprise negative application conditions, they also must not create elements that are prohibited by negative application conditions of the other transformation; otherwise an *add-forbid* conflict occurs. To summarize, two direct graph transformations are in conflict, if one of both disables the other. Furthermore, as shown in [26], based on the local Church-Rosser theorem [15], we may further conclude that two parallel independent direct transformations can be executed in any order with the same final result.

In the domain of model versioning, no widely accepted categorization of different types of merge conflicts has been established yet. Nevertheless, two detailed categorizations have been proposed by Westfechtel [117] and Taentzer et al. [110,111]; these categorizations concern *generic* conflicts between *atomic* operations only. In the following, we summarize these definitions briefly.

The conflict categorization by Westfechtel [117] is defined using set-theoretical rules and distinguishes between *context-free conflicts* and *context-sensitive conflicts*. Context-free conflicts denote contradicting modifications of the same feature value at the same model element; thus, such conflicts are independent of the context of the model element. Context-sensitive conflicts take also the context of a concurrently modified model element into account. With the term "context", Westfechtel refers to the neighbor elements of a model elements, such as its container or referenced model elements. Context-sensitive conflicts are again classified into *(i) containment conflicts*, which occur, for instance, if both developers move the same model element to different containers so that no unique container can be chosen automatically, *(ii) delete conflicts*, which denote deletions of elements that have been updated or moved concurrently, and *(iii) reference conflicts*, which concern contradicting changes to bi-directional references.

Taentzer et al. [110,111] present a fundamental categorization of conflicts based on graph theory. Thus, models are represented in terms of graphs, and changes applied to the models are formalized using graph modifications. On the most general level of this categorization, Taentzer et al. distinguish between *operation-based conflicts* and *state-based conflicts*. Operation-based conflicts are caused by two directly interfering graph modifications. More precisely, two graph modifications are conflicting, if either the source or the target node of an edge that has been inserted in one graph modification has been deleted concurrently in the other graph modification. State-based conflicts denote inconsistencies concerning the consistency rules of the respective modeling language in the merged graph; that is, the final *state* of the graph after applying the concurrent graph



(a) Two-Way Merge (b) Three-Way Merge

Fig. 6. Two-Way and Three-Way Merge Strategies

modifications to the common ancestor graph violates graph constraints stemming from the definition of the modeling language. In [111], Taentzer et al. refine this fundamental categorization for conflicts in EMF-based models. Thus, also the additional modeling features of EMF-based models are taken into account, such as containment references, feature multiplicities, and ordered features. Therefore, conflict patterns are introduced, which indicate combinations of concurrently applied operations leading to a conflict when applied to common model elements. In particular, they define *delete-use*, *delete-move*, *delete-update*, *update-update*, *move-move*, and *insert-insert* conflicts.

3.5 Merging

When only non-conflicting changes are detected between two versions of an artifact, merging is a straightforward task. In case of two-way merging, only the revised versions V_{r1} and V_{r2} are compared [18]. As deletions cannot be determined (cf. Section 2.1), the merged version V_m is constructed as joint union of both input artifacts, as depicted in Fig. 6a. In case of three-way merging, the merge is more reliable, as it takes the common ancestor V_o of both artifacts into account and is thus able to consider deleted elements [18]. The merge is performed by applying the union of all changes detected between the common ancestor and both revised versions to the common ancestor version. Consider the example in Fig. 6b. Element 3 is deleted in V_{r1}, while in V_{r2} element 2 is deleted and element 4 is added. Consequently, the merged version V_m is constructed by deleting elements 2 and 3 from the ancestor version V_o and adding the new element 4 resulting in a version consisting of elements 1 and 4. As the more powerful three-way merging approach is the preferred strategy in almost all versioning systems [66], we neglect two-way merging in the following. However, whenever conflicting changes are detected between the two revised versions V_{r1} and V_{r2}, the merged version cannot be uniquely determined, regardless whether two-way or three-way merging is employed.

Manual conflict resolution. The most pragmatic solution is to shift the responsibility of merging to the user, whenever it comes to conflicting changes. Versioning systems for code like Subversion typically employ a manual merge strategy. Then, the two parallel evolved versions are shown to the user side by side. Conflicting and non-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. This approach works satisfactory well for line-oriented artifacts like source code and is thus employed in most model versioning systems too. However, the graph based structure of models impedes manual merging, as dependent changes are not located close together in a sequence, but may be scattered across the model. Considering visual models with their dual representation manifested in the abstract syntax and the graphical concrete syntax, renders manual merging even harder. If these representations are merged separately, the user's perception is completely destroyed. Even if graphical comparison of both versions side by side mitigates the representational gap, the manual effort for identifying corresponding elements on the two-dimensional canvas increases with the model's size. Recently, Gerth et al. [36] propose to support manual merging of process models by guiding the modeler through conflict resolution. They apply non-conflicting changes automatically and suggest one of three strategies depending on the conflict type at hand.

Automatic conflict resolution. As manual conflict resolution is error-prone and cumbersome, it seems naturally, that avoiding conflicts by automatic merge strategies is a preferable goal. Munson and Dewan present a flexible framework for merging arbitrary objects, which may be configured in terms of merge policies [69]. 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. Edwards [25] proposes further strategies for conflict management in collaborative applications and allows to distinguish manual and automatic resolution in his merge policies. Automatic conflict resolution is achieved by calculating all possible combinations of parallel performed operations leading to a valid version. Alanen & Porres [1] use a fixed policy in their merge algorithm for MOF based models to interleave the differences performed in V_{r2} with the differences of V_{r1}. Cicchetti et al. [17] allow to adapt conflict detection and merging by defining conflict patterns describing specific difference patterns, which are not allowed to occur in V_{r1} and V_{r2} together. These conflict patterns are supplemented with a reconciliation strategy, stating which side should be preferred in the merge process. While policy based approaches require user intervention in certain conflict cases where no policy is at hand, Ehrig et al. [28] present a formal merge approach based on graph transformation theory, yielding a merged model by construction.

Conflict tolerance. In contrast to automatic merging, 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 [74]. 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. Schwanke and Kaiser [102] propose an adapted programming environment for identifying, tracking, tolerating, and periodically resolving inconsistencies. Similarly, Balzer [4] allows to tolerate inconsistencies in programming environments and database systems by relaxing consistency constraints and annotating inconsistent parts with so called *pollution markers*.

To summarize, existing merge strategies are manifold. Manual merge approaches provide on the one hand most user control, but require on the other hand high effort and bear the risk of loosing changes. Automatic merge approaches in contrast, accelerate merging and reduce manual intervention. However, this benefit comes at the cost of loss of control. Conflict tolerance reveals a completely different strategy and allows to temporarily tolerate an inconsistent state of the merged artifact instead of immediately rolling back conflicting changes. The drawback of conflict tolerance is the need for dedicated editors and that the attached pollution markers may violate the grammar of highly structured artifacts like models. However, the variety of existing merge approaches reflects the pivotal role of the merge process.

4 State-of-the-art Model Versioning Systems

In the previous section, we discussed underlying concepts and existing fundamental techniques acting as a basis for building a model versioning system. In this section, we present the current state-of-the-art in model versioning and evaluate the features of existing solutions stemming from industry and academia.

4.1 Features of Model Versioning Approaches

Before we survey the state-of-the-art model versioning systems, we first discuss the particular features of model versioning systems that we have investigated in this survey.

Operation recording versus model differencing. As already introduced in Section 2, we may distinguish between approaches that obtain operations performed between two versions of a model by applying *operation recording* or by *model differencing*. If an approach applies model differencing, which is, in general, more flexible concerning the adopted modeling editors, it is substantial to consider the techniques conducted in the match function for identifying corresponding

model elements because the quality of the match is crucial for an accurate subsequent operation detection. We may distinguish between match functions that rely on universally unique IDs (*UUIDs*), and those applying heuristics based on the model element's *content* (i.e., feature values and contained child elements).

Composite operation detection. The knowledge on applied composite operations is the prerequisite for considering them in the merge process. Therefore, it is a distinguished feature whether an operation detection component is also capable of detecting applications of composite operations besides only identifying atomic operations. It is worth noting that, in case of model differencing, the state-based a posteriori detection of composite operation applications is highly challenging as stated in Section 6 of [21].

Adaptability of the operation detection. Obviously, generic operation detection approaches are, in general, more flexible than language-specific approaches because it is very likely that several modeling languages are concurrently applied even within one project and, therefore, should be supported by one model versioning system. However, neglecting language-specific aspects in the operation detection phase might lead to a lower quality of the detected set of applied operations. Therefore, we investigate whether generic operation detection approaches are adaptable to language-specific aspects. In particular, we consider the adaptability concerning language-specific match rules, as well as the capability to extend the detectable set of language-specific composite operations.

Detection of conflicts between atomic operations. One key feature of model versioning systems is, of course, their ability to detect conflicts arising from contradictory operations applied by two developers in parallel. Consequently, we first investigate whether the approaches under consideration are capable of detecting conflicts between contradictory atomic operations. In this survey, we do not precisely examine *which* types of conflicts are supported. We rather investigate whether conflicts among atomic operations are considered at all.

Detection of conflicts caused by composite operations. Besides conflicts caused by contradicting atomic operations, conflicts might also occur if a composite operation applied by one developer is not applicable anymore, after the concurrent operations of another developer have been performed. Such a conflict occurs, if a concurrent operation causes the preconditions of an applied composite operation to fail. Therefore, we investigate whether the model versioning approaches adequately consider composite operations in their conflict detection phase.

Detection of inconsistencies. Besides conflicts caused by operations (atomic operations and composite operations), a conflict might also occur if the merged model contains errors in terms of the modeling language's well-formedness and validation rules. Consequently, we examine model versioning approaches under consideration whether they perform a validation of the resulting merged model.

Adaptability of the conflict detection. With this feature, we review the adaptability to language-specific aspects of the conflict detection approach. This involves techniques to configure language-specific conflict types that cannot be detected by analyzing of the applied operations only generically.

Graphical visualization of conflicts. Developers largely create and modify models using a graphical diagramming editor. Thus, also the occurred conflicts should be visualized and resolved graphically on top of the model's concrete syntax. Hence, we investigate whether developers have to cope with switching the visualization for understanding and resolving conflicts, or whether they are allowed to stick with their familiar way of working with their models.

Adaptable resolution strategies. If a model versioning system offers techniques for resolving certain types of conflicts automatically, the correct resolution strategy is key. However, the correct resolution strategy may depend strongly on the modeling language, the aim of the models, the project culture, etc. Thus, it is important to allow users to adapt the offered resolution strategies to their needs.

Flexibility concerning the modeling language. This feature indicates whether model versioning systems are tailored to a specific modeling language and, therefore, are only usable for one modeling language, or whether they are generic and, therefore, support all modeling languages defined by a common meta-metamodel.

Flexibility concerning the modeling editor. Model versioning systems may be designed to work only in combination with a specific editor or modeling environment. This usually applies to approaches using operation recording. In contrast, model versioning systems may avoid such a dependency and refrain from relying on specific modeling environments by only operating on the evolved models put under version control.

4.2 Evaluation Results

In this section, we introduce current state-of-the-art model versioning systems and evaluate them on the basis of the features discussed in the previous section. The considered systems and the findings of this survey are summarized in Table 1 and discussed in the following. Please note that the order in which we introduce the considered systems is alphabetically and has no further meaning.

ADAMS. The "Advanced Artifact Management System" (ADAMS) offers process management functionality, supports cooperation among multiple developers, and provides artifact versioning [19]. ADAMS can be integrated via specific plug-ins into modeling environments to realize versioning support for models. In [20], De Lucia et al. present an ADAMS plug-in for ArgoEclipse[8] to enable version support for ArgoUML models. Because artifacts are stored in a proprietary ADAMS-specific format to be handled by the central repository, models have to be converted into that format before they are sent to the server and translated back to the original format, whenever the model is checked out again. ADAMS applies state-based model differencing based on UUIDs. Added model elements, which, as a consequence, have no comparable UUIDs, are matched using simple heuristics based on the element names to find corresponding elements concurrently added by another developer. The differences are computed at the client and sent to the ADAMS server, which finally performs the merge.

[8] `http://argoeclipse.tigris.org`

Table 1. Evaluation of State-of-the-art Model Versioning Systems

	Operation Detection						Conflict Detection				Resolution		Flexibility	
	Operation Recording	Model Differencing		Composite Operations	Adaptability		Operation-based Conflicts		Inconsistencies	Adaptability	Graphical Visualization	Adaptable Resolution Strategies	Modeling Language	Modeling Editor
		UUID	Content		Match	Operations	Atomic	Composite						
ADAMS	-	✓	✓	-	-	-	✓	-	-	~	-	-	-	✓
Alanen and Porres	-	✓	-	-	-	-	✓	-	✓	-	-	-	-	✓
Cicchetti et al.	n/a	n/a	n/a	n/a	n/a	n/a	✓	✓	-	✓	-	✓	✓	✓
CoObRA	✓	-	-	✓	-	-	✓	-	~	-	✓	-	-	-
EMF Compare	-	✓	✓	~	-	~	✓	-	-	~	-	-	✓	✓
EMFStore	✓	-	-	✓	-	~	✓	~	✓	-	-	-	✓	~
Gerth et al.	-	✓	-	✓	-	-	✓	✓	✓	-	-	-	-	✓
Mehra et al.	-	✓	-	-	-	-	✓	-	-	-	✓	-	✓	✓
Oda and Saeki	✓	-	-	-	-	-	~	-	✓	-	-	-	✓	-
Odyssey-VCS 2	-	✓	-	-	-	-	✓	-	-	~	-	-	✓	✓
Ohst, Welle, Kelter	-	✓	-	-	-	-	✓	-	-	-	✓	-	-	✓
RSA	-	✓	✓	-	-	-	✓	-	✓	-	✓	-	-	✓
SMOVER	-	✓	-	-	-	-	~	-	-	~	-	-	✓	✓
Westfechtel	-	n/a	n/a	-	-	-	✓	-	~	-	-	-	✓	✓

Legend	
✓	Feature applies.
~	Feature partially applies.
-	Feature does not apply.
n/a	Not applicable or unknown.

