FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Conflict-tolerant Model Versioning

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Sozial- und Wirtschaftswissenschaften

eingereicht von

Dipl.-Ing. Konrad Wieland
Matrikelnummer 0326085

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel

Diese Dissertation haben begutachtet:

(o.Univ.-Prof. Dr. Gerti Kappel) (Univ.-Prof. Geraldine Fitzpatrick, PhD.)

Wien, 09.12.2011

(Dipl.-Ing. Konrad Wieland)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Conflict-tolerant Model Versioning

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Sozial- und Wirtschaftswissenschaften

by

Dipl.-Ing. Konrad Wieland
Registration Number 0326085

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel

The dissertation has been reviewed by:

(o.Univ.-Prof. Dr. Gerti Kappel) (Univ.-Prof. Geraldine Fitzpatrick, PhD.)

Wien, 09.12.2011

(Dipl.-Ing. Konrad Wieland)

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Konrad Wieland
Zwinzstr. 1b/13, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Danksagung

"Schöne Tage - nicht weinen, dass sie vergangen,
sondern lächeln, dass sie gewesen."
Rabindranath Tagore

An dieser Stelle möchte ich mich bei allen Menschen, die mich in den letzten Jahren begleitet und unterstützt haben, herzlichst bedanken.

Zu allererst möchte ich mich bei meiner Betreuerin Gerti Kappel bedanken, die mir die Chance gegeben hat, im Zuge des Projekts AMOR[1] eine Dissertation zu schreiben. Mit viel Geduld und wertvollen Ratschlägen stand mir Gerti Kappel immer zur Seite. Auch bei Geraldine Fitzpatrick, die mir in neuen Forschungsbereichen den Weg wies und wertvollen Input für diese Dissertation lieferte, möchte ich mich hier bedanken. Thank you!

Martina Seidl und Manuel Wimmer standen mir in den letzten Jahren als erste Ansprechpersonen jederzeit zur Verfügung, was nicht als Selbstverständlichkeit erachtet werden darf. Dankbar bin ich ihnen außerdem für die vielen motivierenden Worte und hilfreichen Ratschläge.

Spezieller Dank gilt Philip Langer, der mich die letzten 8 Jahre an der TU Wien begleitete und in allen Belangen unterstützte. Ohne ihn wäre ich nicht so weit gekommen und hätte diese Dissertation wahrscheinlich nie zu einem Ende gebracht. Danke für Alles!

Danken möchte ich auch dem gesamten AMOR Team, das den Grundstein dieser Dissertation legte: Petra Brosch, Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, Martina Seidl und Manuel Wimmer. An dieser Stelle möchte ich mich auch bei der Österreichischen Forschungsförderungsgesellschaft FFG bedanken, die das Projekt finanziert hat.

Großes Dank gilt auch den folgenden Personen, die maßgeblich dazu beigetragen haben, dass ich die letzten Jahre am Institut stets in positiver Erinnerung behalten werde: Robert Engel, Katja Hildebrandt, Christian Huemer, Philipp Liegl, Tanja Mayerhofer, Dieter Mayrhofer, Christian Pichler, Michael Pöttler und Marco Zapletal.

Diese Dissertation wäre ohne Unterstützung von guten Freunden und Familie nie möglich gewesen. Meine Eltern Elisabeth und Wolfgang und meine Schwester Roswitha gaben mir immer Rückhalt im Leben. Dafür möchte ich mich von ganzem Herzen bedanken.

In den letzten Monaten hat vor allem eine Person mit mir viel durchmachen müssen: meine Freundin Gini. Vielen Dank gilt ihrem Verständnis, ihrer Geduld und ihrer Unterstützung vor allem in schwierigeren Zeiten. Sie ist nicht nur eine wichtige Konstante in meinem Leben, sondern ein unersetzlicher Teil, der mir Ausgeglichenheit und Stärke gibt.

[1] Project AMOR (FIT-IT No. 819584)

to my grandfather

Abstract

Model-driven software engineering (MDSE), which has recently gained momentum in academia as well as in industry, changed the way in which modern software systems are built. In MDSE, the task of programming, i.e., writing code in a textual programming language, is replaced by modeling in a language such as the Unified Modeling Language (UML). The powerful abstraction mechanisms of models are not only used for documentation purposes, but also for compiling executable code directly out of models. With the rise of MDSE, several problems solved for traditional software engineering became urgent again because well established solutions are not directly transferable from code to models. Among others, the *collaborative development of models* is currently only limited supported by modeling tools and, consequently, it is mostly a one-(wo)man show. Especially in the field of *model versioning*, which supports the asynchronous modification of modeling artifacts by multiple developers, only first solutions start to emerge.

The urgent need for a suitable infrastructure supporting effective model versioning has been widely recognized by researchers as well as practitioners. Currently, however, there is a lack of empirical studies on the needs of software developers in practice concerning the collaborative development of software systems. The first contribution of this thesis tackles this problem and provides an extensive *survey about versioning in practice* by the means of an online questionnaire and qualitative expert interviews. One result of the empirical study shows that conflicts due to parallel modifications are considered harmful and, thus, developers try to avoid them. Conflicts, however, should not be seen as negative result of collaboration but as chance for discussing ideas and for improving the system under development. As consequence, the second contribution is a *conflict-tolerant model versioning* approach, where the developers may commit their changes in the central repository without worrying about possible conflicts. This approach merges two or more parallel versions by applying dedicated merge rules and, by this, it incorporates all modifications of the developers. This builds a good basis for discussing and resolving conflicts collaboratively. Finally, when resolving conflicts a high degree of user interaction is required. When setting models under version control with state-of-the art tools, however, conflicts are hardly accessible for the users. Also the empirical study has shown, that current version control systems lack for a dedicated *representation and visualization*. Moreover, user support is required to better understand the reasons behind the conflicting changes. The third contribution tackles these deficiencies by visualizing occurred conflicts in terms of *model annotations* and enriching them automatically with additional meta information to better understand the parallel evolution of the model under development. The implemented prototype is evaluated by means of a quasi-experimental study, which demonstrates the advantages of developing models in a collaborative manner.

Kurzfassung

Modell-getriebene Softwareentwicklung verändert die Art und Weise wie moderne Softwaresysteme entwickelt werden. Modelle werden nicht mehr ausschließlich zu Dokumentationszwecke verwendet, sondern auch zur Generierung ausführbaren Codes. Mit dem Aufkommen der Modell-getriebenen Softwareentwicklung werden einige Probleme, die bereits für die traditionelle Softwareentwicklung gelöst wurden, wieder aktuell, da sich diese Lösungen nicht direkt auf Modelle anwenden lassen. Unter Anderem wird die Entwicklung von Modellen im Team derzeit nur sehr eingeschränkt unterstützt. Speziell im Bereich der Modellversionierung, der die asynchrone Bearbeitung von Modellen durch mehrere EntwicklerInnen unterstützt, erschienen lediglich die ersten Lösungsansätze. Die Notwendigkeit effektiver Modellversionierungssysteme wurde sowohl von der Wissenschaft als auch von der Industrie erkannt. Es fehlen jedoch empirische Studien, die die Bedürfnisse der SoftwareentwicklerInnen bei der kollaborativen Entwicklung von Softwaresystemen aufzeigen. Daher wird in dieser Dissertation eine umfassende Studie präsentiert, die durch einen online Fragebogen und qualitative Experteninterviews diese Bedürfnisse aufzeigt. Ein Ergebnis dieser Studie ist, dass Konflikte, die durch parallele Änderungen von Modellen entstanden sind, als schädlich empfunden werden und dass daher von den EntwicklerInnen versucht wird, diese zur Gänze zur vermeiden. Jedoch sollten Konflikte nicht als negatives Resultat gemeinsamen Modellierens gesehen werden, sondern als Chance um Ideen zu diskutieren und um das zu entwickelnde System zu verbessern. Daher wird in dieser Dissertation ein Konflikt-toleranter Modellversionierungsansatz vorgestellt, bei dem die EntwicklerInnen ihre Änderungen in ein zentrales Modell-Repository einchecken können, ohne sich um Konflikte sorgen zu müssen. Dieser Ansatz fügt durch entsprechende Merge Regeln mehrere parallele Versionen eines Modells zu Einer zusammen und berücksichtigt die Änderungen aller EntwicklerInnen. Dieses gemergte Modell bildet eine gute Basis für Diskussionen über die verschiedenen Änderungen und dient dazu, aufgetretene Konflikte gemeinsam aufzulösen. Um über Konflikte entscheiden zu können, ist es notwendig, diese entsprechend zu visualisieren. Dies wurde auch durch die empirische Studie bestätigt. Jedoch bieten aktuelle Versionierungssysteme keine adäquate Repräsentation von Konflikten. Um Konflikte zwischen zwei Modellversionen auflösen zu können, ist es auch notwendig zu verstehen, wie der jeweilige Konflikt entstanden ist. Auch diese Anforderung kann von state-of-the-art Versionierungswerkzeugen nicht erfüllt werden. Aus diesen Gründen wird in dieser Dissertation eine Visualisierung von aufgetreten Konflikten auf Basis eines Modellannotationsmechansimus vorgestellt, mit dem das Modell auch mit zusätzlichen Metainformationen angereichert wird, um besser die parallele Evolution eines zu entwickelnden Modells zu verstehen. Dieser Mechanismus wurde in Form eines Prototyps implementiert, der mittels einer experimentellen Studie evaluiert wurde, die die Vorteile kollaborativen Modellierens veranschaulicht.

Contents

CHAPTER 1

Introduction

1.1 Background

Software engineering, as any other engineering discipline, must provide the ability and means to build systems which are so large and complex that they have to be built by teams or even by teams of teams of engineers [GJM02]. Mistrík et al. [MGHW10] point out the importance of collaboration support in software engineering as follows:

> "Collaboration among individuals [..] is central to modern software engineering. It takes many forms: joint activity to solve common problems, negotiation to resolve conflicts, creation of shared definitions, and both social and technical perspectives impacting all software development activity. [..] The grand challenge is not only to ensure that developers in a team deliver effectively as individuals, but that the whole team delivers more than just the sum of its parts."