The ADAMS plug-in supports ArgoUML models only. Interestingly, ADAMS can be customized to a certain extent. For instance, it is possible to customize the unit of comparison; that is, the smallest unit, for which, if concurrently modified, a conflict is raised. In [20], it is also mention that the conflict detection algorithm may be customized for specific model types with user-defined *correlation rules*, which specify when two operations should be considered as conflicting. However, it remains unclear, how these rules are exactly specified and how these rules influence the conflict detection. The implementation promoted in this publication is not available to further review this interesting customization feature. Composite operations and state-based conflicts are not supported.

Approach by Alanen and Porres. One of the earliest works on versioning UML models was published by Alanen and Porres [1], who present metamodel-independent algorithms for difference calculation, model merging, as well as

conflict resolution. They identified seven elementary operation types a developer may perform to modify a model. For calculating the differences between the original version and the modified version, a match between model elements is computed based on UUIDs first. Subsequently, the created, deleted, and changed elements are identified based on the match computed before; composite operations are, however, not considered. Alanen and Porres provide an algorithm to compute a union of two sets of operations, whereas also an automatic merging for values of ordered features is presented. Furthermore, Alanen and Porres also popose to validate the merged result and envision a semi-automatic resolution process. Their work serves as a fundamental contribution to the model versioning domain and influenced many other researchers strongly.

Approach by Cicchetti, Di Ruscio, and Pierantonio. Cicchetti et al. [17] present an approach to specify and detect language-specific conflicts arising from parallel modifications. Their work does not address the issue of obtaining differences, but proposes a model-based way of representing them. Howsoever the differences are computed, they are represented by instantiating an automatically generated language-specific difference metamodel (cf. Section 3.3). Conflicts are specified by manually created conflict patterns. Language-specific conflict patterns are represented in terms of forbidden difference patterns. Thereby, the realization of a customizable conflict detection component is possible. Although the authors do not discuss how differences and applications of composite operations are obtained, their approach supports also conflicts caused by composite operations. The authors also allow to specify reconciliation strategies (i.e., automatic resolution strategies) to specific conflict patterns. It seems to be a great deal of work to establish a complete set of conflict patterns and resolution patterns for a specific language; nevertheless, in the end, a highly customized model versioning system can be achieved.

CoObRA. The Concurrent Object Replication framework CoObRA developed by Schneider et al. [101] realizes optimistic versioning for the UML case tool Fujaba[9]. CoObRA records the operations performed on the model elements and stores them in a central repository. Whenever other developers update their local models, these operations are fetched from this repository and replayed locally. To identify equal model elements, UUIDs are used. Conflicting operations are not applied (also the corresponding local change is undone) and finally presented to the user, who has to resolve these conflicts manually. In [100], the authors also shortly discuss state-based conflicts (i.e., inconsistencies). CoObRA is capable of detecting a small subset of such conflicts when the underlying modeling framework rejects the execution of a certain operation. For example, a class cannot be instantiated anymore if the respective class has been concurrently deleted. However, for instance, concurrent additions of an equally named class is not reported as conflict. The authors also shortly mention composite operations in terms of a set of atomic operations grouped into commands. The operation recording component seems to be capable of grouping atomic operations into commands to

[9] `http://www.fujaba.de`

allow for a more comprehensible undo mechanism. In particular, one command in the modeling editor might cause several atomic operations in the log; if the user aims to undo the last change, the complete command is undone and not only the latest atomic change. In their papers, however, no special treatment of these commands in the merge process is mentioned.

EMF Compare. The open-source model comparison framework EMF Compare [14], which is part of the Eclipse Modeling Framework Technology project, supports generic model comparison and model merging. EMF Compare provides two-way and three-way model comparison algorithms for EMF-based models. EMF Compare's model comparison algorithm consists of two phases, a matching phase and a differencing phase (cf. Section 3.3). EMF Compare provides a merge service, which is capable of applying difference elements in a difference model to allow for merging models. It also offers basic conflict detection capabilities and user interfaces for displaying match and difference models. All these features of EMF Compare are generic; consequently, they can be applied to any EMF-based model irrespectively of the modeling language these models conform to. However, EMF Compare can be extended programmatically for language-specific matching and differencing. Thus, it is not adaptable in the sense that it can be easily configured for a specific language, but it constitutes a programmatically extensible framework for all tasks related to model comparison.

EMFStore. The model repository EMFStore, presented by Koegel et al. [48], has been initially developed as part of the Unicase[10] project and provides a dedicated framework for model versioning of EMF models. After a copy of a model is checked out, all operations applied to this copy are tracked by the modeling environment. Once all modifications are done, the recorded operations are committed to a central repository. For recording the operations, a framework called *Operation Recorder* [41] is used, which allows to track any modifications performed in an EMF-based editor. Also transactions (i.e., a series of dependent operations) can be tracked and grouped accordingly. Having two lists of the recorded operations, in particular, the list of uncommitted local operations and the list of new operations on the server since the last update, the relationships the *requires relationship* and the *conflicts relationship* are established among the operations. The former relationship expresses dependencies between operations, the later indicates contradicting modifications. As the exact calculation of these relationships requires expensive computations, heuristics are applied to obtain an approximation for setting up those relationships. The conflict detection component classifies two operations as conflicting, if the same attribute or the same reference is modified. Furthermore, the authors introduce levels of severity to classify conflicts. They distinguish between hard conflicts and soft conflicts referring to the amount of user support necessary for their resolution. Whereas hard conflicts do not allow including both conflicting operations within the merged model, for soft conflicts this is possible (with the danger of obtaining

[10] http://www.unicase.org

an inconsistent model). Summarizing, EMFStore is completely operation-based; that is, the actual model states are never considered for detecting conflicts. This also entails that a removed and subsequently re-added model element is treated as a new model element so that all concurrent operations to the previously removed element are reported as conflict. Composite operations can be recorded and saved accordingly. In the conflict detection, however, composite operations are not specifically treated. If an atomic change within a composite operation conflicts with another change, the complete transaction is indeed marked as conflicting; the intentions behind composite operations, as well as potentially violated preconditions of composite operations are not specifically considered.

Approach by Gerth et al. Gerth et al. [35] propose a conflict detection approach specifically tailored to the business process modeling language (BPMN) [83]. To identify the differences between two process models (cf. [54]), in a first step, a mapping between corresponding elements across two versions of a process model is computed based on UUIDs which are attached to each element. In the next step, for each element that has no corresponding counterpart in the opposite version, a operation is created representing the addition or deletion. The resulting operations are specific to the type of the added or deleted element (e.g., InsertAction or DeleteFragment). Finally, this list of operations is hierarchically structured according to the fragment hierarchy of the process model in order to group those atomic operations into so-called compound operations. Consequently, these compound changes group several atomic operations into composite additions or deletions. Having identified all differences in terms of operations between two process models, syntactic, as well as semantic conflicts among those concurrent operations can be identified using a term formalization of process models. According to their definitions, a syntactic conflict occurs if an operation is not applicable after another operation has been performed. A semantic conflict is at hand whenever two operations modify the same elements so that the process models are not “trace equivalent”; that is, all possible traces of a process model are not exactly equal. Obviously, rich knowledge on the operational semantics of process models has to be encoded in the conflict detection to be able to reveal semantic conflicts. Although the authors presented an efficient way of detecting such conflicts, no possibility to adapt the operation detection and conflict detection mechanisms to other languages is foreseen.

Approach by Mehra, Grundy, and Hosking. The publication by Mehra et al. [65] mainly focuses on the graphical visualization of differences between versions of a diagram. Therefore, they provide a plug-in for the meta-CASE tool *Pounamu*, a tool for the specification and generation of multi-view design editors. The diagrams created with this tool are serialized in XMI and are converted into an object graph for comparison. In their proposed comparison algorithm, the differences are obtained by applying a state-based model differencing algorithm, which uses UUIDs to map corresponding model elements. The obtained differences are translated to Pounamu editing events, which are events corresponding to the actions performed by users within the modeling environment. Differences

cover not only modifications performed on the model, but also modifications performed on the graphical visualization. The differences between various versions are visualized in the concrete syntax so that developers may directly accept or reject modifications on top of the graphical representation developers are familiar with. In their works, also conflict detection facilities are shortly mentioned, however, not discussed in detail.

Approach by Oda and Saeki. Oda and Saeki [87] propose to also generate versioning features along with the modeling editor generated from a specified metamodel as known from metamodeling tools. The generated *versioning-aware* modeling editors are capable of recording all operations applied by the users. In particular, the generated tool records operations to the logical model (i.e., the abstract syntax tree of a model), as well as the diagram's layout information (i.e., the concrete syntax). Besides recording, the generated modeling tool includes check in, check out, and update operations to interact with a central model repository. It is worth noting that only the change sequences are sent to the repository and not the complete model state. In case a model has been concurrently modified and, therefore, needs to be merged, conflicts are identified by re-applying all recorded operations to the common ancestor version. Before each change is performed in the course of merging, its precondition is checked. In particular, the precondition of each change is that the modified model element must exist. Thereby, delete-update conflicts can be identified. Update-update conflicts, however, remain unrevealed and, consequently, the values in the resulting merged model might depend on the order in which the recorded updates are applied because one update might overwrite another previous update. Composite operations and their specific preconditions are not particularly regarded while merging. The approach also does not enable to specify additional language-specific conflicts. Although metamodel violations can, in general, be checked in their tool, they are not particularly considered in the merge process. As the versioning tool is generated from a specific metamodel, the generated tool is language dependent; the approach in general, however, is independent from the modeling language. However, the approach obviously forces users to use the generated modeling editor to be able to use their versioning system.

Odyssey-VCS 2. The version control system Odyssey-VCS by Oliveira et al. [89] is dedicated to versioning UML models. Operations between two versions of a model are identified by applying state-based model differencing relying on UUIDs for finding corresponding model elements. Language-specific heuristics for the match functions may not be used. Also language-specific composite operations are neglected. Interestingly, however, for each project, so-called behavior descriptors may be specified, which define how each model element should be treated during the versioning process. Consequently, the conflict detection component of Odyssey-VCS is adaptable, in particular, it may be specified which model elements should be considered to be atomic. If an atomic element is changed in two different ways at the same time, a conflict is raised. These behavior descriptors (i.e., adaptations) are expressed in XML configuration files. Thus, Odyssey-VCS is customizable for different projects

concerning the unit of comparison, as well as whether to apply pessimistic or optimistic versioning. Conflicts coming from language-specific operations, as well as additional language-specific conflicts, however, may not be configured. More recently, Odyssey-VCS 2 [70] has been published, which is capable of processing any EMF-based models and not only UML models. A validation of the resulting merged model is not considered.

Approach by Ohst, Welle, and Kelter. Within the proposed merge algorithm, also Ohst et al. [88] put special emphasis on the visualization of the differences. Therefore, differences between the model as well as the layout of the diagram are computed by applying a state-based model differencing algorithm relying on UUIDs. Conflict detection, however, is not discussed in detail; only update-update and delete-update conflicts are shortly considered. After obtaining the differences, a preview is provided to the user, which visualizes all modifications, even if they are conflicting. The preview diagram can also be modified and, therefore, allows users to resolve conflicts easily using the concrete syntax of a diagram. For indicating the modifications, the different model versions are shown in a unified document containing the common parts, the automatically merged parts, as well as the conflicts. For distinguishing the different model versions, coloring techniques are used. In the case of delete-update conflicts, the deleted model element is crossed out and decorated with a warning symbol to indicate the modification.

IBM Rational Software Architect (RSA). The Eclipse-based modeling environment RSA [11] is a UML modeling environment built upon the Eclipse Modeling Framework. Under the surface, it uses an adapted version of EMF Compare for UML models offering more sophisticated views on the match and difference models for merging. These views show the differences and conflicts in the graphical syntax of the models. The differencing and conflict detection capabilities are, however, equal to those of EMF Compare, besides that RSA additionally runs a model validation against the merged version and, in case an validation rule is violated, the invalid parts of the model are graphically indicated.