To support collaborative development, Software Configuration Management (SCM) provides key tools and techniques for making the parallel development of software systems more manageable [Tic88]. Amongst others, SCM offers Version Control Systems (VCSs), which allow reusing single-user development environments for parallel development. Central repositories, to which developers can commit their changes and from which developers can update their local version to the latest version in the repository, support the management and administration of software artifacts under development. Of course, this holds true not only for traditional, code-centric software engineering, but also for *model-driven software engineering* (MDSE) [Sch06], which has recently gained momentum in academia as well as in industry, changing the way in which modern software systems are built. In MDSE, the task of programming, i.e., writing code in a textual programming language such as Java, is replaced by modeling in a graphical modeling language such as the Unified Modeling Language [Obj03]. The powerful abstraction mechanisms of models are not only used for documentation purposes, but also for compiling executable code directly out of models [Béz05]. MDSE, however, has huge implications on the versioning process and conflict management as described in the following.

1.2 Problem Statement

With the rise of MDSE, several problems solved for traditional software engineering became urgent again because well established solutions are not directly transferable from code to models. For example, to provide collaborative modeling support, text-based versioning systems such as Subversion[1] and CVS[2] have been reused, but quickly it has been realized that XML Metadata Interchange (XMI) serializations of models are not the appropriate representation for detecting and resolving conflicts between concurrently edited model versions. These incompatibilities might be explained by the graph-based structure of models, which must be taken into account by dedicated algorithms for matching, comparing, and merging models. While there has been considerable work to understand and support code versioning, the implications and issues when using model artifacts are less clear [ABK+09].

Dedicated VCSs for model versioning have been proposed which realize model specific comparison, conflict detection, conflict resolution, and merge components. However, after having surveyed the state of the art of model versioning systems and the literature stemming from this heterogeneous research area (cf. [BKL+11a]), the following drawbacks have been identified.

Deficiency 1: Lack of empirical studies about (model) versioning in practice. In the research field of model versioning, a plethora of research directions exists trying to meet the technical challenges of model versioning systems, mostly concerned with precise conflict detection and supportive conflict resolution [ASW09]. However, there is currently a lack of empirical studies trying to derive the "real" needs of software developers in practice concerning the collaborative development of software systems [Men02]. Such studies are highly needed, because several possibilities exist for how software can be developed collaboratively, both on the technical level as well as on the organizational level. The latter aspect is often ignored, especially in the model versioning research field. If we are able to understand real world experiences with model versioning then we would be better able to identify criteria which should determine the selection of versioning technologies as well as collaboration processes from an organizational viewpoint. Furthermore, lessons learned of current best practices in collaborative software development for the various development artifacts may be inferred from such studies. To the best of our knowledge, only few investigations have been carried out in order to find answers to these questions. For example, several issues arising from practice when merging different versions of a model are identified in Bendix et al. [BE09]. However, these findings are based on informal interviews within one company and do "not pretend to be general". Furthermore, the premise of Bendix et al. is that "model-centric development and its problems do not vary much from company to company" which has not been proven so far. In addition, models can be used in different ways, namely as sketches to discuss ideas and design alternatives, as blueprint for implementation, or for direct code generation [Fow03] and, thus, the collaborative development of models seems to vary from company to company.

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

Deficiency 2: Conflicts are considered harmful. In general, modeling activities always capture the personal view of the modeler on the system under development and the fact that each team member perceives the system under development differently is directly reflected by the models [Gru93]. Especially in early phases of the software development lifecycle, as long as the team has not established a consolidated view on the system, the understanding and the intention of the different team members might diverge.

One challenge within the modeling process is to create a common knowledge base. Tool support is needed allowing the creation of a common representation of subjective knowledge within the team in order to avoid misunderstandings. In this situation, awareness about conflicting modeling activities might be valuable in order to find a common solution by learning about the reasons that lead to the conflicts [RKdV08b].

The traditional tool support for collaborative software development, usually follows the paradigm of either avoiding conflicts or of resolving conflicts as soon as possible, because they are seen as "harmful". Conflict avoidance is for example realized by the means of *pessimistic version control systems* which lock artifacts for exclusive modification by exactly one developer. In contrast, *optimistic versioning approaches* support distributed, parallel team-work. This comes along with the price of conflict resolution when concurrently evolved versions of one model are merged. However, in such versioning systems the developer who is committing her changes is solely responsible to resolve the occurred conflicts immediately, which is an error-prone, time-consuming, and, thus, also an unpleasant task. The developer does not know the intentions of the others and a high probability exists that some modifications may be inevitably lost as they are removed in an undocumented manner. Such an approach is adequate for code when the specification of the system is already established and the code has to be executable at any point in time, e.g., by running test cases. In contrast to code, models are often used in an informal manner for collecting ideas and discussing design alternatives in brainstorming periods. As mentioned above, models may serve as sketches in early project phases [Fow03]. Models are then used to manage and improve communication among the team members by establishing common domain knowledge. Then, this loss of information is especially problematic.

To summarize, the challenge here is to introduce a new approach for model versioning, in which conflicts are not seen as negative result of collaboration but as a chance to discuss design alternatives or misinterpretations. The discussion of the conflicts might help to eliminate flaws in the design which are probably harder to eliminate at the later point in time of the software life cycle. The resulting merged model should incorporate all modifications of the participants to finally reflect a consolidated version.

Deficiency 3: Conflicts are difficult to access. When developing models in teams following the optimistic versioning paradigm conflicts may arise, which have to be resolved and user interaction is often required. A high degree of automation would enable an effective and time-saving development of models [MD94]. However, a plethora of conflicts exists, where automation is currently at its limit or is not appropriate at all [Men02]. Further difficulties in merging software models arise from the fact that they express aspects of a software system at a very high level of abstraction and, therefore, reveal a high amount of semantics, domain specific knowledge, and modeling experience.

When merging models advanced user support in terms of proper representation and visualization of the conflicts make the manual resolution practicable [BKL+11b]. It has been quickly realized that XMI serializations are not the appropriate representation for detecting and resolving conflicts between concurrently edited model versions, because developers are familiar with the concrete graphical syntax but not with computer internal representations. Thus, some dedicated approaches have been proposed for visualizing differences of models. They construct a dedicated view using the concrete syntax, which combines and highlights changes of both models using coloring techniques [MGH05, OWK03]. Hence, the modeler remains in her familiar modeling environment. However, these approaches require the implementation of specific editor extensions.

Furthermore, additional information to support change and conflict awareness is highly valuable when changing models asynchronously [TG04]. This information should incorporate which conflicting changes have been performed, who has performed them and when did the changes take place. Thus, making this information more explicit would support the developers when merging parallel versions of a model and would lead to better understanding of the reasons behind the conflicts. In addition, understanding how conflicts are resolved is very challenging in state-of-the-art versioning tools especially when two or more parallel versions exist.

To summarize, the challenge here is to find an approach which presents conflicts in a user-friendly way and which enriches the model with information about the performed changes and occurred conflicts. Furthermore, this approach should also extend the model with additional meta information to better understand the evolution of the models and to support collaborative negotiation and resolution of occurred conflicts when merging different versions of a model.

1.3 Contributions

The overall goal of this thesis is to provide a better collaboration support in the context of model versioning without adding restrictions regarding the employed modeling editor or modeling language. Before discussing the individual contributions of this thesis in detail, we present our optimistic versioning process including a conflict-tolerant merge approach from a developer's point of view. In contrast to optimistic versioning, pessimistic versioning does not allow for parallel modifications of the developers by locking the individual artifacts when modifying them. The optimistic versioning process is depicted in Figure 1.1 and provides the context for the contributions. Developer 1 and 2 may check out the same model in parallel indicated as `Version 1` from a common, central repository. Both perform changes on their local working copies `Version 1a` and `Version 1b`. Developer 1 finishes her work firstly and commits her changes back to the central repository. No conflicts occur, because no one else has committed changes in the meanwhile. When Developer 2 tries the same, the versioning system rejects his changes if they are conflicting with the changes of Developer 1. If conflicts occur, the highlighted merge process is passed through.

First of all, the changes applied by the developers on their local working copies have to be identified. In the next step, conflicts between concurrent changes are calculated and reported. The Conflict-tolerant Merge creates a merged version and annotates the reported conflicts. In the validation phase, the merged model is checked whether it is valid and well-formed regarding

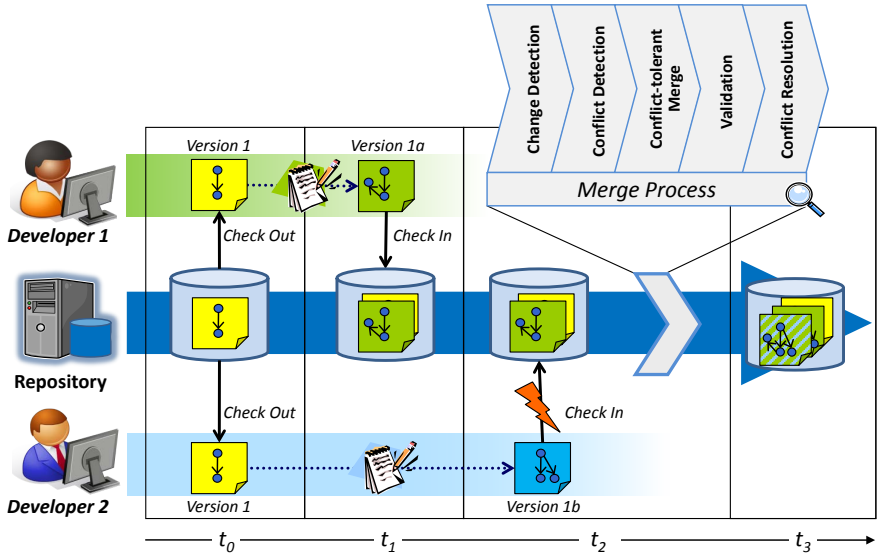

Figure 1.1: Versioning Process

the modeling language's rules. These inconsistencies and occurred conflicts have to be resolved to finally obtain a consolidated model in the central repository.