SMOVER. The semantically-enhanced model versioning system by Reiter et al. [96], called SMOVER, aims at reducing the number of falsely detected conflicts resulting from syntactic variations of semantically equal modeling concepts. Furthermore, additional conflicts shall be identified by incorporating knowledge on the modeling language's semantics. This knowledge is encoded by the means of model transformations, which rewrite a given model to so-called semantic views. These semantic views provide a canonical representations of the model, which makes certain aspects of the modeling language more explicit. Consequently, also potential semantic conflicts might be identified when the semantic view representations of two concurrently evolved versions are compared. It is worth noting that the system itself is independent from the modeling language and language-specific semantic views can be configured to adapt the system to

[11] `http://www.ibm.com/developerworks/rational/library/05/712_comp/index.html`

a specific modeling language. The differences are identified using a state-based model differencing algorithm based on UUIDs. Therefore, the system is independent of the used modeling editor. However, this differencing can not be adapted to specific modeling languages and only works in a generic manner. SMOVER also only addresses detecting conflicts regarding the semantics of a model and does not cover syntactic operation-based conflicts.

Approach by Westfechtel. Recently, Westfechtel [117] presented a formal approach for merging EMF models. Although no implementation of his work is available, it provides well-defined conflict rules based on set-theoretical conflict definitions. In [117], Westfechtel does not address the issue of identifying differences between model versions and rather focuses on conflict detection only and assumes the presence of change-based differences that can be obtained by, for instance, EMF Compare. Westfechtel's approach is directly tailored to EMF models and defines *context-free merge rules* and *context-sensitive merge rules.* Context-free merge rules determine "the set of objects that should be included into the merged versions and consider each feature of each object without taking the context [i.e., relationships to other objects] into account" [117]. The presented algorithm also supports merging of ordered features and specifies when to raise update-update conflicts. The conflict types defined by Westfechtel have been discussed in Section 3.4 already. Besides these operation-based conflicts, Westfechtel also addresses conflicts arising from the well-formedness rules of EMF. However, no techniques that enable further language-specific constraints are discussed. Moreover, he only addresses conflicts among atomic operations and is not adaptable to language-specific knowledge.

4.3 Summary

After surveying existing model versioning approaches, we may conclude that the predominant strategy is to apply state-based model differencing and generic model versioning. The majority of model differencing approaches rely on UUIDs for matching. However, only ADAMS combines UUIDs and (very simple) content-based heuristics. The detection of applications of composite operations is only supported by approaches applying operation recording. The only approach that is capable of detecting composite operations by using a state-based model comparison approach is Gerth et al.; however, their approach is specifically tailored to process models and the supported composite operations are limited to compound additions and deletions. Consequently, none of the surveyed generic approaches is capable of detecting applications of more complex composite operations having well-defined pre- and postconditions without directly recording their application in the editor. Furthermore, none of the approaches are adaptable in terms of additional match rules or composite operation specifications. EMF Compare and EMFStore foresee at least an interface to be implemented in order to extend the set of detectable applications of composite operations. In EMF Compare, however, the detection algorithm has to be provided by an own implementation.

In EMFStore, additional commands may be plugged into the modeling editor programmatically for enabling EMFStore to record them.

Obviously, all model versioning approaches provide detection capabilities for conflicts caused by two concurrent atomic operations. Unfortunately, most of them lack a detailed definition or at least a publicly available implementation. Therefore, we could not evaluate which types of conflicts can actually be detected by the respective approaches. In this regard, we may highlight Alanen and Porres, EMF Compare, EMFStore, Gerth et al., and Westfechtel. These either clearly specify their conflict detection rules in their publications or publish their detection capabilities in terms of a publicly available implementation.

Only Cicchetti et al. and Gerth et al. truly consider composite operations in their conflict detection components. However, in the case of Cicchetti et al., all potentially occurring conflict patterns in the context of composite operations have to specified manually. It is not possible to derive automatically the conflict detection capabilities regarding composite operations from the specifications of such operations. The approach by Gerth et al. is limited to specific modeling language and supports only rather simple composite operations. EMFStore partially respects composite operations: if a conflict between two atomic operations is revealed and one atomic operation is part of a composite operation, the complete composite operation is reverted. However, additional preconditions of composite operations are not considered. None of the surveyed approaches aims at respecting the original intention behind the composite operation; that is, incorporating concurrently changed or added elements in the re-application of the composite operation when creating the merged version.

Several of the surveyed approaches take inconsistent merge results into account. CoObRA is capable of detecting at least a subset of all potentially occurring violations of the modeling language's rules. Westfechtel only addresses the basic well-formedness rules coming from EMF, such as spanning containment tree. The approaches proposed by Alanen and Porres, EMFStore, Gerth et al., Oda and Saeki, and the RSA perform a full validation after merging.

Most of the proposed conflict detection approaches are not adaptable. ADAMS and Odyssey-VCS provide some basic configuration possibilities such as changing the unit of comparison. EMF Compare can be programmatically extended to attach additional conflict detection implementations. Only Cicchetti et al. and SMOVER allow to plug in language-specific artifacts to enable revealing additional conflicts. However, in the approach by Cicchetti et al., the conflict patterns have to be manually created in terms of object models, which seems to be a great deal of work requiring deep understanding of the underlying metamodel. Due to the lack of a public implementation, it is hard to evaluate the ease of use and the scalability of this approach. SMOVER allows to provide a mapping of a model to a semantic view in order to enable the detection of semantically equivalent or contradicting parts of a model. The comparison and conflict detection algorithm that is applied to the semantic views, however, is not adaptable. Consequently, SMOVER only aims to detect a very specific subset of conflicts only.

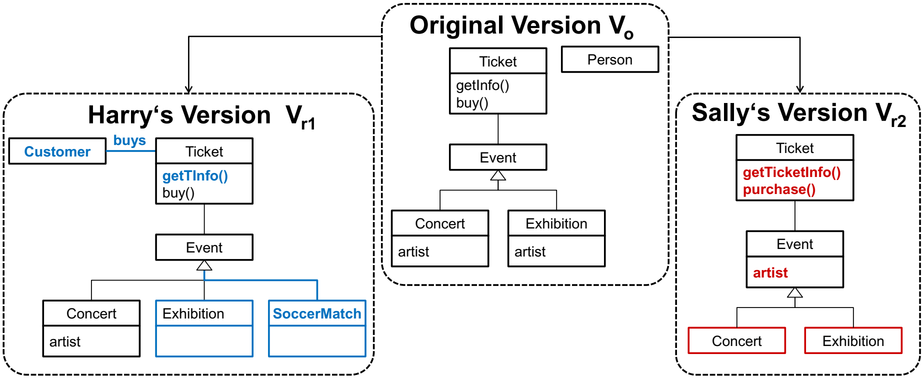


Fig. 7. Model Versioning Example

Conflict resolution has not gained much attention among existing approaches yet. Only four of the 15 surveyed approaches, namely CoObRA, Mehra et al., Ohst et al., and the RSA provide dedicated views for visualizing conflicts adequately to help developers to understand and resolve conflicts. Most notably concerning conflict resolution is the approach by Cicchetti et al., which allows to specify dedicated conflict resolution strategies for certain conflict patterns.

5 An Introduction to AMOR

In this section, we introduce the *adaptable model versioning system* AMOR[12], which has been jointly developed at the Vienna University of Technology[13], the Johannes Kepler University Linz[14], and SparxSystems[15]. Therefore, we first present a model versioning scenario in Section 5.1 serving as running example throughout the remainder of this section. Next, we discuss the goals of AMOR and give an overview of the AMOR merge process in Section 5.2. Subsequently, we describe each step in the merge process in more detail in the sections 5.3 to 5.6.

5.1 Running Example

Consider the following example. The modelers Harry and Sally work together on a project, where an event managing system has to be developed. Both modelers check out the latest version of the common repository (cf. Original Version V_o in Fig. 7) and start with their changes. Harry renames the class Person to Customer and adds an association buys from Customer to Ticket. He further renames the operation getInfo() in class Ticket to getTInfo(). In Harry's opinion exhibitions do

[12] http://www.modelversioning.org
[13] http://www.tuwien.ac.at
[14] http://www.jku.at
[15] http://www.sparxsystems.eu

not have an artist. Thus he deletes the property artist from the class Exhibition. Afterwards, he checks in his revised version resulting in Harry's Version in Fig. 7. In the meanwhile, unaware of Harry's changes, Sally performs the following changes. She renames both operations in the class Ticket. The operation getInfo() is renamed to getTicketInfo() and the operation buy() is renamed to purchase(). She identifies the property artist, which is common to all subclasses of the class Event, as undesirable redundancy, and performs the refactoring Pull Up Field to shift the property to the superclass. Finally, she deletes the isolated class Person and commits her revised version to the common repository. However, the commit fails, as her changes partly contradict Harry's changes.

In the following, we discuss the technical details how the model versioning system AMOR detects and reports the occurred conflicts and accompany Sally while she is merging her changes with Harry's changes.

5.2 AMOR at a Glance

The main goal of AMOR is to combine the advantages of both generic and language-specific model versioning by providing a generic, yet adaptable model versioning framework. The generic framework offers versioning support for all modeling languages conforming to a common meta-metamodel out of the box and enables users to enhance the quality of the versioning capabilities by adapting the framework to specific modeling languages using well-defined adaptation points. Thereby, developers are empowered to balance flexibly between reasonable adaptation efforts and the required level for versioning support. For realizing this goal, we aligned the development of each component according to the following design principles.

Flexibility concerning modeling language and editor. In traditional, code-centric versioning, mainly language-independent systems that do not pose any restrictions concerning the used editor gained significant adoption in practice. Thus, we may draw the conclusion that a versioning system that only supports a restricted set of languages and that has an inherent dependency on the used editor might not find broad adoption in practice. Also, when taking into consideration that domain-specific modeling languages are becoming more and more popular, language-specific systems seem to be an unfavorable choice.

Therefore, AMOR is designed to provide generic versioning support irrespective of the used modeling languages and modeling editors. Generic versioning is accomplished by using the reflective interfaces of the Eclipse Modeling Framework [107] (EMF) serving as reference implementation of OMG's MOF standard [77] (cf. Section 3.1). Thereby, all modeling languages can be handled immediately for which an EMF-based metamodel is available.

AMOR is also independent of the used modeling editor and does not rely on specific features on the editor side. Therefore, we may not apply editor-specific operation recording to obtain the applied operations. Instead, AMOR works only with the states of a model before and after it has been changed and derives the applied operations using state-based model differencing.

Easy adaptation by users. Generic versioning systems are very flexible, but they lack in precision in comparison to language-specific versioning systems because no language-specific knowledge is considered. Therefore, AMOR is adaptable with language-specific knowledge whenever this is needed. Some existing model versioning approaches are adaptable in terms of programming interfaces. Hence, it is possible to implement specific behavior to adapt the system according to their needs. Especially with domain-specific modeling languages, a plethora of different modeling languages exists, which often are not even publicly available. Bearing that in mind, it is hardly possible for versioning system vendors to pre-specify the required adaptations to incorporate language-specific knowledge for all existing modeling languages. Thus, users of the versioning system should be enabled to create and maintain those adaptation artifacts by themselves. This, however, entails that these adaptation artifacts do not require deep knowledge on the implementation of the versioning system and programming skills. Therefore, AMOR is designed to be adapted by providing descriptive adaptation artifacts and uses, as far as possible, well-known languages to specify the required language-specific knowledge. No programming effort is necessary to enhance AMOR's versioning capabilities with respect to language-specific aspects. Besides aiming at the highest possible adaptability, the *ease of adaptation* is one major goal of AMOR.

AMOR Merge Process. The merge process of AMOR is depicted in Fig. 8. This figure presents a more fine-grained view on the same merge process that is depicted in Fig. 1. Furthermore, we now illustrate explicitly the artifacts that are exchanged between the steps of this process. The input of this merge process are three models: the common original model V_o and two concurrently changed models, V_{r1} and V_{r2}.

The first phase of the merge process concerns the operation detection. The goal of this phase of the process is to detect precisely which operations have been applied in between V_o and V_{r1}, as well as between V_o and V_{r2}. As argued above, AMOR aims to be independent from the modeling editor. Hence, a state-based model comparison is performed, which is carried out in three steps in AMOR. First, the revised models are each *matched* with the common original model V_o. Therefrom, two *match models* are obtained, which describe the correspondences among the original model and the revised models. Next, the applied *atomic operations* are computed. Besides these atomic operations, AMOR also provides techniques for detecting *composite operations*, such as model refactorings [109], among the applied atomic operations. The output of this phase of the process are two difference models $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$, which describe all operations performed in the concurrent modifications. The operation detection is discussed more precisely in Section 5.3.