The general merge process has been conjointly elaborated in the model versioning project AMOR[3], which is presented in more detail in Chapter 3. Next to this thesis, the PhD theses of Philip Langer [Lan11] and Petra Brosch [Bro11] have been written within AMOR. Whereas those dissertations deal with conflict detection and specific aspects of conflict resolution, this thesis contributes the Conflict-tolerant Merge component using a newly developed model annotation mechanism. This component supports the developers in resolving conflicts collaboratively. In the following, the contributions of this thesis, which build solutions for the aforementioned deficiencies, are presented in more detail.

Contribution 1: Survey on Versioning in Practice. To tackle Deficiency 1, a comprehensive empirical study is provided in this thesis, including on the one hand the results of an online survey and on the other hand in-depth qualitative interviews. The overall goal of the empirical study is to gain insights of how versioning is currently used in practice. By conducting an online survey, we wanted to find out the state-of-the-art habits and processes for versioning software artifacts. We were interested in factors such as team-size or the geographical distribution of the team, because we conjectured that these have an impact on the motivation why versioning systems are used. On the basis of these results, we further conducted expert interviews with the aim to get insights how models are handled in their companies when developing them in a team. By following this comprehensive approach, we are able to reason more objectively about the influence of certain team characteristics on the collaborative software development. Furthermore, these studies gave as valuable insights on collaborative software development, which

[3] www.modelversioning.org

point out many interesting research issues. For instance, when a software system is developed in teams, only single-user environments are used in general, both for programming as well as for modeling. The collaboration features are out-sourced to additional versioning systems which coordinate the work of the various developers. Nevertheless, successful collaborative software development is more than just relying on current features of versioning systems. On the one hand the devision of labor and, on the other hand, the management of conflicts is of paramount importance. In this thesis, we describe our attempts to get a better understanding of requirements for model versioning in practice, based on which the following contributions of this thesis were elaborated.

Contribution 2: Conflict-tolerant Model Versioning System. If conflicting modifications occur in standard approaches, the developer, who committed her changes later than the other, is alone responsible for the conflict resolution and has to resolve the conflict immediately. As Deficiency 2 pointed out, when software models are typically employed for brainstorming, analysis, and design purposes, especially in early project phases, such an approach bears the danger of losing important viewpoints of different stakeholders and domain engineers, resulting in a lower quality of the overall system specification. Thus, in this thesis, we propose *conflict-tolerant model versioning* to overcome this problem. This new paradigm for optimistic versioning does not force the developers to resolve conflicts immediately. Our system supports deferring the resolution decision until a consolidated decision of the involved parties has been elaborated. Well-defined merge rules are used by our algorithm to incorporate all changes of the modelers, also when overlapping changes happen, and to mark conflicts by dedicated annotations which are introduced in Contribution 3.

To summarize, with our conflict-tolerant model versioning system, modelers may commit their changes without worrying about conflicts and their resolution. They check-in their changes and if conflicts arise they do not have to be resolved immediately. This approach is based on the assumption that conflicts are not considered as negative results of collaboration, but as chance for improvements.

Contribution 3: Collaboration Support through Model Annotations. When merging models in a conflict-tolerant way as described above, occurred conflicts are annotated. Special focus is set on *change and conflict awareness* to inform the user which changes resulted in a conflict, where and when have these changes been performed and who was responsible for these changes with the goal to better understand the reason behind the conflicts. Thus, the merged model is enriched with meta-information annotations as specified by the merged rules. These annotations provide information about the concurrent changes, the involved users, and time-related metadata.

Additionally, the resolution process in the so-called consolidation phase is also supported by this annotation mechanism to allow the resolution of the occurred conflicts collaboratively and in an asynchronous manner. The resolution process itself is supported by a *conflict resolution model*, with which the participants can be assigned to specific conflicts to propose resolutions and find collaboratively a consolidated version. By this method, we transform conflicts into collaborations which results in a higher acceptance of the consolidated model. In addition, the

versioning system should allow for a better comprehension of the evolution of a model also in cases where conflicts arise. Since we have introduced a lifecycle to the conflicts and since we store the information how a conflict is resolved, modelers can be fully aware of what happened in the development process of a model.

The developed model annotation mechanism is following the principle of our model versioning system, which avoids restrictions regarding the employed modeling editor or modeling language. Therefore, we ported the *lightweight extension mechanism* known from UML Profiles to the realm of Eclipse Modeling Framework (EMF) models. By this, every EMF-based model may be annotated with stereotypes containing tagged values. Stereotype applications may be visualized on top of the graphical representation of a model to support the user in resolving all annotated conflicts directly in the model. Annotations are saved in a separate model to avoid polluting the merged model. However, when committing the merged model comprising conflicting changes, the annotation model is saved alongside the merged model to also allow other modelers to investigate and resolve existing conflicts.

1.4 Methodology

The methodological approach used in this thesis conforms to the design science approach presented by Hevner et al. [HMPR04]. In general, the types of output produced by design research are representational constructs, models, methods, or instantiations [MS95]. The principle layout of design science research is depicted in Figure 1.2.

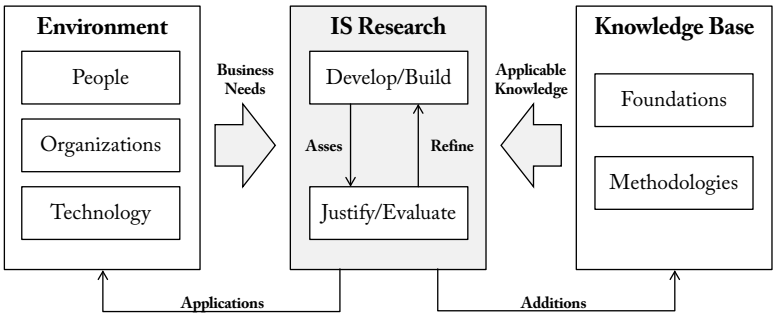

Figure 1.2: Information Systems Research Framework based on [HMPR04]

The development of an artifact and its refinement based on different evaluation methods constitutes the core of this research process. The design is driven by the environment defining business needs and addressing these needs assures research relevance. Existing foundations and methodologies are applied in the design of the artifact and, after evaluating this artifact, the knowledge base is updated or new knowledge is added. In the following, we outline how we use this approach within this thesis according to the design-science research guidelines [HMPR04].

- **Design as an Artifact:** The contributions of this thesis produce a viable artifact in terms of a new paradigm for versioning software models. The conflict-tolerant merge component

as well as the generic model annotation mechanism called EMF Profiles were developed to support this paradigm.

- **Problem Relevance:** These artifacts have been developed to solve important and relevant business problems. An online survey and expert interviews were conducted pointing out the needs for supporting a team of developers in merging parallel versions of a model.

- **Design Evaluation:** During the development phase, tests were conducted for refining and improving the individual artifacts. In addition, the utility of the artifacts are rigorously demonstrated via a well-executed, quasi-experimental study.

- **Research Contributions:** The contributions of the thesis are the artifacts themselves. The major parts have been reviewed as well as assessed by several researchers in terms of peer reviews inside the modeling and CSCW communities. The reviewers confirm that the contributions presented in this thesis solve important, previously unsolved problems.

- **Research Rigor:** In the construction and evaluation phase rigorous and well-defined methods are applied by following the design science research guidelines. The artifacts are designed to meet the business needs identified by an online survey and expert interviews. Foundations of related disciplines such as collaborative modeling, software configuration management, or tolerating inconsistencies in software engineering, on which the development of the presented contributions are based, were evaluated. In addition, well-known methodologies providing guidelines for evaluating the artifacts were considered (e.g., [CSG63], [RH09], [Yin02]).

- **Design as a Search Process:** The design of the artifacts was based on a search process in which different design alternatives were considered. Some of them are also discussed in this thesis.

- **Communication of Research:** This thesis as well as previous work on which the contributions are based, are published within dedicated research communities to ensure repeatability. This builds a new knowledge base for further research.

1.5 Thesis Outline

Parts of this thesis have been published in peer-reviewed books, journals, conferences, and workshops. In the following, the structure of the thesis is outlined and related publications are mentioned accordingly.

Chapter 2: Related Work. In this chapter, we present approaches and tools supporting the collaborative development of models. Furthermore, we introduce fundamental concepts of versioning software artifacts and discuss state-of-the-art model versioning systems based on previous work [BKL+11a].

Chapter 3: A Tour of AMOR. This thesis has been elaborated in the context of the research project "AMOR". Therefore, we provide in this chapter an overview of the concepts and findings that have been conjointly elaborated among all project team members. In particular, we introduce the AMOR merge process, its principles, and a categorization of merge conflicts. This chapter also contains content published in [BKS+10] and in [BKL+11a].

Chapter 4: Survey on Versioning in Practice. In this chapter, we present the results of our online survey and expert interviews, with which we investigated versioning habits and processes in practice and underline the relevance of the contributions of this thesis. In addition, lessons learned of this survey are discussed. The questionnaire underlining the online survey is depicted in Appendix A. The survey is also published in [WFK+11].

Chapter 5: Turning Collaborations into Annotations. Based on previous work [BLS+10b, WLS+11], this chapter presents a new paradigm for optimistic versioning. Conflicts are tolerated to resolve them later on in a collaborative setting leading to one consolidated version of the parallel modified model. Dedicated merge rules as well as a conflict resolution model is provided.

Chapter 6: Making AMOR Collaboration-Aware. In this chapter, the extension of AMOR's core system is presented. The most important components to realize a collaboration-aware model versioning system are presented. One key component, the light-weight model annotation mechanism called "EMF Profiles", is presented in more detail and is also published in [LWWC11].

Chapter 7: Evaluation. This chapter provides the evaluation of the contributed artifacts. The evaluation has been performed as a quasi-experimental study. The elaboration as well as the results of the study are presented and discussed.

Chapter 8: Conclusion. Finally, the contributions of this thesis are summarized and their limitations are critically discussed. In addition, an outlook on future work concludes this thesis.

CHAPTER 2

Related Work

Concerning the overall goal of this thesis, in this chapter we survey scientific approaches related to the context and contributions of this thesis. Thus, two categories of related work have been identified: (i) collaborative modeling and (ii) versioning.

First of all, we present dedicated approaches for supporting collaborative software development and modeling in general in Section 2.1. In addition, when developing models collaboratively, conflicts may arise and, thus, different strategies emerged how to handle them. We present representative approaches for each strategy.

Furthermore, when developing software, version control systems are often the first choice to support collaboration. As this thesis is concerned with versioning software models, we present in Section 2.2 techniques and systems in the area of software versioning and survey state-of-the-art model versioning systems.

2.1 Collaborative Modeling

Software engineering, as any other engineering discipline, must provide the ability and means to build systems which are so large and complex that they have to be built by teams or even by teams of teams of engineers [GJM02]. In the last decades, a plethora of tools and methods have been developed to support collaboration. In 1988, Johansen firstly introduced the a conceptualization of Computer Supported Collaborative Work (CSCW) systems by considering the context of the system's use leading to the so-called "CSCW Matrix" depicted in Figure 2.1 [Joh88]; it is also published in [Bae95].

In this matrix, two dimensions are distinguished: On the one hand a distinction is drawn whether a group of people is working at the same place (collocated) or not (remote). And on the other hand, whether group members may work at the same time (synchronously) or not (asynchronously).

Face-to-face interactions take place if the members of a team are working in the same room at the same time. Examples are decision rooms, single display groupware, etc. If the team members collaborate also at the same time, but are situated not in the same place, a so-called *remote*

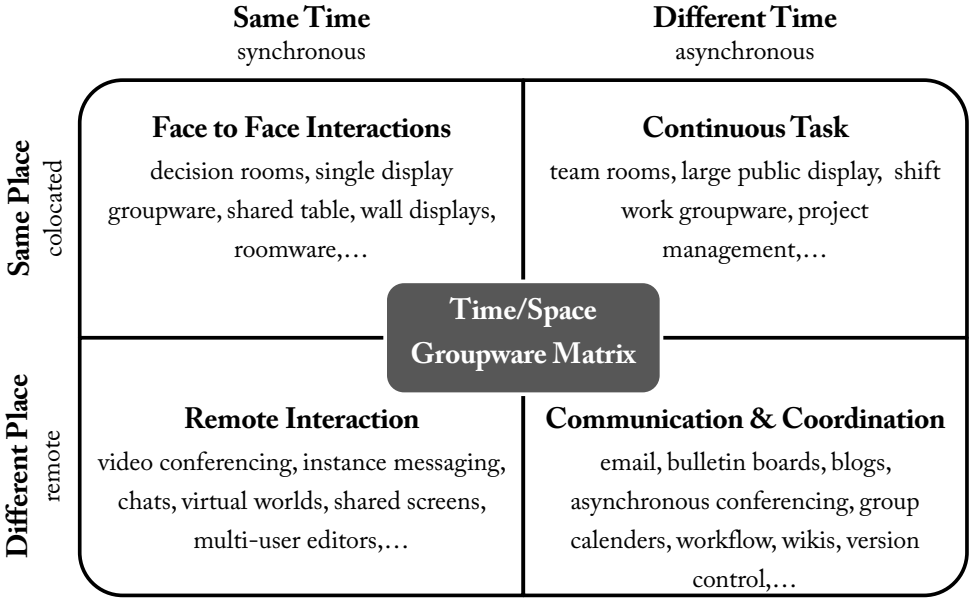

Figure 2.1: CSCW Matrix [Joh88]

interaction takes place. This situation occurs when using a video conferencing system, virtual worlds or shared screens, etc. Baecker [Bae95] defines asynchronous groupware as support for "communication and problem solving among groups of individuals who contribute at different times, and typically also are geographically dispersed." If the team members are indeed distributed the term *communication & collaboration* is used. This is applicable when using, for example, wikis or version control systems for supporting collaborative development. If working asynchronously but the team is collocated at the same place a so-called *continuous task* is performed. Examples for that are team rooms or typical project management tasks.

This thesis focuses on collaboratively developing software models, thus, we review in the following state-of-the-art tools and techniques which support different team-based modeling tasks. In general, models can be used in different ways, namely as sketch to discuss ideas and design alternatives, as blueprint for implementation, or to generate code directly out of them [Fow03]. Thus, various tools and techniques exist to support the collaborative development of models. In in the following, we only want to point out some of them.

To support face-to-face modeling sessions, Renger et al. [RKdV08a] propose an interactive whiteboard for synchronous modeling. Such approaches perfectly fit for conceptual designing phases to create a shared understanding between different stakeholders about a system representation. Also in the area of requirements engineering groupware for supporting face-to-face sessions have been developed [NE00].

If the team is not located in the same place, support for remote collaboration is necessary. Shared screens can be used for synchronously editing models. However, with shared screens only one modeler may perform changes whereas the others act as observers. Thus, for example, Fluegge [Flu09] and Thum et al. [TSS09]) propose collaborative editors for modeling with which

remote interactions are supported.

For asynchronous and remote software development versioning systems are indispensable for supporting coordination and communication. Since these line-based and text-based systems are not applicable for models, model versioning systems are required. In Section 2.2 different versioning techniques are presented and state-of-the-art model versioning systems are discussed. However, the price for developing models collaboratively, especially in an asynchronous way, is that conflicts may arise due to parallel changes of the same model elements. Thus, the awareness of conflicts and, furthermore, dealing with them are of significant importance. Over the years, three different ways to handle conflicts in collaborative software development have solidified [Edw97, Men02]: (i) avoiding conflicts, (ii) resolving conflicts, and (iii) tolerating conflicts. Different approaches following one of these strategies are outlined in the following:

2.1.1 Conflict Avoidance

Probably the simplest way to deal with conflicts is to establish mechanisms which make the occurrence of conflicts impossible. Conflict avoidance is realized either by *real-time collaborative editing* or by *pessimistic versioning*. Conflicts can be minimized in the former, because the developers work synchronously together on the same artifact and each developer is aware of the modifications others have performed. Conflicts cannot occur in the later, because exclusive rights to modify an artifact are granted.

Already in 1968, Engelbart and English showcased an approach to collaborative editing in the conferencing system presented at his "mother of all demonstrations" [EE68]. Over the years, several dedicated environments for real-time collaboration have been proposed (for textual artifacts as well as for models) which provide sophisticated notification and communication mechanisms indicating that a resource is currently touched by an other team member (e.g., [She03, FMP06]). Examples of collaborative modeling editors are SLIM [TSS09], DAWN [Flu09], and SpacEclipse [GMB+11].

Pessimistic versioning, also known as locking, is supported by most standard versioning systems such as Subversion (SVN), as well as by several dedicated model versioning systems, allowing only the developer to modify a certain artifact if she possesses the lock on this artifact. Code versioning (pessimistic as well as optimistic which we discuss below) has a long history in computer science—the first systems date back to the 70s—and have become indispensable for efficient software development [CW98, Men02]. As discussed before, a particular challenge in model versioning is posed by the selection of an adequate level of granularity determining the size of a lock since this has significant impacts on the work efficiency. The price of conflict avoidance, resulting in consolidated states of the system only, has to be paid with restricted flexibility during work, because the developers have to coordinate meetings for the real-time editing approach or they have to spend time idle when waiting for resources to be unlocked again.

2.1.2 Conflict Resolution

When applying optimistic versioning, the developers intentionally risk conflicts for more flexibility during work, because optimistic versioning tools put no dependency constraints on the

developers. From time to time, when they save their work in the central repository, the different versions of the artifact under development have to be merged. If a conflict is detected, it has to be resolved immediately; otherwise it may not be committed in the central repository. Thus, the developers may assume that the most recent version stored in the central repository never contains any inconsistencies, i.e., that none were introduced when merging the concurrently evolved versions. The resolution step may be supported with the help of specific rules or policies [MD94]. Although very simple, line-wise comparison has proven to be effective for textual artifacts like source code. For software models, more sophisticated comparison and conflict detection algorithms are required. This is because they have to take into account the graph-based structure and their rich semantics which would be lost if models are serialized in flat text files [ABK+09]. Therefore, recently several dedicated model versioning approaches emerged which are presented in Section 2.2. Despite the precise conflict detection components, merge problems might still occur, because conflicts are usually resolved by the person who does the later check-in and this person might be (on purpose or by accident) undo the modifications performed by other team members. For this problem, we offered a solution in [BSW+09], with which we suggested to perform conflict resolution in the team instead of by an individual in order to avoid misunderstandings. In many situations, conflicts indicate misunderstandings in the specification, which might be especially valuable in early phases of the development process. Hence, the idea is to collect and to tolerate conflicts and resolve them at a later point in time that is discussed in the following.

2.1.3 Conflict Tolerance

During the late 80s to the late 90s, several works have been published which aim at managing inconsistencies. Inconsistencies may arise, when syntactic properties (context-free or context-sensitive) or semantical concerns (structural or behavioral) are violated. Conflicts may be seen as a certain kind of inconsistencies in the software engineering process. One of the most interesting commonalities of these works is that the authors considered inconsistencies not only as negative result of collaborative development, but also see them as necessary means for identifying aspects of systems which need further analysis or which need to reflect different viewpoints of different stakeholders [NER01, SZ01]. Originally, the need for inconsistency-aware software engineering emerged in the field of programming languages, especially when very large systems are developed by a team. Schwanke and Kaiser [SK88] have been one of the first who proposed to live with inconsistencies by using a specially adapted programming environment for identifying, tracking, and tolerating inconsistencies to a certain extent. A similar idea was followed by Balzer [Bal89] for tolerating inconsistencies by relaxing consistency constraints. Instead of forcing the developer to resolve the inconsistencies immediately as they appear, he proposed to annotate them with so called “pollution markers”. Those markers comprise also meta-information for the resolution such as who is likely capable to resolve the inconsistency and for marking code segments which are influenced by the detected inconsistencies. Furthermore, Finkelstein et al. [FGH+94] presented the “ViewPoints” framework for multi-perspective development allowing inconsistencies between different perspectives and their management by employing a logic-based approach which allows powerful reasoning even in cases where incon-

sistencies occur [HN98]. In [BLS+10b] we proposed a first idea to tolerate conflicts in the context of model versioning.