Based on the two difference models computed in the previous phase of the process, the next phase of the process aims to detect *conflicts* among the concurrently applied operations. Thereby not only *atomic operation conflicts* (e.g., delete-update conflicts), but also conflicts among *composite operations* are revealed in the respective steps of this phase. All detected conflicts are saved into

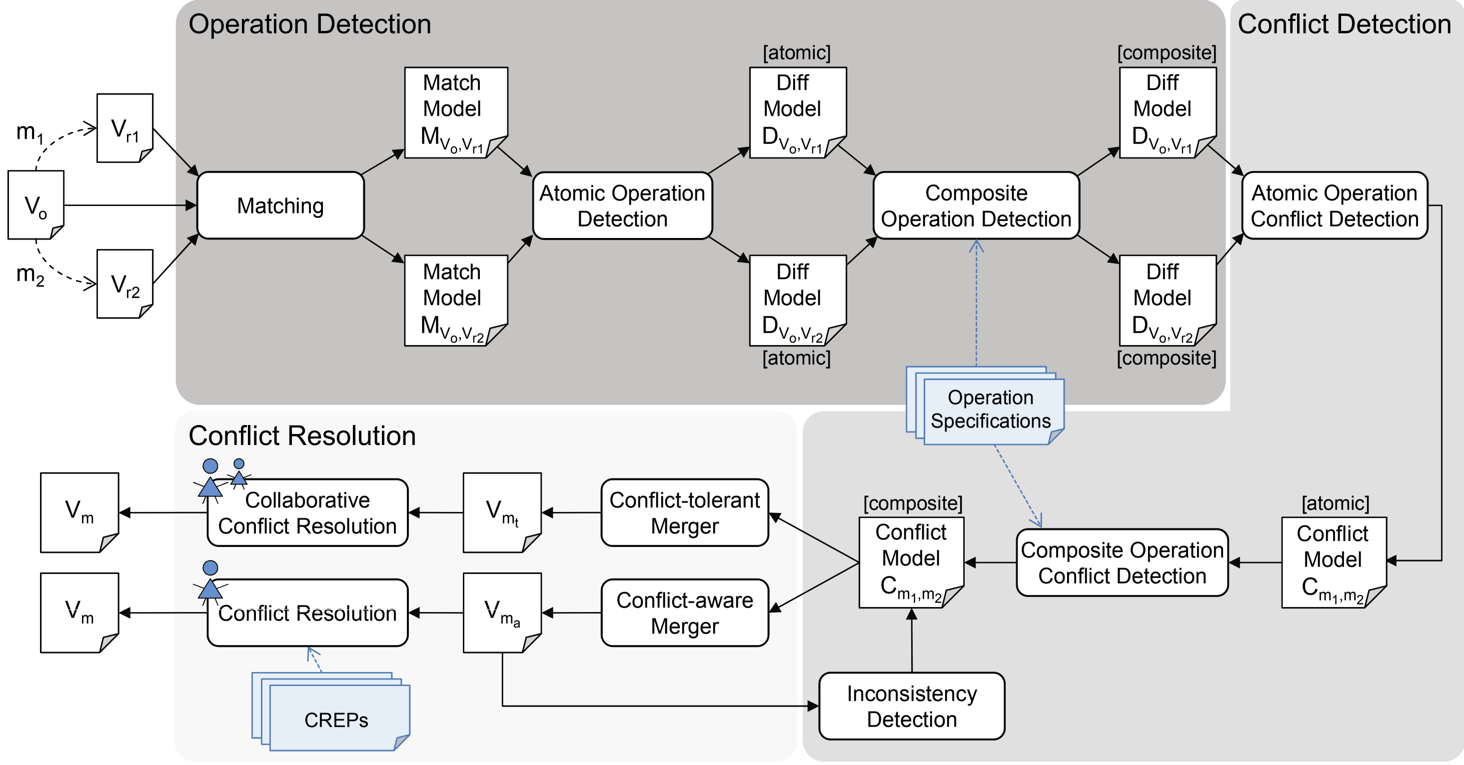


Fig. 8. AMOR Merge Process

a *conflict model* called C_{m_1,m_2} in Fig. 8. More information on how conflicts are detetected in AMOR is provided in Section 5.4.

The computed differences and detected conflicts serve then as input for the *conflict resolution* phase. AMOR's conflict resolution process may be adapted and provides two interchangeable strategies. Both strategies combine the strength of automatic merging and inconsistency toleration and calculate a *tentative merge*, which is discussed in Section 5.5. The tentative merge acts as base for either *collaborative* conflict resolution or for *recommendation supported* conflict resolution, as elaborated in Section 5.6.

5.3 Operation Detection

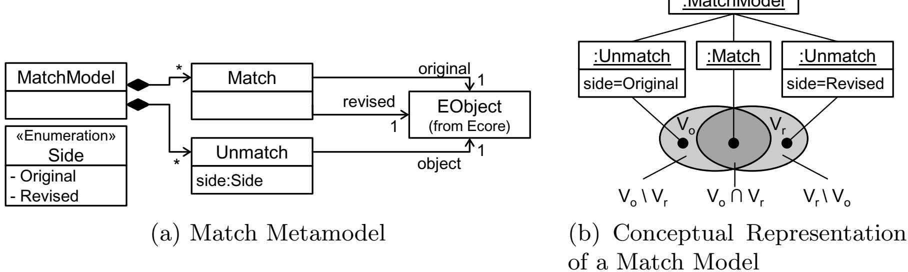
The first phase of the merge process is the operation detection with the goal to detect precisely which operations have been applied in between V_o and V_{r1}, as well as V_o and V_{r2}. This phase consists of three steps, model *matching*, *atomic operation detection*, and *composite operation detection* (cf. Fig. 8), which are discussed in the following.

Model Matching. AMOR aims to be independent from the modeling editor. Hence, state-based model differencing is applied. The first step of model differencing is model matching, which computes the corresponding model elements between the original model V_o and the revised models V_{r1} and V_{r2}. The computed correspondences are saved in two distinct match models $M_{V_o,V_{r1}}$ and $M_{V_o,V_{r2}}$. Therein, one correspondence connects a model element in the original model V_o with its corresponding revised model in V_{r1} or V_{r2}, respectively.

Computing correspondences between model elements. Even if UUID-based matching is probably the most efficient and straightforward technique for obtaining the actual model changes, there are some drawbacks of this approach. In particular, if model elements loose their UUID, they cannot be matched anymore. Unfortunately, such a scenario is happening quite frequently; not only because the developer deletes and re-creates a similar model element subsequently, but also because of improperly implemented copy & paste or move actions in certain modeling editors causing the model elements' UUIDs to be lost (e.g., in the tree-based Ecore editor).

To address these drawbacks, we apply a two-step matching process: first, a UUID-based matching is applied to obtain a base match, which is improved subsequently by applying user-specified language-specific match rules to the pairs of model elements that could not be matched based on their UUIDs. Thereby, the advantages of UUID-based matching is retained and its drawbacks are reduced significantly. As the comparatively slow rule-based matching is kept at a minimum with this approach, the additional execution time of the model matching phase should still be reasonable.

Representing correspondences. Having obtained the model element correspondences between the original model V_o and a revised model, called V_r to refer



(a) Match Metamodel

(b) Conceptual Representation of a Match Model

Fig. 9. Representing Model Correspondences

to both V_{r1} and V_{r2}, they have to be represented in some way for their further usage. Therefore, we introduce the match metamodel depicted in Fig. 9a. Please note that this match metamodel is largely equivalent to the one used in EMF Compare [14]. Basically, a match model is a so-called *weaving model* [31], which adds additional information to two existing models by introducing new model elements that refer to the model elements in the original and the revised model. In particular, a match model comprises an instance of the class MatchModel, which contains, for each pair of matching model elements, an instance of the class Match. This instance refers to the corresponding model element in the original version through the reference original and the revised version through the reference revised. If a model element, either in the original model and in the revised model, could not be matched, an instance of the class Unmatch is created, which refers to the unmatched model element in the respective model. The attribute side indicates whether the unmatched model element resides in the original or the revised model.

A match model groups the model elements in V_o and V_r into three distinct sets (cf. Fig. 9b). The first set constitutes all model elements that are contained in the original version, but not in the revised version (i.e., $V_o \setminus V_r$). The second set contains all model elements that are contained in both models (i.e., $V_o \cap V_r$) and the third set comprises all model elements that are contained in the revised model but not in the original model (i.e., $V_r \setminus V_o$). In EMF, attribute and reference values of a model element are possessed by the respective model element. Thus, they are considered as being a property of the model element rather than being treated as its own entity. Consequently, in the match model only corresponding *model elements* are linked by Match instances.

Atomic Operation Detection. Having obtained the correspondences among model elements in an original model V_o and a revised model V_r, we may now proceed with deriving the atomic operations that have been applied by the user to V_o in order to create V_r. As already mentioned, match models only indicate the corresponding model elements and those model elements that only exist either in V_o or in V_r. Corresponding model elements, however, might not be entirely equal as their attribute values or reference value might have been modified. Therefore,

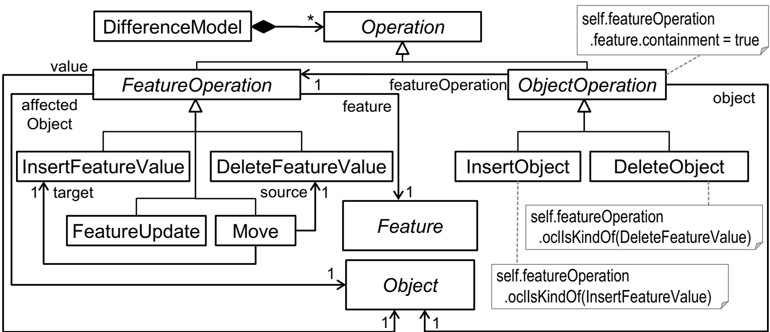


Fig. 10. Difference Metamodel

we further derive a *diff model* from the match model to also represent operations affecting attribute values and reference values before we may search for conflicts among concurrently performed operations. To put the detection of atomic operations in the context of a model versioning scenario, recall that we have two modifications, m_1 and m_2, and one match model comprising the correspondences for each side, $M_{V_o,V_{r1}}$ and $M_{V_o,V_{r2}}$. Therefore, we also have two *diff models* (cf. Fig. 8). In particular, $D_{V_o,V_{r1}}$, which is computed from $M_{V_o,V_{r1}}$, represents the operations applied in m_1 and $D_{V_o,V_{r2}}$, derived from $M_{V_o,V_{r2}}$, represents the operations applied in m_2.

Computing operations from corresponding model elements. Starting from a match model, the detection of applied operations is largely straightforward. In particular, the atomic operation detection component first iterates through all Match instances of this match model and performs a fine-grained feature-wise comparison of the two corresponding model elements. Thereby, the feature values of each feature of both corresponding model elements are checked for equality. If a feature value of one model element differs from the respective value of the corresponding model element, the respective feature of the model element has been subjected to a modification in the revision. After all Match instances have been processed, we proceed with iterating through all Unmatch instances. Depending on its value at the attribute side, we either encounter an *addition* of a new model element if the side is Revised, or a *deletion* if the side is Original.

Representing operations. To represent the applied operations, we introduce a difference model, which is depicted in Fig. 10. Due to space limitations, we only present the *kernel* of the difference metamodel in this paper, which does not reflect more advanced modeling features of EMF models, such as ordered features. For a complete specification of differences between EMF models, we kindly refer to Section 5.2.2 in [56]. In this kernel difference model, we distinguish between two types of operations: FeatureOperation, which modifies the value of a feature, and ObjectOperation, which represent the insertion or deletion of a

model element. Please note that model elements are referred to as *objects* in this metamodel for the sake of generalization.

If the respective features are multi-valued, values can be inserted or deleted from the feature. For expressing such operations, we use two concrete subclasses of FeatureOperation in the difference metamodel, namely InsertFeatureValue and DeleteFeatureValue. If feature values are single-valued, it is not possible to add or delete feature values. Instead, they only can be *updated*, whereas the old value is overwritten. Therefore, we introduce the operation type FeatureUpdate. If the respective feature is defined to be a containment feature, it may contain other model elements. In this case, model elements may also be *moved* from one container to another container, whereas the identity of the moved model element is retained. Such an operation is represented by the class Move, which links the deletion of it in the source feature and the insertion of it in the target feature. All types of feature operations refer to the object that has been changed using the reference affectedObject, to the affected feature in the modeling language's metamodel (reference affectedFeature), and to the inserted or deleted feature value (reference value). In case of a reference, this value is a model element and in case of an attribute, the value is a simple data type such as String, or Boolean, etc. However, we omitted to distinguish explicitly between model elements and simply typed data values in Fig. 10 for the sake of readability. It is worth noting that, in case of a InsertFeatureValue, the reference value refers to the inserted value in the revised model (V_{r1} or V_{r2}) and, in case of a DeleteFeatureValue, it refers to the deleted value in the original model V_o.

Besides modifying feature values in existing objects, users may also insert and delete entire objects (i.e., model elements). Therefore, the metamodel contains the two classes InsertObject and DeleteObject, which are subclasses of ObjectOperation. Except for root objects, objects are always contained by another object through a *containment feature*. Consequently, inserting and removing an object is realized by a feature operation affecting the respective containment feature. Thus, object operations are further specified by a reference to the respective instance of a FeatureOperation, which gives information on the inserted or deleted object (reference value), the container of the inserted or removed object (reference affectedObject), and the containment feature through which the object is or originally was contained (reference affectedFeature). Certainly, as defined by the invariants in Fig. 10, a valid instance of InsertObject must refer to an instance of InsertFeatureValue and a valid instance of DeleteObject must refer to an instance of DeleteFeatureValue, whereas the affected feature has to be a containment feature.

Example of a difference model. To exemplify the difference metamodel, a concrete instance is depicted in terms of an object diagram in Fig. 11, which represents a subset of the operations applied by Sally in the example introduced in Section 5.1. Please note that we depicted the object diagrams representing the original and the revised model in gray for the sake of readability. This figure depicts an excerpt of the original UML class diagram V_o and Sally's revision V_{r2}, the respective difference model $D_{V_o,V_{r2}}$, as well as an excerpt of the metamodel

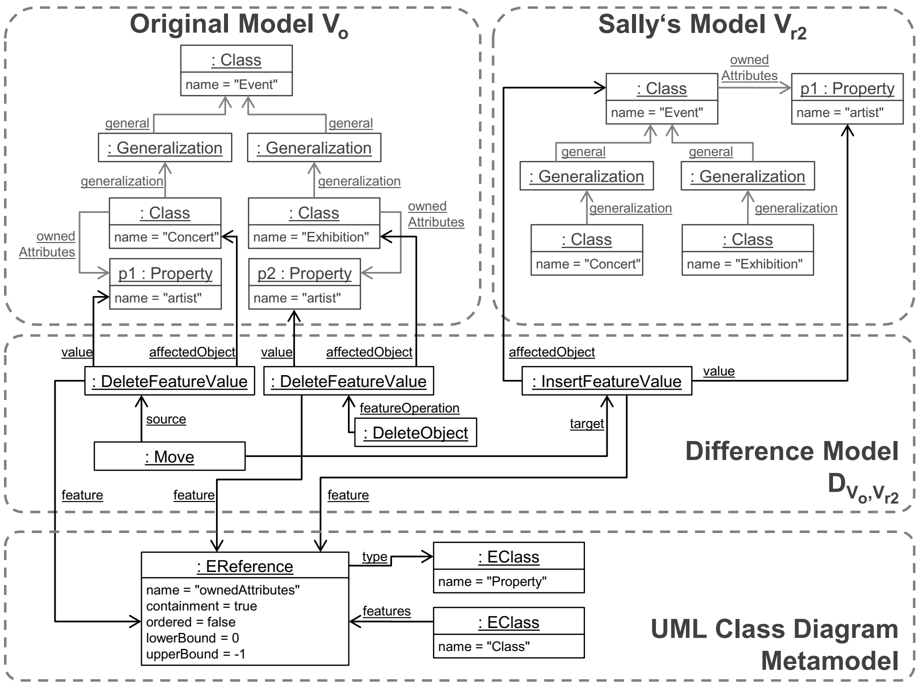


Fig. 11. Example of a Difference Model

for UML class diagrams. Sally moved the attribute artist from the class Concert to its superclass Event and deleted the equally named attribute artist from the class Exhibition. As a result, the difference model contains five operations. First of all, there is an instance of Move, which represents the shift of the attribute artist. As a move is realized by a deletion and a subsequent insertion of a feature value, the instance of Move refers to an instance of DeleteFeatureValue via the reference source and an instance of InsertFeatureValue with the reference target. Besides, Sally deleted the second attribute artist from the class Exhibition, which is represented by an instance of DeleteObject. This deletion is realized by another instance of DeleteFeatureValue referring to the deleted value (i.e., the deleted attribute artist), the affected object (i.e., the containing class Exhibition), as well as the UML metamodel feature the deleted value originally resided in (i.e., the feature ownedAttributes).

Composite Operation Detection. Having represented the applied atomic operations, we proceed with detecting applications of composite operations. Composite operations, such as model refactorings [109], pose specific pre- and postconditions and consist of a set of atomic operations that are executed in a transaction in order to fulfill one common goal. Thus, the knowledge on applications of composite operations between two versions of a model significantly helps in many scenarios to better respect the original intention of a developer, as well

as to reveal additional issues when merging two concurrent modifications [22,68]. Such an additional merge issue may occur, if concurrent modifications *invalidate* the preconditions of a composite operation or when concurrent modifications *influence* the execution of the composite operation.

Composite operations are specific to a certain modeling language. Thus, users should be empowered to pre-specify them for their employed modeling languages on their own in order to adapt the generic operation detection step of AMOR. Once a composite operation is specified and configured in AMOR, applications of composite operations, as well as conflicts concerning composite operations can be detected. However, specifying composite operations, or more generally, *endogenous model transformation*, is a challenging task, because users have to be capable of applying dedicated model transformation languages and have to be familiar with the metamodels of the models to be transformed. To ease the specification of such transformations, we make use of a novel approach called *model transformation by demonstration* (MTBD) [13,56,108], which is introduced briefly in the following.

Specifying composite operations. The general idea behind MTBD is that users apply or "demonstrate" the transformation to an example model once and, from this demonstration, as well as from the provided example model, the generic model transformation is derived semi-automatically. To realize MTBD, we developed the following specification process.

In a first step, the user creates the *initial model* in a familiar modeling environment. This initial model contains all model elements that are required in order to apply the composite operation. Next, each element of the initial model is annotated automatically with an ID, and a so-called *working model* (i.e., a copy of the initial model for demonstrating the composite operation by applying its atomic operations) is created. Subsequently, the user performs the complete composite operation on the working model by applying all necessary atomic operations. The output of this step is the *revised model*, which is together with the initial model the input for the following steps of the operation specification process. Due to the unique IDs, which preserve the relationship among model elements in the initial model and their corresponding model elements in the revised model, the atomic operations of the composite operation may be obtained precisely using our model comparison approach presented above. The obtained operations are saved in a *difference model*. Subsequently, an initial version of *pre-* and *postconditions* of the composite operation is inferred by analyzing the initial model and the revised model, respectively. The automatically generated conditions from the example might not always entirely express the intended pre- and postconditions of the composite operation. They only act as a basis for accelerating the operation specification process and may be refined by the user. In particular, parts of the conditions may be activated, deactivated, or modified within a dedicated environment. If needed, additional conditions may be added. After the configuration of the conditions, an operation specification is generated, which is a model-based representation of the composite operation consisting of the diff model and the revised pre- and postconditions, as well as the initial and

Excerpt of the Preconditions	Excerpt of the Postconditions
•**Class_0** [Event] •**Class_1** [Concert] • generalisations->includes(Generalisation_0) • ownedAttributes->includes(Property_0) •**Generalisation_0** • general = Class_0 • **Property_0** [artist] •**Class_2** [Exhibition] • generalisations->includes(Generalisation_1) • ownedAttributes->includes(Property_1) • **Generalisation_1** • general = Class_0 • **Property_1** [artist] • *name = Property_0.name*	•**Class_0** [Event] • **Property_0** [artist] •**Class_1** [Concert] • generalisations->includes(Generalisation_0) • ownedAttributes->includes(Property_0) •**Generalisation_0** • general = Class_0 •**Class_2** [Exhibition] • generalisations->includes(Generalisation_1) •**Generalisation_1** • general = Class_0

Legend: ... iteration

Fig. 12. Excerpt of the Pre- and Postconditions of "Pull Up Field"

revised example model. Thus, this model contains all necessary information for executing the composite operation, as well as for detecting applications of it. For more information on the specification process, and how these specifications can be executed automatically to arbitrary models, we kindly refer to [13,56,108].

Example for specifying "Pull Up Field". To provide a better understanding of how composite operations can be specified, we discuss the specification process for developing the composite operation "Pull Up Field". In the first step, the user creates the initial model, which contains all model elements that are necessary to apply the composite operation. The resulting initial model is equivalent to the original model depicted in Fig. 11. Now, a copy of this initial model is created automatically, to which the user now applies all atomic operations, which leads to the revised model again shown in Fig. 11. From these two models, we now compute the applied atomic operations using the previously discussed model comparison resulting in the difference model of our previous example presented in Fig. 11. Besides for computing the atomic operations, these two models also act as input for the automatic derivation of the pre- and postconditions in the form of OCL expressions [84], which may now be fine-tuned by the user. The resulting conditions, after the refinement by the user, are illustrated in Fig. 12. The pre- and postconditions are structured according to the model elements in the initial and revised model, respectively, and my refer to each other using a model element identifier (e.g., Class_0), which is comparable to a variable. The user only has to refine the precondition `name = Property_0.name` in order to restrict the names of the properties that are pulled up to the common superclass to be equal. In other words, the property Property_1, which resides in the second subclass Class_2 must have the same name as the other property Property_0, which is contained by the class Class_1, for acting as a valid property in this composite operation. Without satisfying this condition, the execution of the refactoring would lead to a change of the semantics of the model, because Class_2 would inherit a differently named property from its superclass after the refactoring has been applied. Besides fine-tuning this precondition, the user may also attach

iterations, which has been done for Property_0, Class_1, and Property_1. With these iterations, the user specifies that all atomic operations that have been applied to these model elements in the demonstration have to be repeated *for all* model elements that match the respective preconditions when applied to an arbitrary model.

Detecting composite operations. After the difference models are computed, we may proceed with the next step in the merge process, which aims at detecting applications of composite operations (cf. Fig. 8). The composite operation detection step relies on a set of composite operation specification. As mentioned above, an operation specification contains a description of the atomic operations applied during the demonstration of the composite operation, which can be thought of as the *difference pattern* of the composite operation. Besides this difference pattern, an operation specification also contains the composite operation's pre- and postconditions.

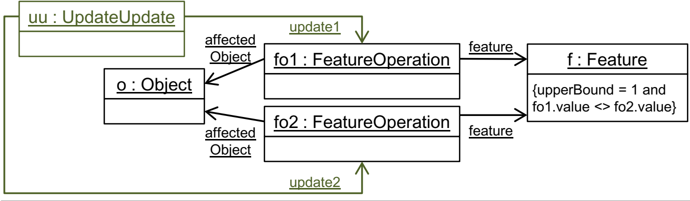
For detecting applications of composite operations, it is searched for occurrences of the composite operations' difference pattern in each of the difference models $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$. If a difference pattern could be found, the respective parts of the original model V_o are evaluated concerning the composite operation's preconditions. If also these preconditions are fulfilled in the original model, also the postconditions of the composite operation are verified for the corresponding model elements in the respective revised model. In case also the postconditions can be evaluated positively in the revised model, an application of the respective composite operation is reported and annotated in the difference model. For more information on how composite operations are detected in AMOR, we kindly refer to Section 5.3 in [56].

5.4 Conflict Detection

Having obtained the operations that have been applied concurrently to the common original model, we may now proceed with detecting conflicts among them. As discussed in Section 3.4, a conflict occurs if two operations do not commute or if one operation is not applicable anymore after the other operation has been performed. The conflict detection in AMOR takes two difference models, $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$, as input and is realized by two subsequent steps: the *atomic operation conflict detection* and the *composite operation conflict detection.*

Atomic operation conflict detection. The goal of the first step, the atomic operation conflict detection, is to find concurrent *atomic operations* that interfere with each other. This step is completely generic and does not demand for language-specific information. Also composite operations remain unconsidered in this step; however, the atomic operations that realize the composite operation applications are still included in the conflict detection mechanisms.

We define the types of atomic operation conflicts by so-called generic *conflict patterns.* These conflict patterns serve, on the one hand, as a clear specification of the existing conflict types, and, on the other hand, they can be used for




```
context UpdateUpdate
inv:  self.update1.affectedObject = self.update2.affectedObject and
      self.update1.feature = self.update2.feature and
      (self.update1.feature.upperBound = 1 and
      self.update1.value <> self.update2.value)
```

Fig. 13. Update-update Conflict Pattern

detecting conflicts. More precisely, if such a conflict pattern matches with two operations in the difference models, $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$, a conflict of the respective type occurred. However, for the sake of efficiency, we refrain from checking the complete crossproduct of all operation combinations among all operations of both difference models. In contrast, both difference models are translated in a first step into an optimized view grouping all operations according to their type into potentially conflicting combinations. Secondly, all combinations are filtered out if they do not spatially affect overlapping parts of the original model. Finally, all remaining combinations are checked in detail by evaluating the conflict rules.

An example for such a conflict pattern is provided in Fig. 13, which defines an *update-update conflict* in terms of an object diagram, as well as an OCL constraint. As already mentioned, if a single-valued feature is concurrently modified in EMF models a conflict occurs, because the merged model may not contain both values and one value overwrites the other; thus, two updates of the same feature value do not commute. Therefore, as illustrated in the conflict pattern in Fig. 13, an *update-update* conflict is raised, if an object o has been concurrently updated at the same feature f by two instances of FeatureOperation, fo1 and fo2, unless both operations set the same new value such that `fo1.value` = `fo2.value`. In our running example presented in Section 5.1, we encounter such a conflict. Harry and Sally both renamed the UML operation getInfo(). Since the name of UML operations is a single-valued feature and the new values for the name of the UML operation are different (getTInfo() vs. getTicketInfo()), an update-update conflict is raised.

All detected conflicts are saved into a conflict model called C_{m_1,m_2}, which is a model-based description of the occurred conflicts. Such a description provides the necessary information concerning the type of the conflict and the involved atomic operations. The complete set of all conflict patterns, as well as the conflict metamodel is discussed more profoundly in Section 6.1 of [56].

Composite operation conflict detection. The next step in the conflict detection phase is the composite operation conflict detection, which takes the knowledge on the ingredients of composite operations, such as preconditions, into account for

revealing additional conflicts. In particular, this step aims to detect scenarios in which modifications of one developer invalidate the preconditions of a composite operation that has been applied in a parallel revision by another developer.

To detect composite operation conflicts, each application of a composite operation in one revision (let us assume this is V_{r1}) is separately checked at the respective opposite revision V_{r2}. Therefore, we first identify the model elements of the opposite revision V_{r2} that correspond to the model elements in V_o to which the composite operation has been applied originally. Next, we evaluated the preconditions of the composite operation with the identified model elements of V_{r2}. If the preconditions are not fulfilled, the composite operation cannot be applied after the concurrent operations have been performed; thus, a conflict is raised and added to the conflict model C_{m_1,m_2}.