The main contribution of this thesis is in line with the mentioned approaches for tolerating inconsistencies in software engineering. In particular, we also aim at detecting, marking, and managing inconsistencies. Concerning the marking of conflicts, we also have a kind of pollution markers as introduced by [Bal89]. However, we are strongly focusing on the parallel development of models, thus we have additional kinds of conflicts as discussed in Section 3.2. Furthermore, we are not only marking, but we have also to merge a tailored version, in which it is possible to mark the conflicts and inconsistencies. Our goal is to support this approach without adapting the implementation of the modeling environment and versioning system. This is mainly supported by the powerful dynamic extension mechanism of EMF Profiles (cf. Section 6.3). A dedicated conflict profile is used to annotate and visualize conflicts within the model without polluting it. Thus, in the following related approaches for annotating models and for visualizing conflicts are discussed.

As already mentioned, in this thesis, we are following the paradigm of developing models independently of time and place. We want to support modeling in a team by developing a versioning system dedicated to software models. Thus, to lay out the foundations of versioning, we collect and unify important concepts, terminologies, and design possibilities regarding artifact and change representation from past achievements. To get a better picture of the present situation, we then survey state-of-the-art model versioning systems.

2.2 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 [ELH+05], 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 changes and form the fundamental building block for the entire field of Source Configuration Management (SCM). Second, versioning systems aim at managing the evolution of software artifacts performed by a distributed team of developers.

In that long history of active research on software versioning, diverse formalisms and technologies emerged. To categorize this variety of different approaches, Conradi and Westfechtel [CW98] 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 artifact's structure and copes with the dimension of 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

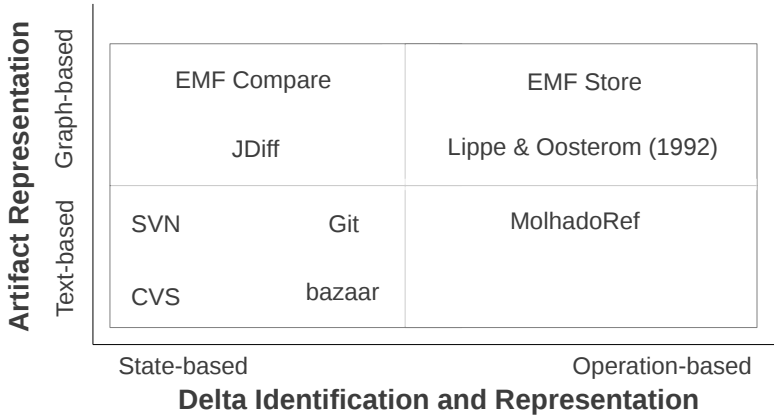

Figure 2.2: Categorization of Versioning Systems

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 thesis, 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. Figure 1.1). In this section, we first outline the fundamental design dimensions of versioning systems. Subsequently, we present some representatives of versioning systems using different designs. Finally, we elaborate on the consequences of different design possibilities considering the quality of the merged version based on an example.

2.2.1 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 (cf. Figure 2.2). 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. Most basically, 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 considers how deltas are *identified*, *represented*, and *merged* in order to create a consolidated version. 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. Existing merge approaches either operate on the *states*, i.e., versions, of an artifact, or on identified operations which have been applied between a common origin model (cf. Version 0 in Figure 1.1) and the two successors (cf. Version 1 and 2 in Figure 1.1).

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. A profound discussion on more complex types of conflicts is given in Chapter 3. For more information on software merging in general, the interested reader is referred to [Men02].

Text-based merge approaches operate solely on the textual representation of a software artifact in terms of flat text files. Within a text file, the atomic unit 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 the graph-based representation of a software artifact for more precise conflict detection and merging. Such approaches de-serialize or translate the versioned software artifact into a specific structure before merging. Mens [Men02] 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. 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. [Men02]) and their increased computational complexity. Furthermore, it is not always trivial to point the developer to the modifications that caused the conflict. If such a trace back to the causing modifications is missing or inaccurate, it might be difficult for developers to understand and resolve the raised conflicts, because they are reported based on a different representation, i.e., the graph, of the artifact, and not in the textual representation the developer is familiar with.

The second dimension in Figure 2.2 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 them and merge all differences which are not contradicting with each other. Such approaches may either be applied to two states (Version 1 and Version 2 in Figure 1.1), called two-way merging, or to three states (including their common ancestor Version 0

in Figure 1.1), 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. [KN06] for an overview of matching techniques). Model matching, or more generally the graph isomorphism problem is NP-hard (cf. [KR96]) and therefore very expensive regarding its run time. 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 with the algorithm shown in Algorithm 2.1. Please note that this algorithm only serves to conceptually clarify basic state-based merging. This algorithm is applicable for both, text-based and graph-based merging, whereas n_X denotes the atomic element n within the product space of Version X; that is, n_o for an element in the common origin version and n_1 or n_2 for an element in the two revised versions, respectively.

Algorithm 2.1 iterates through each element n_o in the common origin version V_o of a software artifact. The following two lines retrieve the elements matching with n_o from the two modified versions V_{r1} and V_{r2}. However, there might be no match for n_o in V_{r1} or V_{r2} since it might have been removed. If n_o has a match in both versions V_{r1} and V_{r2}, the algorithm checks if it has been modified in the versions V_{r1} and V_{r2}. If the matching element is different from the original element n_o, i.e., it has been modified, 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 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 unit n_o is used for the merged version. Next, the algorithm checks if there is no match for n_o in one of the two modified versions, i.e., it has been removed. If so, the algorithm determines whether it has been concurrently modified and raises, in this case, a Delete/Update conflict. If the element has not been modified, it is removed from the merged version. The element n_o is also removed, if there is no match 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}, which have no match in the original version V_o and which, consequently, are 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 changes, also composite operations such as refactorings (e.g., [KHWH10]). The knowledge on applied refactorings may significantly increase the quality of the merge as stated by Dig et al. [DMJN08]. The downside of operation recording is the strong dependency on the used 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 effi-

input : Common origin model V_o, two revised models V_{r1} and V_{r2}
output: The merged model version V_m

```
1  foreach n_o ∈ V_o do
2      n_1 ← match(n_o in V_r1);
3      n_2 ← match(n_o in V_r2);
4      if hasMatch(n_o in V_r1) ∧ hasMatch(n_o in V_r2) then
5          if diff(n_o, n_1) ∧ ¬ diff(n_o, n_2) then
6              Use n_1 in V_m
7          end
8          if ¬ diff(n_o, n_1) ∧ diff(n_o, n_2) then
9              Use n_2 in V_m
10         end
11         if diff(n_o, n_1) ∧ diff(n_o, n_2) then
12             Raise Update/Update conflict
13         end
14         if ¬ diff(n_o, n_1) ∧ ¬ diff(n_o, n_2) then
15             Use n_o in V_m
16         end
17     end
18     if hasMatch(n_o in V_r1) ∧ ¬ hasMatch(n_o in V_r2) then
19         if diff(n_o, n_1) then
20             Raise Delete/Update conflict
21             else Remove n_o in V_m
22         end
23     end
24     if ¬ hasMatch(n_o in V_r1) ∧ hasMatch(n_o in V_r2) then
25         if diff(n_o, n_2) then
26             Raise Delete/Update conflict
27             else Remove n_o in V_m
28         end
29     end
30     if ¬ hasMatch(n_o in V_r1) ∧ ¬ hasMatch(n_o in V_r2) then
31         Remove n_o in V_m
32     end
33 end
34 foreach n_1 ∈ V_r1 do
35     if ¬ hasMatch(n_1 in V_o) then Add n_1 to V_m
36 end
37 foreach n_2 ∈ V_r2 do
38     if ¬ hasMatch(n_2 in V_o) then Add n_2 to V_m
39 end
```

Algorithm 2.1: State-based Merge Algorithm

cient 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 (Version 0 to Version 1 and Version 0 to Version 2) for commutativity to reveal conflicts (cf. [LvO92]). 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 changes within the cross product of all atomic changes 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 changes are not commutative. After checking for commutativity, operation-based merge approaches apply all non-conflicting (commutative) changes 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 enable to gather more information (e.g., change order and refactorings), than state-based differencing. In particular, state-based approaches do 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. Furthermore, one part of the computational complexity which was saved in contrast to state-based matching and differencing is lost again due to operation sequence cleansing and non-trivial checking for commutativity. 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 [KHWH10].

Anyhow, the border between state-based and operation-based merging is sometimes blurry. Indeed, we can clearly distinguish whether the changes are recorded or differences are derived from the states, however, some *state-based approaches* derive the *applied operations* from the states and use operation-based conflict detection techniques. 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 these 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.

Selected Representatives of Versioning Systems

In Figure 2.2, we cited some representatives for each combination of the two dimensions in the domain of source code versioning as well as in the domain of model versioning. In the following, we briefly introduce and compare the representatives listed in Figure 2.2. For a more detailed description of existing model versioning approaches we kindly refer to Section 2.2.2.

The combination of *text-based and state-based merge approaches* are probably the most adopted ones in practice. For instance, traditional central Version Control Systems such as CVS[1] and SVN[2] use state-based three-way merging of flat text files. The smallest indivisible unit of merging in these systems is usually a *line* within a text file, as it is the case for the Unix *diff* utility [HM76]. Lines are matched across different versions by searching for the Least Common Sub-sequence (LCS). For efficiency, usually only *completely equal* lines are matched and, therefore, no dedicated difference function for deriving the actual difference between two lines is required: A line is simply either matched and therefore equal, or unmatched and therefore considered to be added or removed at a certain position in a text file. Consequently, parallel modifications to *different* lines can be merged without user intervention as long as they are at different positions. As soon as the same line is modified in both versions (Version 1 and Version 2) or modified and concurrently deleted, a conflict is annotated in the merged file. As stated earlier, due to their syntax and semantics unawareness, compile-time and run-time errors might be introduced by the merge. The same applies to the distributed version control systems (DVCS) git[3] and bazaar[4], since they are also state-based and line-based. The major difference to SVN and CVS is their distributed nature. DVCS disclaim a single central repository and take a peer-to-peer approach instead. Developers commit their changes to a local repository, i.e., a peer, and push them to other remote peers as they wish. Besides several other organizational advantages, this enables a higher commit frequency since a commit does not immediately affect other developers. Changes might therefore be grouped into *atomic commits* and pushed to other peers more easily which is a step towards operation-based merging.