Such a conflict is illustrated in the running example presented in Section 5.1. Sally applied the refactoring “Pull Up Field” by moving the property artist from the classes Concert and Exhibition to their common superclass Event. As discussed in Section 5.3, a precondition of this composite operation is that each subclass must contain a property having the same name as the property that is moved to the common superclass. However, in the concurrent revision, Harry deleted the property artist and added another subclass named SoccerMatch without having a property named artist. Thus, the composite operation is not applicable to the revised version of Harry, because no valid match could be found for the preconditions of the refactoring. Consequently, a composite operation conflict, which is also referred to as *composite operation contract violation*, is added to the conflict model C_{m_1,m_2}.

5.5 Merging

With the conflict model C_{m_1,m_2} at hand, we may now proceed with merging the parallel evolved models. Merging is the intricate task of usually one developer, i.e., the developer who performs the later check-in, of integrating all changes into one consolidated version of the model. However, several strategies exist how merging may be realized, as discussed in Section 3.5. To justify AMOR’s claim for adaptability not only with respect to supported modeling languages and detectable conflicts, also the merge process may be configured. The overall goal is to support the developer in understanding the evolution of the other developer’s version as well as how this version contradicts her own changes to the model. In fact, AMOR provides two interchangeable merge strategies, each tailored to specific needs of a project’s stage.

Conflict-tolerant merging is adopted in the early phases of a project, i.e., analysis phase and design phase, where a common perception of the system under study is not yet established. These phases are considered critical, as mistakes are likely to happen and the costs of fixing such errors are high, when detected in later phases [37]. Thus, in order to keep all viewpoints on the system, conflicts are not resolved immediately after each commit, but are tolerated until a specified milestone is reached. Then, to minimize the risk of losing any modifications,

all developers may resolve conflicts together in a meeting or with the help of a tool-supported collaborative setting as proposed in [118,12].

Conflict-aware merging is primary designed to support merging of a single developer after each commit. If conflicts are not collaboratively resolved, it is even more important, that the developer in charge of merging is supported to effectually understand the model's evolution. In a manual merge process, the developer has to navigate through several artifacts to collect information necessary to comprehend the intentions behind all operations and the reasons of occurred conflicts. To support this process, we combine automatic merge strategies with the benefits of pollution markers known from the field of tolerating inconsistencies [4,74,102]. We therefore calculate an automatically merged version which reveals all operations and conflicts at a single glance by introducing dedicated annotations. This automatically merged version acts as basis for conflict resolution and is thus denoted *tentative merge*.

In the remainder of this section, we present details how the tentative merge is calculated employing the conflict-aware merge strategy of AMOR.

Design Rationale. In order to fully exploit the abstraction power of models, modeling languages, such as UML, are usually complemented with a graphical concrete syntax to hide the complexity of the abstract syntax. As developers are mostly used to the graphical concrete syntax only, merging shall also be performed directly using the concrete syntax of models. The major goal of conflict-aware merging is to provide the model's evolution in a single graphical view without losing any model elements or modifications. Our design rationale is based on the following requirements.

- *User-friendly visualization.* Information about performed operations and resulting merge conflicts shall be presented in the concrete syntax of the model retaining the original diagram layout.
- *Integrated view.* All information necessary for the merge shall be visualized within a single diagram to provide a complete view on the models evolution.
- *Standard conform models.* The models incorporating the merge information shall be conform to the corresponding metamodel without requiring heavyweight modifications.
- *Model-based representation.* The merge information shall be explicitly represented as model elements to facilitate model exchange between modeling tools, as well as postponing the resolution of certain conflicts.
- *No editor modifications.* The visualization of the merge information shall be possible without modifying the graphical editors of modeling tools.

In the following, we elaborate on the technical details of the conflict-aware merge strategy with respect to the mentioned requirements.

Model Versioning Profile. As mentioned above, we use annotations to mark conflicting or inconsistent parts of the merged model. Those conflicts are tolerated to a certain extent and eventually corrected. In our case, conflicts are tolerated during the merge phase.

Annotations extend the model and carry information. Hence, annotations need an appropriate representation in the modeling language's abstract and concrete syntaxes. However, creating new modeling languages goes hand in hand with building new editors and code generators, as well as preparing documentation and teaching materials, among others. Further, several modeling languages have already matured and may not be neglected when setting up versioning support. Thus, mechanisms are needed to *customize existing languages*. When directly *modifying existing languages*, the aforementioned issues remain unsolved, as newly introduced metamodel elements cannot be parsed by existing editors. According to [3], a lightweight language customization approach is desirable. For customizing immutable modeling languages like UML, *UML profiles* are the means of choice. UML profiles provide a language-inherent, non-intrusive mechanism for dynamically adapting the existing language to specific needs. As UML profiles are not only part of UML, but defined in the infrastructure specification [85], various modeling languages, which are defined as instance of the common core may be profiled and thus dynamically tailored. 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 conforming to UML, that they are naturally handled by current UML tools. Recently, in an endeavor to broaden the idea of UML profiles to modeling languages based on implementations of Essential MOF [77], such as Ecore [23], several works have been published [51,57,61]. The profiling mechanism inherently reflects our design rationale and is thus our means of choice. UML profiles are standardized by the OMG and act as conceptual role model for Ecore based implementations. Thus, we discuss the annotations for the conflict-aware merge based on UML profiles in the following.

The information on detected operations and conflicts is already available in the difference models $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$, as well as the conflict model C_{m_1,m_2}, as described in Section 5.3 and Section 5.4, respectively. However, we assemble the difference and conflict models and generate a dedicated *model versioning profile* to realize the gluing of the available information into the model. Thus, the versioning profile reflects the separation on changes and conflicts and explicates additional information on the respective users, which was only implicitly available beforehand. An excerpt of the versioning profile is depicted in Fig. 14. Detailed information on the versioning profile may be found in Section 5.3 in [11].

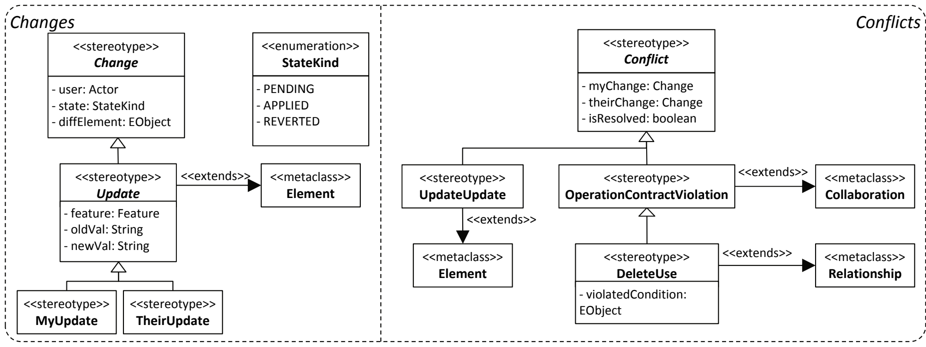


Fig. 14. Excerpt of the Model Versioning Profile

Changes. The versioning profile provides stereotypes for each kind of atomic operation, i.e., ≪Add≫, ≪Delete≫, ≪Update≫, ≪Move≫, and a stereotype for composite operations, i.e., ≪CompositeChange≫. Each change stereotype is specialized as ≪MyChange≫ and ≪TheirChange≫ to explicate the user who performed the change. An atomic change is always performed on a single UML element, i.e., `Class`, `Generalization`, `Property`, etc., and thus, is defined to extend `Element`, the root metaclass of the UML metamodel (cf. stereotype ≪Update≫ in the left part of Fig. 14). In contrast, a composite change incorporates a set of indivisible atomic operations. Therefore, we introduce a new UML `Collaboration` interlinking the involved elements, which is annotated with a ≪CompositeChange≫ stereotype. Further metadata regarding the respective users, the application state of the change, and the affected feature of the changed element including its old and new value in case of updates is stored in tagged values.

Conflicts. The conflict part of the versioning profile defines stereotypes for all conflict patterns subsumed in the conflict metamodel. Again, stereotypes for overlapping operations regarding a single element, such as ≪UpdateUpdate≫ in the right part of Fig. 14, extend the metaclass `Element`, while violations, like an ≪OperationContractViolation≫ comprise different modeling elements and are thus annotated on newly introduced `Collaboration` elements. 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 [55]) indicating how the contract is violated by the model element. The stereotype ≪DeleteUse≫ are applied on model elements already existing in the original model, which are involved in a composite operation and 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.

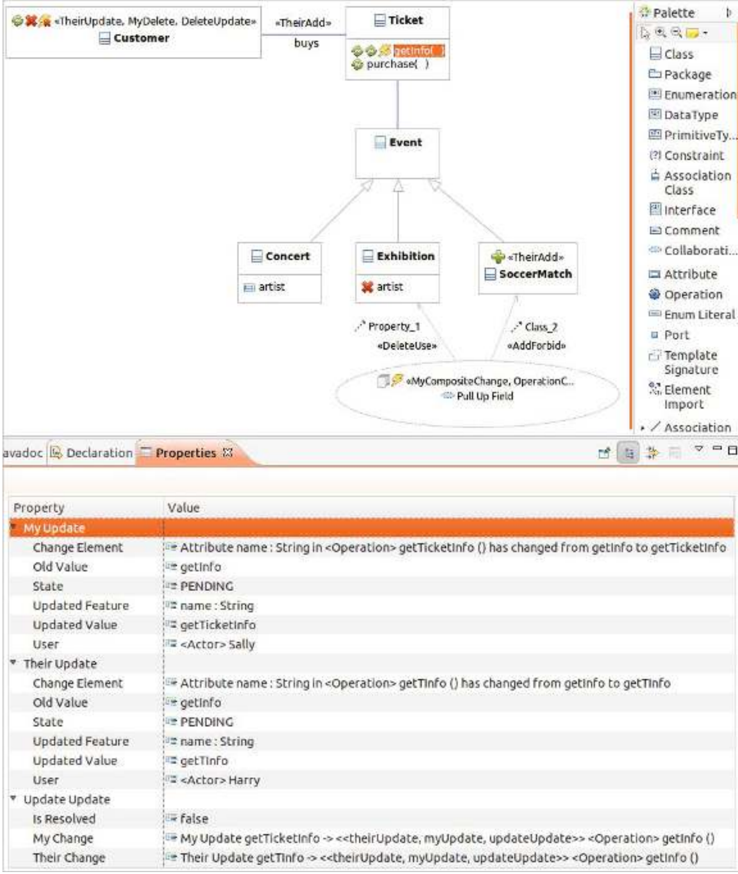


Fig. 15. Tentative Merge

Merge Algorithm. With the versioning profile at hand, we may now proceed with merging the parallel evolved versions V_{r1} and V_{r2}. The conflict-aware merge strategy automatically calculates a tentatively merged version V_{m_a}, by embracing all non-conflicting operations in an element preserving manner. Additionally, conflicting operations are not ignored, but integrated via dedicated stereotypes of the versioning profile. The merge algorithm takes the common original model V_o, the difference models $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$, and the conflict model C_{m_1,m_2} as input and produces a tentative merge, as depicted in Fig. 15. Details on merging the concrete syntax of the models may be found in Chapter 5 in [11]. The algorithm for merging the abstract syntax proceeds as follows. In order to keep the original model V_o untouched, it is initially copied to the output model V_{m_a}.

1. *Merge atomic operations.* For each atomic change of $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$, it is checked, whether the change is involved in a conflict pattern described in the conflict model C_{m_1,m_2}, or whether it is considered non-conflicting. If

the change is non-conflicting and non-deleting, it is applied to the tentative merge V_{m_a} and annotated with the respective stereotype. In the running example of Fig. 7, the renaming of the operation buy() to purchase() performed by Sally is executed and annotated as ≪MyUpdate≫, as Sally did the later check-in and the merge is performed reflecting her viewpoint. The corresponding metadata of the difference model is stored in tagged values of the stereotype. Additionally, the tagged value user is set to Sally and the state value is set to APPLIED. Even if deletions are non-conflicting, they are not yet executed, as they otherwise cannot be annotated and information would get lost. Similarly, conflicting operations are annotated with state value set to PENDING without applying them to the tentative merge. Thus, the class Person deleted by Sally and renamed to Customer by Harry is annotated with the stereotypes ≪TheirUpdate≫ and ≪MyDelete≫ in this step.

2. *Annotate overlapping operations.* After processing all atomic operations, conflicts due to overlapping atomic operations are annotated. In the running example, the operation getInfo() of class Ticket is concurrently renamed resulting in an update-update conflict. Thus, the operation is annotated with an ≪UpdateUpdate≫ stereotype, which links to the respective change stereotypes via tagged values. Further, it the tagged value indicating the resolution state is set to `false`, as depicted in Fig. 15.
3. *Merge composite operations.* To express that composite operations consists of an indivisible unit of atomic operations, a UML `Collaboration` is added to the tentative merge, which links to the involved elements and is annotated with a ≪CompositeChange≫ stereotype. If no composite operation conflict is reported in the conflict model C_{m_1,m_2}, the composite change is replayed to the tentative merge and set to APPLIED. Otherwise, it is still PENDING.
4. *Annotate operation contract violations.* If a composite operation conflict is at hand, a ≪CompositeOperationConflict≫ stereotype is attached to the collaboration. Further, the relationships linking to the affected model elements of conflicting operations are named according to the the operation specification's template names and subsequently annotated. In the running example, the deletion of the property artist in class Exhibition (cf. Property_1 in Fig. 12) and the added class SoccerMatch (Class_2) violate the precondition of the refactoring Pull Up Field. Thus, the relationships are annotated with ≪DeleteUse≫ and ≪AddForbid≫, respectively.
5. *Validate model.* Next, the tentative merge is then validated and, if constraints are violated, the violation is added to the conflict report C_{m_1,m_2}.
6. *Annotate constraint violations.* Finally, the detected violations are annotated in the tentative merge, by again introducing UML `Collaboration` elements linking to the model elements involved in the violation.

5.6 Conflict Resolution

In AMOR, conflict resolution is performed on top of the tentative merge. The stereotypes included in the tentative merge may be further exploited to provide dedicated tooling support for conflict resolution. For example, resolution actions

in form of "take my change", "take their change", or "revert this change" may be easily implemented, as the link to the difference model is retained. However, as the same kind of conflicts are likely to reoccur, AMOR provides a recommender system for conflict resolution, which suggests even Conflict Resolution Patterns going beyond a combination of applied changes. These patterns are stored in AMOR's conflict resolution pattern repository and are currently predefined in the same manner as composite operations (cf. Section 5.3).

Coming back to the running example, Sally has to resolve several conflicts, which are annotated in the tentative merge. She starts resolving the delete-update conflict annotated in the class Customer. As there is currently no conflict resolution pattern stored in the conflict resolution recommender system for the resolution of delete-update conflicts, only the choices of applying one of the respective changes is available. Sally decides to revert her delete operation. The application states stored in the tagged values of the overlapping delete and update operations are automatically set to REVERTED and APPLIED, respectively. Further, the tagged value isResolved of the ≪DeleteUpdate≫ stereotype is set to `true`. She continues with the update-update conflict of the operation getInfo(), where she prefers her change. Now, only the composite operation conflict remains left. Sally disagrees with Harry's opinion that exhibitions do not have an artist, and she reverts his change, resulting in a removal of the delete-use conflict. The conflict resolution recommender system announces, that a conflict resolution pattern is found for the remaining add-forbid conflict. The resolution is performed by introducing a new class into the inheritance hierarchy. The new class gets subclass of the class Event and superclass of the classes Concert and Exhibition. In this way, it displaces the original superclass as target for the Pull Up Field refactoring. Sally has only to provide a name for the new class and is confident with this solution. Finally, she commits the resolved version to the repository.

6 Open Challenges

In this paper, we introduced the fundamental technologies in the area of model versioning and surveyed existing approaches in this research domain. Besides, we gave an overview on the model versioning system AMOR and showcased its techniques based on a model versioning example. Although the active research community accomplished remarkable achievements in the area of model versioning in the last years, this research field still poses a multitude of interesting open challenges to be addressed in future, which we outline in the following.

Intention-aware model versioning. When merging two concurrently modified versions, ideally the merged version represents a combination of *all intentions* each developer had in mind when performing their operations. Merging intentions is usually more than just naively combining all non-conflicting atomic operations of both sides. When a developer modifies a model, this is done in order to realize a certain goal rather than simply modifying some parts of it. However, capturing the developer's intention from a set of operations is a major challenge.

A first step in this direction is respecting the original intention of composite operations, such as model refactorings, in the merge process. Composite operations constitute a set of cohesive atomic operations that are applied in order to fulfill one common goal. Therefore, detecting applications of well-defined composite operations and regarding their conditions and intention during the merge is a first valuable step towards intention-aware versioning [22,56,68].

However, further means for capturing and respecting the intention of applied operations may be investigated, such as allowing developers to annotate his/her intention for a set of applied operations in a structured and automatically verifyable manner. For instance, a developer might want to change a metamodel in order to limit its instatiability. Thus, an "issue witness" in terms of an instance model that should not be valid anymore can be annotated to give the set of applied operations more meaning. After merging concurrent operations, the versioning system may verify whether the original intention (i.e., the non-instatiability of the issue witness) is still fulfilled.

Semantics-aware model versioning. Current model versioning systems mainly facilitate matching and differencing algorithms operating on the syntactic level only. However, syntactical operations that are not conflicting may still cause semantic issues and unexpected properties (e.g., deadlocks in behavior diagrams). Thus, a combination of syntactic and semantic conflict detection is highly valuable but very challenging to achieve, because currently no commonly agreed formal semantics exists for widespread employed modeling languages, such as UML. First approaches for performing semantic differencing are very promising [62,63,64,73]. As these approaches focus only on two-way comparison and operate on a restricted set of modeling languages and constructs, the application of semantic differencing techniques in model versioning systems is not directly possible yet and the definition of a formal semantics for a comprehensive set of the UML, including intra-model dependencies, is a challenge on its own. Furthermore, as models may be used as sketch in the early phases of software development, as well as for specifying systems precisely to generate code, a satisfactory compromise has to be found to do justice to the multifaceted application fields of modeling.

Overall, we conclude this tutorial with the observation that the research area of model versioning still offers a multitude of tough challenges despite the many achievements which have been made until today. These challenges must be overcome in order to obtain solutions which ease the work of the developers in practice. The final aim is to establish methods which are so well integrated in the development process that the developers themselves do not have to care about versioning tasks and that they are not distracted from their actual work by time consuming management activities. Therefore, different facets of the modeling process itself have to be reviewed to gain a better understanding of the inherent dynamic. Versioning is about supporting team work, i.e., about the management of people who work together in order to achieve a common goal. Consequently, versioning solutions require not only the handling of technical issues like adequate differencing and conflict detection algorithms or adequate

visualization approaches, but also the consideration of social and organizational aspects. Especially in the context of modeling, the current versioning approaches have to be questioned and eventually revised. Here the requirements posed on the versioning systems may depend on the intended usage of the models.

References

1. Alanen, M., Porres, I.: Difference and Union of Models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
2. Altmanninger, K., Brosch, P., Kappel, G., Langer, P., Seidl, M., Wieland, K., Wimmer, M.: Why Model Versioning Research is Needed!? An Experience Report. In: Proceedings of the MoDSE-MCCM 2009 Workshop @ MoDELS 2009 (2009)
3. Atkinson, C., Kühne, T.: A Tour of Language Customization Concepts. Advances in Computers 70, 105–161 (2007)
4. Balzer, R.: Tolerating Inconsistency. In: Proceedings of the 13th International Conference on Software Engineering (ICSE 1991), pp. 158–165. IEEE (1991)
5. Barrett, S., Butler, G., Chalin, P.: Mirador: A Synthesis of Model Matching Strategies. In: Proceedings of the International Workshop on Model Comparison in Practice (IWMCP 2010), pp. 2–10. ACM (2010)
6. Berners-Lee, T., Hendler, J.: Scientific Publishing on the Semantic Web. Nature 410, 1023–1024 (2001)
7. Bézivin, J.: On the Unification Power of Models. Software and Systems Modeling 4(2), 171–188 (2005)
8. Bézivin, J.: From Object Composition to Model Transformation with the MDA. In: Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS 2001), pp. 350–355. IEEE (2001)
9. Bézivin, J., Gerbé, O.: Towards a Precise Definition of the OMG/MDA Framework. In: Proceedings of the 16th Annual International Conference on Automated Software Engineering, ASE 2001, pp. 273–280 (2001)
10. Booch, G., Brown, A.W., Iyengar, S., Rumbaugh, J., Selic, B.: An MDA Manifesto. MDA Journal (5) (2004)
11. Brosch, P.: Conflict Resolution in Model Versioning. Ph.D. thesis, Vienna University of Technology (2012)
12. Brosch, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kappel, G.: Concurrent Modeling in Early Phases of the Software Development Life Cycle. In: Kolfschoten, G., Herrmann, T., Lukosch, S. (eds.) CRIWG 2010. LNCS, vol. 6257, pp. 129–144. Springer, Heidelberg (2010)
13. Brosch, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In: Schürr, A., Selic, B. (eds.) MoDELS 2009. LNCS, vol. 5795, pp. 271–285. Springer, Heidelberg (2009)
14. Brun, C., Pierantonio, A.: Model Differences in the Eclipse Modeling Framework. UPGRADE, The European Journal for the Informatics Professional 9(2), 29–34 (2008)
15. Church, A., Rosser, J.: Some Properties of Conversion. Transactions of the American Mathematical Society, 472–482 (1936)

16. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. Journal of Object Technology 6(9), 165–185 (2007)
17. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing Model Conflicts in Distributed Development. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MoDELS 2008. LNCS, vol. 5301, pp. 311–325. Springer, Heidelberg (2008)
18. Conradi, R., Westfechtel, B.: Version Models for Software Configuration Management. ACM Computing Surveys 30(2), 232 (1998)
19. De Lucia, A., Fasano, F., Oliveto, R., Tortora, G.: ADAMS: Advanced Artefact Management System. In: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR 2006), pp. 349–350. IEEE (2006)
20. De Lucia, A., Fasano, F., Scanniello, G., Tortora, G.: Concurrent Fine-Grained Versioning of UML Models. In: Proceedings of the European Conference on Software Maintenance and Reengineering, pp. 89–98. IEEE (2009)
21. Dig, D., Comertoglu, C., Marinov, D., Johnson, R.: Automated Detection of Refactorings in Evolving Components. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 404–428. Springer, Heidelberg (2006)
22. Dig, D., Manzoor, K., Johnson, R., Nguyen, T.: Effective Software Merging in the Presence of Object-Oriented Refactorings. IEEE Transactions on Software Engineering 34(3), 321–335 (2008)
23. Eclipse: Eclipse Modeling Framework Project (EMF), `http://www.eclipse.org/modeling/emf` (accessed November 04, 2011)
24. Eclipse: EMF UML2, `http://www.eclipse.org/modeling/mdt/?project=uml2` (accessed December 05, 2011)
25. Edwards, W.K.: Flexible Conflict Detection and Management in Collaborative Applications. In: Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology, UIST 1997, pp. 139–148. ACM (1997)
26. Ehrig, H.: Introduction to the Algebraic Theory of Graph Grammars (A Survey). In: Ng, E.W., Ehrig, H., Rozenberg, G. (eds.) Graph Grammars 1978. LNCS, vol. 73, pp. 1–69. Springer, Heidelberg (1979)
27. Ehrig, H., Ehrig, K.: Overview of Formal Concepts for Model Transformations Based on Typed Attributed Graph Transformation. Electronic Notes in Theoretical Computer Science 152, 3–22 (2006)
28. Ehrig, H., Ermel, C., Taentzer, G.: A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications. In: Giannakopoulou, D., Orejas, F. (eds.) FASE 2011. LNCS, vol. 6603, pp. 202–216. Springer, Heidelberg (2011)
29. Ermel, C., Rudolf, M., Taentzer, G.: The AGG Approach: Language and Environment. In: Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools, ch. 14, vol. 2, pp. 551–603. World Scientific Publishing Co., Inc. (1999)
30. Estublier, J., Leblang, D., Hoek, A., Conradi, R., Clemm, G., Tichy, W., Wiborg-Weber, D.: Impact of Software Engineering Research on the Practice of Software Configuration Management. ACM Transactions on Software Engineering and Methodology (TOSEM) 14(4), 383–430 (2005)
31. Fabro, M., Bézivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: A Generic Model Weaver. In: Proceedings of the 1re Journe sur l'Ingnierie Dirige par les Modles, IDM 2005 (2005)

32. Feather, M.: Detecting Interference When Merging Specification Evolutions. In: Proceedings of the International Workshop on Software Specification and Design (IWSSD 1989), pp. 169–176. ACM (1989)
33. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)
34. France, R., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: Proceedings of Future of Software Engineering @ ICSE 2007, pp. 37–54 (2007)
35. Gerth, C., Küster, J.M., Luckey, M., Engels, G.: Precise Detection of Conflicting Change Operations Using Process Model Terms. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MoDELS 2010. LNCS, vol. 6395, pp. 93–107. Springer, Heidelberg (2010)
36. Gerth, C., Küster, J., Luckey, M., Engels, G.: Detection and Resolution of Conflicting Change Operations in Version Management of Process Models. Software and Systems Modeling, pp. 1–19 (Online First)
37. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering, 2nd edn. Prentice Hall PTR, Upper Saddle River (2002)
38. Greenfield, J., Short, K.: Software Factories: Assembling Applications With Patterns, Models, Frameworks and Tools. In: Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA 2003), pp. 16–27. ACM (2003)
39. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of Semantics? Computer 37(10), 64–72 (2004)
40. Heckel, R., Küster, J., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 161–176. Springer, Heidelberg (2002)
41. Herrmannsdoerfer, M., Koegel, M.: Towards a Generic Operation Recorder for Model Evolution. In: Proceedings of the International Workshop on Model Comparison in Practice @ TOOLS 2010. ACM (2010)
42. International Organization for Standardization and International Electrotechnical Comission: Information Technology—Syntactic Metalanguage—Extended BNF 1.0 (December 1996), http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip
43. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
44. Kehrer, T., Kelter, U., Taentzer, G.: A rule-based approach to the semantic lifting of model differences in the context of model versioning. In: Proceedings of the International Conference on Automated Software Engineering (ASE 2011). IEEE (2011)
45. Kelter, U., Wehren, J., Niere, J.: A Generic Difference Algorithm for UML Models. In: Software Engineering, pp. 105–116. LNI, GI (2005)
46. Khuller, S., Raghavachari, B.: Graph and network algorithms. ACM Computing Surveys 28(1), 43–45 (1996)
47. Kim, M., Notkin, D.: Program Element Matching for Multi-version Program Analyses. In: Proceedings of the International Workshop on Mining Software Repositories (MSR 2006). ACM (2006)
48. Koegel, M., Herrmannsdoerfer, M., Wesendonk, O., Helming, J.: Operation-based Conflict Detection on Models. In: Proceedings of the International Workshop on Model Comparison in Practice @ TOOLS 2010. ACM (2010)