MolhadoRef [DMJN08], a representative for *text- and operation-based approaches*, aims at improving the merge result by also considering refactorings applied to object-oriented (Java) programs. Applications of refactorings are recorded in the development environment. When two versions are merged, all recorded refactorings are undone in both modified versions, then the versions, excluding the refactoring applications, are merged in a traditional text-based manner, and, finally, all refactorings are re-applied to this merged version. This significantly improves the merge result and avoids unnecessary conflicts in many scenarios. However, as already mentioned, a strong dependency to the applied editor is given because the editor has to provide operation logs. Furthermore, handling refactorings requires language-specific knowledge encoded in the merge component.

Several *state-based approaches* exist which operate on a *graph-based representation* of the versioned software artifact. In Figure 2.2, we cite two representatives for graph-based and state-based approaches—one for source code, namely *JDiff* [AOH07], and one for software models, namely *EMF Compare*[5] [BP08]. JDiff is a graph-based differencing approach for Java source code. Corresponding classes, interfaces and methods are matched by their qualified name or signature. This matching also accounts for the possibility to interact with the user in order to improve the match of renamed but still corresponding elements due to the absence of unique

[1] `http://www.cvshome.org`
[2] `http://subversion.tigris.org`
[3] `http://git-scm.com`
[4] `http://bazaar.canonical.com`
[5] `http://www.eclipse.org/emft/projects/compare`

identifiers. For matching and differencing the method bodies, the approach builds enhanced control-flow graphs representing the statements in the bodies and compares them. By this, JDiff can provide information that accurately reflects the effects of code changes on the program at the statement level. EMF Compare is a model comparison framework for EMF based models. It facilitates heuristics for matching model elements and can detect differences between matched elements on a fine-grained level (metamodel features of each model element). The matching and differencing is applied on the generic model-based representation of the elements.

There are several purely *operation-based approaches* which record changes directly and apply merging on a *graph-based representation*. The first publication which introduced operation-based merging was elaborated by Lippe and Oosterom [LvO92]. They proposed to record all changes applied to an object-oriented database system. After the precise change-sets are available due to recording, they are merged by re-applying all their changes to the common ancestor version. In general, a pair of changes is conflicting if they are not commutative. *EMF Store* [KHWH10] is an operation- and graph-based versioning system for software models. Since EMF Compare and EMF Store are representatives of model versioning systems, they are further elaborated on in Section 2.2.2.

Consequences of Design Decisions

To highlight the benefits and drawbacks of the four possible combinations of the versioning approaches based on Figure 2.2, we present a small versioning example depicted in Figure 2.3 and conceptually apply each approach for analyzing its quality in terms of the detected conflicts and derived merged version.

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 Figure 2.3 and defined by the EBNF-like Xtext[6] 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 Figure 2.3. In the initial version (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 changes in Version 1 and Version 2 are highlighted with bold fonts or edges in Figure 2.3. 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. Figure 2.4(a) for the result) where the artifact's representation is a single line and the match function only matches completely equal lines (as with SVN, CVS, Git, bazaar, etc), the first line is correctly merged since it has only been modified in Version 1 and remained untouched in Version 2 (cf. Algorithm 2.1). 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 represented by line 5 in

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

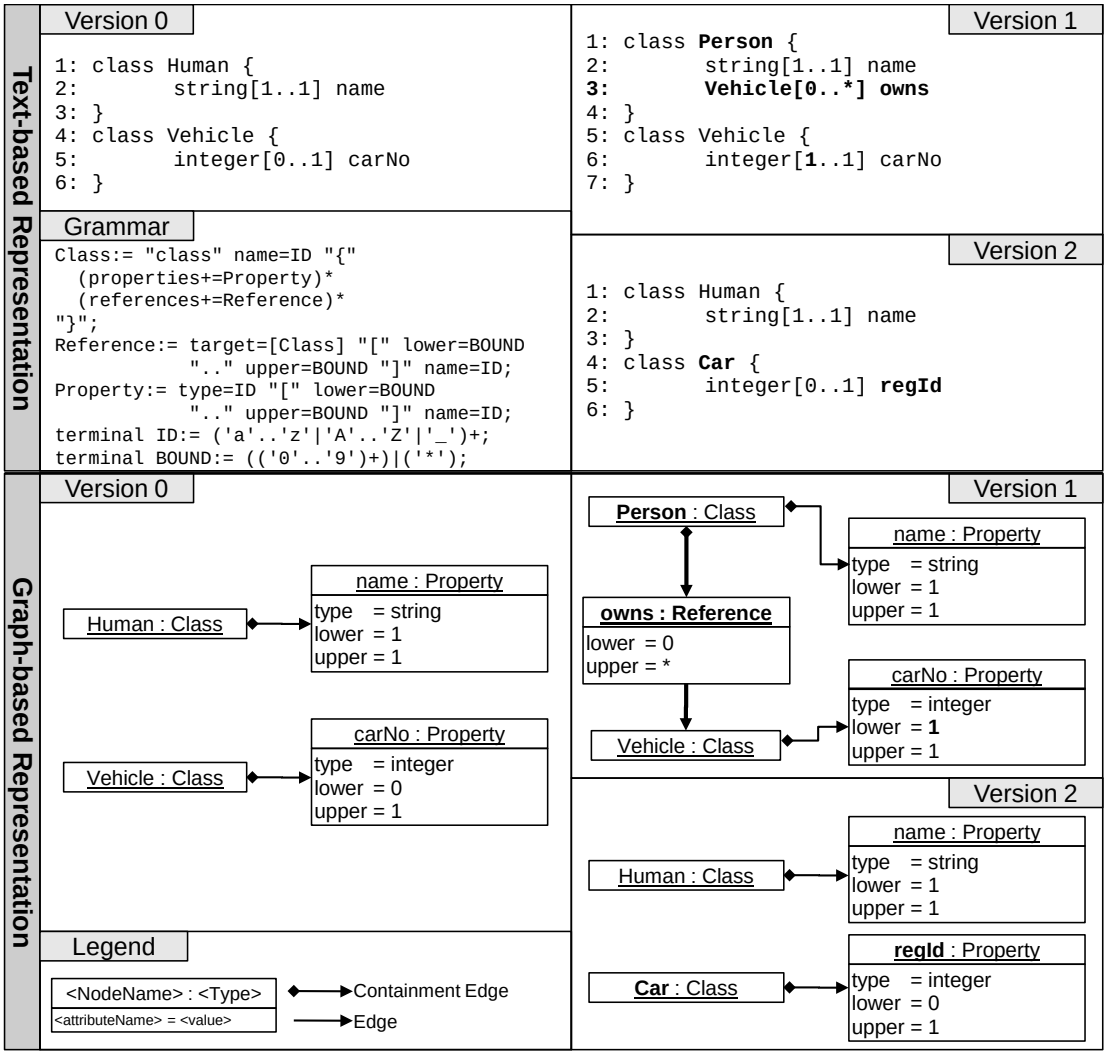

Figure 2.3: Versioning Example

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

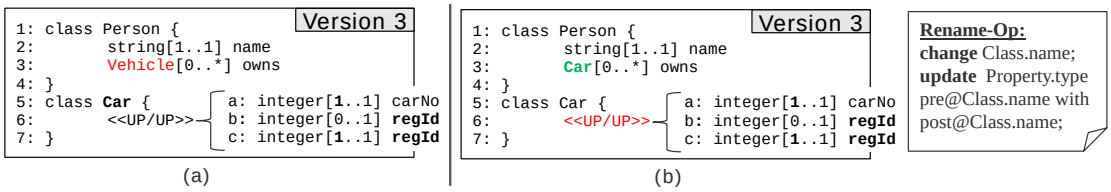

Figure 2.4: Text-based Versioning Example: (a) state, (b) operation

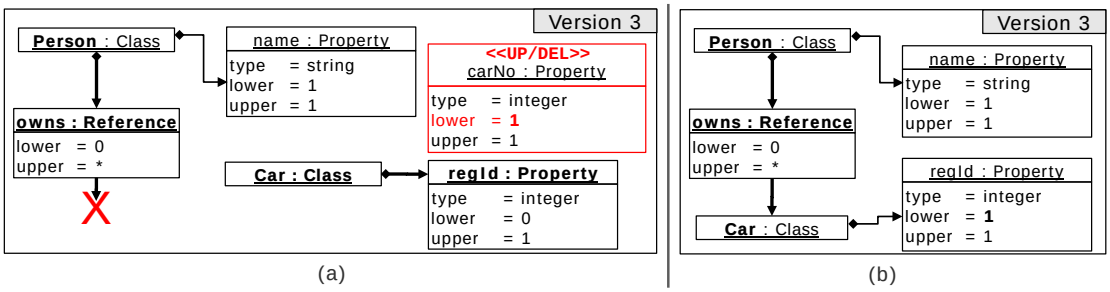

Figure 2.5: Graph-based Versioning Example: (a) state, (b) operation

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 the MolhadoRef software configuration management system) solve such an issue by incorporating knowledge on applied refactorings in the merge. Since a *rename* is a refactoring, MolhadoRef would be aware of 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 Figure 2.4(b).

Graph-based versioning. Applying the merge on top of the *graph-based representation* depicted in Figure 2.3 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 Figure 2.3. 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 [TELW10, Wes10]. 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 since purely state-based approaches usually take the element's changes of only one 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 Figure 2.5(a).

Purely *graph- and operation-based approaches* are capable of automatically merging the presented example (cf. Figure 2.5(b)). 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, i.e., they are commutative, and therefore, they all can be re-applied to Version 0 in order to obtain a correctly merged version.

To sum up, 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 once because even in small MDE projects several modeling languages are usually combined. Therefore, a generic infrastructure which may be adapted for several modeling languages is as valuable as it is challenging to design.

2.2.2 State of the Art in Model Versioning

In the previous section, general versioning concepts have been introduced without putting special emphasis on *model* versioning. These general concepts, being the result of intensive research efforts of the past thirty years, constitute the basics for dedicated *graph-based model versioning systems*, which emerged more recently. In this section, we focus on the state of the art in model versioning and survey existing approaches in this area. These approaches are categorized based on a set of features they offer and characteristics they have.

Selected Model Versioning Systems

In this section, we introduce current state-of-the-art model versioning systems and evaluate them according to the features and characteristics discussed in the previous section. The considered systems and the findings of this survey are summarized in Section 2.1 and profoundly discussed in the following. Please note that the order in which we introduce the considered systems has no further meaning. The name of the system is stated if it is available.

Alanen and Porres. One of the earliest works on the versioning of UML models was the paper by Alanen & Porres [AP03], who presented various metamodel independent algorithms for difference calculation, model merging as well as conflict resolution. They identified seven elementary operations which are grouped into positive and negative operations. Whereas positive operations add model elements and can therefore be used to represent any model, negative operations have the opposite effect and remove model elements. For calculating the differences between the original version and the modified version, first an unambiguous mapping is created which allows the calculation of the necessary changes to obtain the new version from the old version. For the mapping, unique identifiers of the model elements are required. For the merge, different situations are considered. Conflicts are reported if an updated element is deleted or two ordered features are added. Then manual intervention is necessary for conflict resolution. Finally, metamodel-aware automatic conflict resolution is suggested in order to repair the model in such a manner that broken well-formed rules are again obeyed.

Oda and Saeki The version control support proposed by Oda & Saeki [OS05] builds upon the facilities offered by a meta-CASE tool which allows the construction of modeling editors for arbitrary modeling languages. Together with the typical functionalities also versioning features like the calculation of differences are included into a modeling editor built from a given metamodel. The generated tool offers a menu for performing check-in, check-out, and update operations on the repository. When a model is changed within such an editor, the modifications are recorded and stored to the central repository. Since the modeling editors are newly built by the meta-CASE tool, the necessary functionality is included as required. Model elements are assigned a unique identifier and model specific operations may be defined. Also layout information is considered within the versioning process.

Ohst, Welle, and Kelter. Within their merge algorithm, Ohst et al. [OWK03] put special emphasis on the visualization of the differences. They offer a preview to the user which contains all modifications even if they are contradicting. The diagram shown in the preview may be modified, conflicts may be manually resolved, and automatically applied merge decisions may be undone. For the merge, unique identifiers of the model elements are required. Considered conflicts are update/update and delete/update conflicts. 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 parts, different colors 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.

Mehra, Grundy, and Hosking. The approach of Mehra et al. [MGH05] also focuses on visualization support for comparison and merging tasks in CASE tools. Therefore, they provide a plugin for the meta-CASE tool Pounamu, a tool for the specification and generation of multi-view design editors. The diagrams are serialized in XMI, which are converted into a Java object graph for comparison. The obtained differences are translated to Pounamu editing events which have been applied on the model. Differences cover not only modifications performed on the model, but also modifications performed on the visualization, e.g., editing events like ResizeShape. The differences between various versions are highlighted in the concrete syntax, i.e., in the diagram view, presented to the modeler who may accept or reject modifications. When a modification is accepted it is applied on the model and stored within the repository.

Cicchetti, Di Ruscio, and Pierantonio. Cicchetti et al. [CDRP08] propose a domain- specific language to specify conflicts and conflict resolution patterns. Conflicts are defined based on a proprietary difference model which describes the modifications performed on subsequent versions of one model. To this end, the authors are able to establish an extendable set of conflicts, represented as forbidden difference pattern. By this means, the realization of a customizable conflict detection component is possible. The difference model conforms to a difference metamodel which is dedicated to the used modeling language. This difference metamodel is automatically generated from the metamodel of the modeling language. When two different versions of a common base model evolve, then the merged version may be obtained by composing the two difference models. The result of the composition of two difference models is again a difference model containing the minimal difference set, i.e., only these modifications are included, which have not been overwritten by other operations. When the modifications are conflicting, this conflict has to be reported or resolved by applying a dedicated reconciliation strategy defined for the conflict pattern.

ADAMS. Beside versioning features, the Advanced Artifact Management System ADAMS offers process management functionality, coordination of multiple modelers, and the management of traceability information [DLFOT06]. ADAMS may be integrated via specific plug-ins into various modeling environments to realize model management and context- awareness, i.e., every modeler knows who else is working on the same model element. De Lucia et al. [DLFST09] present an ADAMS plug-in for versioning ArgoUML models. For ArgoUML, XMI files with the diagram information, and additional files with layout information and meta information about the models are considered and transformed into an internal format to be stored within the model repository. If a model is checked-out or updated, the model stored in the repository is again converted back to the tool specific format. On the client-side, the deltas are calculated when the model is modified based on the assumption that unique identifiers are available. Only the deltas are committed to the central versioning server where the merge process is performed. In ADAMS it is possible to configure the unit of comparison. Changes to uncorrelated elements are automatically merged, whereas for conflicting modifications manual intervention is necessary. For newly introduced model elements, simple matching heuristics are applied to check whether another modeler has introduced the same element. If a potential duplication is detected, it is reported to the modeler and like in a conflict situation manual intervention is nec-

essary. In ADAMS, the layout information is also considered as a model and is therefore also set under version control.

AMOR. To better compare our approach with the state-of-the-art systems, we also cover AMOR in this evaluation. Our model versioning system AMOR [BKS+10], which is presented in more detail in Chapter 3, implements a conflict detection component for the EMF metamodeling language Ecore which reports not only conflicts resulting from atomic changes, but also from composite changes. These composite changes are not tracked following an operation-based approach, but they are recalculated based on the versions of one model which are potentially conflicting. Due to this state-based approach, the conflict detection of AMOR is independent of any modeling environment. For handling the conflicts, AMOR offers two different approaches: (i) immediate conflict resolution and (ii) conflict tolerance, i.e., living with inconsistencies. If the conflicts shall be resolved immediately, a conflict resolution recommender guides the modeler during the conflict resolution process by suggesting potential, automatically executable resolution patterns. In some situations, it might be preferable, to defer the conflict resolution to a later point in time. AMOR offers a mechanism to incorporate the changes of any modeler into one model which is annotated with information about the conflicts [WLS+11].

CoObRA. The Concurrent Object Replication framework CoObRA developed by Schneider et al. [SZN04] realizes optimistic versioning for the UML case tool Fujaba[7]. CoObRA records the changes performed on the model elements and aligns incremental changes into groups. The change protocols are committed to a central repository. When other modelers want to update their local models, these changes are fetched from this repository and replayed on the local model. To identify equal model elements, unique identifiers are introduced and the model elements are enhanced with versioning information. Conflicting changes are not applied (also the corresponding local change is undone) and finally presented to the user who has to resolve these conflicts manually. Repair mechanisms to fix model inconsistencies resulting from the merge are shortly reported.

EMF Compare. The open-source, Java-based component EMF Compare [BP08], which is part of the Eclipse Modeling Framework Technology (EMFT) project[8], supports generic model comparison and model merging. EMF Compare reports differences between Ecore models based on two-way or three-way comparison approaches. Within the Eclipse environment, the differences are indicated on a tree-based representation of the models where conflicting changes are highlighted in a dedicated color. Programmatic access of EMF Compare is also possible. For comparing two models, EMF Compare distinguishes two phases: a matching phase and a differencing phase building a match model as well as a difference model. The matching phase relies on four metrics based on type, name, value, and relationship similarity. The difference model provides information about inserted, deleted, and updated elements. The comparison and merge algorithms are kept generic in order to make them applicable for any Ecore-based modeling language, but the adaption to language-specific features is explicitly intended.

[7] `http://www.fujaba.de`
[8] `http://www.eclipse.org/modeling/emft`

IBM Rational Software Architect (RSA). The RSA[9], a UML modeling environment built upon the Eclipse Modeling Framework, provides two-way and three-way merge functionality for UML models. During the merge, not only syntax and the low-level EMF semantics are considered, but even the semantics of UML elements is taken into account. The differences are shown either in a tree-editor, or directly in the diagram. If the later view on the differences is chosen, then modified elements are highlighted. Conflict resolution must be done by the modeler manually, by either rejecting or accepting changes. Furthermore, the RSA offers a model validation facility which checks the conformance of the merged version to the UML metamodel.

EMF Store. The model repository EMF Store presented by Koegel et al. [KHWH10], which has been initially developed as part of the Unicase[10] project, provides a dedicated framework for model versioning of EMF models. When a copy of a model is checked out, changes are tracked within the client and committed to the repository. With this operation-based approach, an efficient and precise detection of composite changes is possible coming along with the drawback that composite operations like refactorings are only detectable if they are explicitly available within the modeling editor. Changes obtained from the head revision of the repository and the changes of the local copy, which have not been checked in so far, are considered. Having the two lists of the performed changes, two kinds of relationships are established: "requires" and "conflicts". Whereas the former relationship expresses dependencies between operations, the later emphasizes contradicting modifications. Since the exact calculation of requires and conflicts relationships would be too expensive, heuristics are applied to obtain an approximation. To keep the conflict detection component flexible, a strategy pattern is implemented, which allows the adaption to specific needs. For example, in Koegel et al. [KHWH10], the *FineGrainedCDStrategy* is proposed, which works on the attribute and reference level. Basically, two changes are conflicting, if the same attribute or the same reference is modified. All operations are classified to a few categories for obtaining potentially problematic situations. 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 an inconsistent model). A wizard guides the merge process.