49. Koegel, M., Herrmannsdoerfer, M., von Wesendonk, O., Helming, J., Bruegge, B.: Merging Model Refactorings – An Empirical Study. In: Proceedings of the Workshop on Model Evolution @ MoDELS 2010 (2010)
50. Kolovos, D.S.: Establishing Correspondences between Models with the Epsilon Comparison Language. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 146–157. Springer, Heidelberg (2009)
51. Kolovos, D.S., Rose, L.M., Drivalos Matragkas, N., Paige, R.F., Polack, F.A.C., Fernandes, K.J.: Constructing and Navigating Non-invasive Model Decorations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 138–152. Springer, Heidelberg (2010)
52. Kolovos, D., Di Ruscio, D., Pierantonio, A., Paige, R.: Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. In: Proceedings of the International Workshop on Comparison and Versioning of Software Models @ ICSE 2009. IEEE (2009)
53. Kühne, T.: Matters of (Meta-) Modeling. Software and Systems Modeling 5, 369–385 (2006)
54. Küster, J.M., Gerth, C., Förster, A., Engels, G.: Detecting and Resolving Process Model Differences in the Absence of a Change Log. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 244–260. Springer, Heidelberg (2008)
55. Lambers, L., Ehrig, H., Orejas, F.: Conflict Detection for Graph Transformation with Negative Application Conditions. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 61–76. Springer, Heidelberg (2006)
56. Langer, P.: Adaptable Model Versioning based on Model Transformation by Demonstration. Ph.D. thesis, Vienna University of Technology (2011)
57. Langer, P., Wieland, K., Wimmer, M., Cabot, J.: From UML Profiles to EMF Profiles and Beyond. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 52–67. Springer, Heidelberg (2011)
58. Ledeczi, A., Maroti, M., Karsai, G., Nordstrom, G.: Metaprogrammable Toolkit for Model-Integrated Computing. In: Proceedings of the IEEE Conference and Workshop on Engineering of Computer-Based Systems, ECBS 1999, pp. 311–317 (March 1999)
59. Lin, Y., Gray, J., Jouault, F.: DSMDiff: A Differentiation Tool for Domain-specific Models. European Journal of Information Systems 16(4), 349–361 (2007)
60. Lippe, E., van Oosterom, N.: Operation-Based Merging. In: ACM SIGSOFT Symposium on Software Development Environment, pp. 78–87. ACM (1992)
61. Madiot, F., Dup, G.: EMF Facet Website (November 2010), http://www.eclipse.org/modeling/emft/facet/
62. Maoz, S., Ringert, J.O., Rumpe, B.: A Manifesto for Semantic Model Differencing. In: Dingel, J., Solberg, A. (eds.) MoDELS 2010 Workshops. LNCS, vol. 6627, pp. 194–203. Springer, Heidelberg (2011)
63. Maoz, S., Ringert, J., Rumpe, B.: ADDiff: Semantic Differencing for Activity Diagrams. In: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2011), pp. 179–189. ACM (2011)
64. Maoz, S., Ringert, J.O., Rumpe, B.: CDDiff: Semantic Differencing for Class Diagrams. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 230–254. Springer, Heidelberg (2011)

65. Mehra, A., Grundy, J., Hosking, J.: A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In: Proceedings of the International Conference on Automated Software Engineering (ASE 2005), pp. 204–213. ACM (2005)
66. Mens, T.: A State-of-the-Art Survey on Software Merging. IEEE Transactions on Software Engineering 28(5), 449–462 (2002)
67. Mens, T., Gorp, P.V.: A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science 152, 125–142 (2006)
68. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. Electronic Notes in Theoretical Computer Science 127(3), 113–128 (2005)
69. Munson, J.P., Dewan, P.: A Flexible Object Merging Framework. In: Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work, CSCW 1994, pp. 231–242. ACM (1994), `http://doi.acm.org/10.1145/192844.193016`
70. Murta, L., Corrêa, C., Prudêncio, J., Werner, C.: Towards Odyssey-VCS 2: Improvements Over a UML-based Version Control System. In: Proceedings of the International Workshop on Comparison and Versioning of Software Models @ MoDELS 2008, pp. 25–30. ACM (2008)
71. Nagl, M. (ed.): Building Tightly Integrated Software Development Environments: The IPSEN Approach. LNCS, vol. 1170. Springer, Heidelberg (1996)
72. Naur, P., Randell, B., Bauer, F.: Software Engineering: Report on a Conference Sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, October 7-11, 1968. Scientific Affairs Division, NATO (1969)
73. Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and Merging of Statecharts Specifications. In: Proceedings of the International Conference on Software Engineering (ICSE 2007), pp. 54–64. IEEE (2007)
74. Nuseibeh, B., Easterbrook, S.M., Russo, A.: Making Inconsistency Respectable in Software Development. Journal of Systems and Software 58(2), 171–180 (2001)
75. Object Management Group: Diagram Definition (DD), `http://www.omg.org/spec/DD/1.0/Beta2/` (accessed: February 21, 2012)
76. Object Management Group: Common Warehouse Metamodel (CWM) Specification V1.1. (March 2003), `http://www.omg.org/spec/CWM/1.1/`
77. Object Management Group: Meta-Object Facility 2.0 (MOF) (October 2004), `http://www.omg.org/cgi-bin/doc?ptc/03-10-04`
78. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/-Transformation Specification. Final Adopted Specification (November 2005), `http://www.omg.org/docs/ptc/07-07-07.pdf`
79. Object Management Group: Model-driven Architecture (MDA) (April 2005), `http://www.omg.org/mda/specs.html`
80. Object Management Group: UML Diagram Interchange, Version 1.0. (April 2006), `http://www.omg.org/spec/UMLDI/1.0/`
81. Object Management Group: XML Metadata Interchange 2.1.1 (XMI) (December 2007), `http://www.omg.org/spec/XMI/2.1.1`
82. Object Management Group: MOF Model to Text Transformation Language (MOFM2T) (January 2008), `http://www.omg.org/spec/MOFM2T/1.0/`
83. Object Management Group: Business Process Modeling Notation (BPMN), Version 1.2 (January 2009), `http://www.omg.org/spec/BPMN/1.2`
84. Object Management Group: Object Constraint Language (OCL), Version 2.2 (February 2010), `http://www.omg.org/spec/OCL/2.2`
85. Object Management Group: OMG Unified Modeling Language (OMG UML), Infrastructure V2.4.1 (August 2011), `http://www.omg.org/spec/UML/2.4.1/`

86. Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure V2.4.1 (July 2011), http://www.omg.org/spec/UML/2.4.1/
87. Oda, T., Saeki, M.: Generative Technique of Version Control Systems for Software Diagrams. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2005), pp. 515–524. IEEE (2005)
88. Ohst, D., Welle, M., Kelter, U.: Differences Between Versions of UML Diagrams. ACM SIGSOFT Software Engineering Notes 28(5), 227–236 (2003)
89. Oliveira, H., Murta, L., Werner, C.: Odyssey-VCS: A Flexible Version Control System for UML Model Elements. In: Proceedings of the International Workshop on Software Configuration Management, pp. 1–16. ACM (2005)
90. Opdyke, W.: Refactoring Object-oriented Frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign (1992)
91. Oracle: Java Metadata Interface (JMI) (June 2002), http://java.sun.com/products/jmi/
92. Parnas, D.: Software Engineering or Methods for the Multi-person Construction of Multi-version Programs. In: Hackl, C.E. (ed.) IBM 1974. LNCS, vol. 23, pp. 225–235. Springer, Heidelberg (1975)
93. Porres, I.: Rule-based Update Transformations and their Application to Model Refactorings. Software and System Modeling 4(4), 368–385 (2005)
94. Pressman, R., Ince, D.: Software Engineering: A Practitioner's Approach. McGraw-Hill, New York (1982)
95. Rahm, E., Bernstein, P.: A Survey of Approaches to Automatic Schema Matching. The VLDB Journal 10(4), 334–350 (2001)
96. Reiter, T., Altmanninger, K., Bergmayr, A., Schwinger, W., Kotsis, G.: Models in Conflict – Detection of Semantic Conflicts in Model-based Development. In: Proceedings of International Workshop on Model-Driven Enterprise Information Systems @ ICEIS 2007, pp. 29–40 (2007)
97. Rivera, J., Vallecillo, A.: Representing and Operating With Model Differences. In: Paige, R.F., Meyer, B. (eds.) TOOLS EUROPE 2008. LNBIP, vol. 11, pp. 141–160. Springer, Heidelberg (2008)
98. Schmidt, D.: Guest Editor's Introduction: Model-driven Engineering. Computer 39(2), 25–31 (2006)
99. Schmidt, M., Gloetzner, T.: Constructing Difference Tools for Models Using the SiDiff Framework. In: Companion of the International Conference on Software Engineering, pp. 947–948. ACM (2008)
100. Schneider, C., Zündorf, A.: Experiences in using Optimisitic Locking in Fujaba. Softwaretechnik Trends 27(2) (2007)
101. Schneider, C., Zündorf, A., Niere, J.: CoObRA – A Small Step for Development Tools to Collaborative Environments. In: Proceedings of the Workshop on Directions in Software Engineering Environments (2004)
102. Schwanke, R.W., Kaiser, G.E.: Living With Inconsistency in Large Systems. In: Proceedings of the International Workshop on Software Version and Configuration Control, pp. 98–118. Teubner B.G. GmbH (1988)
103. Selic, B.: The Pragmatics of Model-driven Development. IEEE Software 20(5), 19–25 (2003)
104. Sendall, S., Kozaczynski, W.: Model Transformation: The Heart and Soul of Model-Driven Software Development. IEEE Software 20, 42–45 (2003)
105. Shvaiko, P., Euzenat, J.: A Survey of Schema-Based Matching Approaches. In: Spaccapietra, S. (ed.) Journal on Data Semantics IV. LNCS, vol. 3730, pp. 146–171. Springer, Heidelberg (2005)

106. Sprinkle, J.: Model-integrated Computing. IEEE Potentials 23(1), 28–30 (2004)
107. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework 2.0. Addison-Wesley Professional (2008)
108. Sun, Y., White, J., Gray, J.: Model Transformation by Demonstration. In: Schürr, A., Selic, B. (eds.) MoDELS 2009. LNCS, vol. 5795, pp. 712–726. Springer, Heidelberg (2009)
109. Sunyé, G., Pollet, D., Le Traon, Y., Jézéquel, J.-M.: Refactoring UML Models. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 134–148. Springer, Heidelberg (2001)
110. Taentzer, G., Ermel, C., Langer, P., Wimmer, M.: Conflict Detection for Model Versioning Based on Graph Modifications. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 171–186. Springer, Heidelberg (2010)
111. Taentzer, G., Ermel, C., Langer, P., Wimmer, M.: A Fundamental Approach to Model Versioning Based on Graph Modifications. Accepted for Publication in Software and System Modeling (2012)
112. Thione, G., Perry, D.: Parallel Changes: Detecting Semantic Interferences. In: Proceedings of the International Computer Software and Applications Conference, pp. 47–56. IEEE (2005)
113. Vermolen, S., Wachsmuth, G., Visser, E.: Reconstructing Complex Metamodel Evolution. Tech. Rep. TUD-SERG-2011-026, Delft University of Technology (2011)
114. W3C: Extensible Markup Language (XML), Version 1.0 (2008), `http://www.w3.org/TR/REC-xml`
115. Wache, H., Voegele, T., Visser, U., Stuckenschmidt, H., Schuster, G., Neumann, H., Hübner, S.: Ontology-based Integration of Information — A Survey of Existing Approaches. In: Proceedings of the Workshop on Ontologies and Information Sharing (IJCAI 2001), pp. 108–117 (2001)
116. Westfechtel, B.: Structure-oriented Merging of Revisions of Software Documents. In: Proceedings of the International Workshop on Software Configuration Management, pp. 68–79. ACM (1991)
117. Westfechtel, B.: A Formal Approach to Three-way Merging of EMF Models. In: Proceedings of the International Workshop on Model Comparison in Practice @ TOOLS 2010, pp. 31–41. ACM (2010)
118. Wieland, K.: Conflict-tolerant Model Versioning. Ph.D. thesis, Vienna University of Technology (2011)
119. Xing, Z., Stroulia, E.: UMLDiff: An Algorithm for Object-oriented Design Differencing. In: Proceedings of the International Conference on Automated Software Engineering (ASE 2005), pp. 54–65. ACM (2005)
120. Xing, Z., Stroulia, E.: Refactoring Detection based on UMLDiff Change-Facts Queries. In: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006), pp. 263–274. IEEE (2006)