Odyssey-VCS. The version control system Odyssey-VCS by Oliveira et al. [OMW05] is dedicated to versioning UML models. For each project, behavior descriptors may be specified which define how each model element should be treated during the versioning process. For the conflict detection, it may be specified which model elements should be considered atomic. If an atomic element is changed in two different ways at the same time, a conflict is raised. Behavior descriptors are expressed in XML and therefore, Odyssey-VCS is customizable for different projects. In the merge algorithm, all possible scenarios are considered, and the resulting actions, such as safely adding both operations, reporting a conflict, doing nothing, etc., are taken. A validation of

[9] `http://www.ibm.com/developerworks/rational/library/05/712_comp/index.html`
[10] `http://www.unicase.org`

the resulting model is not provided. Odyssey-VCS may be used either with a standalone client or with arbitrary modeling tools. The communication with the server is realized with Web services. More recently, Odyssey-VCS 2 by Murta et al. [MCPW08] has been released which is built on top of Ecore resulting in a gain of flexibility concerning reflective processing of the model elements. Consequently, the conflict detection and merge algorithm is expressed in a more generic manner. Additionally, Odyssey-VCS is capable of both, pessimistic versioning and optimistic versioning. In the latter case, explicit branching is performed for storing not only the merged version, but also the working copies the merge is based on.

SMOVER. The semantically-enhanced versioning system SMOVER by Reiter et al. [RAB+07] aims at reducing the number of falsely detected conflicts resulting from syntactic variations of one modeling concept. Furthermore, additional conflicts shall be identified by using knowledge about the modeling language. This knowledge is encoded by the means of model transformations which rewrite a given model to so-called semantic views. These semantic views provide canonical representations of the model which makes certain aspects of the modeling language more explicit. Consequently, more precise information about potential conflicts might be obtained when the semantic view representation of two concurrently evolved versions are compared.

Features and Characteristics of Model Versioning Approaches

Physical Model Management (MM). At some point in time, it is necessary to physically store the model versions. Therefore, a repository is required as well as a format in which the model versions may be accessed by the versioning system.

Repository: Whereas some systems offer a complete solution with an integrated repository where the historical information of the artifacts is stored, other approaches realize only the model merge component and rely on available repositories, which administrate files of arbitrary kinds.

Standard Format: The models may either be serialized in a standard format, i.e., XMI, or a format specific for the editor/versioning system. Consequently, transformations might be necessary before the versioning system may be used. If a direct import and processing of the XMI serialization, like it is the case in AMOR and Odyssey-VCS, versioning might be performed independently of any modeling editor.

Differences. The various systems follow different approaches how differences are obtained and represented, which are the basis for the calculation of conflicts.

Operation Tracking: Overall, the differences are either calculated retrospectively by a state-based algorithm or directly tracked during the modeling activity in operation-based approaches. Concerning the latter, more information is available, but a tighter coupling to the editors is given. All approaches belong to one of these two categories, only AMOR is a special case as discussed in the previous section. In AMOR, the atomic changes are obtained by a state-based comparison, from which composite changes might be retrospectively recovered.

Matching Heuristics: All approaches use unique identifiers to match elements occurring in all versions. EMF Compare, ADAMS as well as AMOR additionally apply certain heuristics when no identifiers are available and to be able to match newly introduced elements.

Difference Model: Some approaches like EMF Compare, AMOR, and Cicchetti et al. represent differences as model. The differences are described in terms of operations from which, the revised model may be recreated when applied to the origin model.

	MM		*Differences*			*Conflicts*				*Flexibility*								
	Repository	**Standard Format**	**Operation Tracking**	**Matching Heuristics**	**Difference Model**	**Conflict Model**	**Conflict-tolerance**	**Conflict Dependencies**	**Graphical Visualization**	**Automatic Resolution**	**Layout Information**	**Modeling Language**	**Modeling Editor**	**Unit of Comparison**	**Detectable Operations**	**Detectable Conflicts**	**Collaborative Resolution**	**N-Way Merge**
Alanen and Porres										×		×	×					
Oda Saeki	×		×								×	×						
Ohst et al.	∼								×		×							
Mehra et al.			×						×		×	×	×					
Cicchetti et al.					×	×				×		×		×	×	×		
ADAMS	×		×	×							×		×	×	×			
AMOR	∼	×		×	×	×	×	×	×	×	∼	×	×		×	×	×	×
CoObRA	×		×						×									
EMF Compare		×		×	×	×						×	×					
RSA	∼	×			×	×												
EMF Store	×		×		×					×		×	×	×	×	×		
Odyssey-VCS	∼	×										×	×	×				
SMOVER	∼	×									×	×	×	×	×			

× ... *applicable feature*

∼ ... *partly applicable feature*

Table 2.1: Evaluation of State-of-the-Art Model Versioning Systems

Conflicts. When modifications are contradicting, conflicts have to be reported. The various systems follow different paradigms to represent, to report, and finally to resolve conflicts.

Conflict Model: Some approaches like EMF Compare, AMOR, and Cicchetti et al. consider conflicts as first-class citizens and encode them not only implicitly within the algorithms. In these approaches, dedicated conflict models are specified. The explicit specification of a conflict model allows the serialization and an extended processing of conflicts.

Conflict-tolerance: Beside several conflict resolution strategies in versioning systems, the possibility to tolerate conflicts to resolve them later on is only provided by AMOR.

Conflict Dependencies: Conflicts might depend on each other. This means that the resolution of one conflict makes the resolution of an other conflict obvious and in some cases even unnecessary. AMOR groups different kinds of conflicts occurred between several concurrent changes of the same model element.

Graphical Visualization: Most approaches report conflicts not using the concrete model syntax, but a tree representation, only. For the human user, much information is lost this way. Some approaches are able to decorate the models with the information about conflicts in the concrete syntax. Only a few approaches exist which highlight changes using coloring techniques as proposed by Ohst et al. [OWK03] and Mehra et al. [MGH05]. However, these approaches require the implementation of special editor extensions. Thus, to the best of our knowledge,

AMOR started the first attempts of tackling these challenges using UML Profiles [BKL+11b] and, furthermore, EMF Profiles [LWWC11].

Automatic Conflict Resolution: Most approaches require manual conflict resolution. In AMOR and the approach of Cicchetti et al. [CDRP08], automatically executable conflict resolution patterns are defined which are recommended to the modeler in charge of the conflict resolution. In EMF Store, hard and soft conflicts may be defined. Soft conflicts do not require any user intervention and only a warning is shown.

Layout Information: When models are modified, also the layout of the diagram is potentially changed. Some systems like the approaches of Oda and Saeki [OS05]) and Mehra et al. [MGH05] also consider layout information to be put under version control. In most versioning systems, however, this information is neglected and no dedicated merging actions are provided.

Flexibility. The versioning systems often put special emphasis on being independent from any modeling language and modeling editor and being extensible with respect to the detectable operations, the detectable conflicts, and the automatically applicable resolution patterns.

Modeling Language: The versioning systems which foster language independence, require the modeling languages whose models shall be put under version control to be specified either in Meta Object Facility (MOF) or in Ecore. Other approaches consider one language only, for example, CoObRA is implemented for models formulated in Fujaba, only.

Modeling Editor: Whereas some versioning systems are tightly integrated with a modeling editor, other versioning systems aim for tool independence. Even operation-based systems might be designed in such a way that they may be used with different editors - then customized plugins have to be implemented, like in ADMAS and EMF Store.

Unit of Comparison: Some versioning systems, such as Odyssey-VCS and ADAMS, allow the configuration of the granularity level. It is possible to specify which model elements are considered as atomic and which have to be further decomposed. This configuration directly influences the number of reported conflicts.

Detectable Operations: In most versioning systems, the set of detectable operations is fixed. In some versioning systems, this set may be extended. For example in AMOR, this extension is supported by the means of operation specifications. In this way, AMOR may be extended to detect composite operations without programming effort.

Detectable Conflicts: In most versioning systems, the detectable conflicts are hardcoded into the conflict detection algorithm. AMOR's conflict detection uses the information stored in the pre- and postconditions of the operation specifications. With the addition of new operation specifications, additional conflicts are therefore detectable. Cicchetti et al. propose to describe conflict patterns by the means of models. The set of conflict patterns is extensible. Thus, further conflicts than the simple Update/Update and Delete/Update conflicts may be described and detected. A detailed discussion of these further potential conflicts is given in the next section.

Collaborative Resolution: In nearly all versioning systems, the user who is checking-in her changes is responsible for resolving the occurred conflicts. In AMOR, collaboration is also supported when resolving conflicts. Several users may be assigned to a conflict, who may prioritize how to resolve it.

N-Way Merge Only AMOR provides the possibility to merge more than two parallel versions into one common merged model as proposed by [BLS+10b] and elaborated in this thesis. this means, that the changes and occurred conflicts between the first two parallel versions are also considered, when a third (or N) version is committed.

2.3 Summary

In this chapter, we have surveyed related work in the area of collaborative modeling and state-of-the-art model versioning systems. As we have seen in Section 2.1 different approaches for developing models have emerged. There are many techniques and tools focusing at the synchronous development of models. With face-to-face sessions or remote interactions conflicts can be minimized or even avoided in advance, but working at the same time is not always possible or even required. Thus, to develop software asynchronously and probably geographically dispersed, version control systems are indispensable. Especially optimistic versioning control systems put no dependencies on developers. Thus, also for models, optimistic versioning is requested. All of the presented state-of-the-art versioning approaches contribute important ideas and concepts for building reliable model versioning systems, whereby for some of them deployable software or even the source code is available; others are still subject to ongoing development work. When we consider the time of publication, the majority of the systems has been presented in the last few years. This might be closely related with the maturity of the Eclipse Modeling Framework, which offers a sophisticated environment for the development of such model manipulation tools.

As we have seen in Section 2.2, the different versioning systems tackle very manifold challenges of the versioning process and therefore it is kind of difficult to directly compare the systems, or even perform a competitive evaluation. However, supportive conflict resolution poses also a big challenge in model versioning and, after evaluating current systems, we can say that it is mainly neglected. To tackle these deficiencies, collaborative modeling approaches can be taken as inspiring examples. Conflicts should not be seen as negative results of collaboration but as chance to discuss design alternatives or misinterpretations. Thus, we enhanced AMOR in such a way that the users are not left alone and, additionally, are better supported when resolving conflicts. Beside a better visualization of conflicts and the consideration of conflict dependencies, we have introduced a conflict tolerant approach to incorporate all versions of the participating modelers and to annotate occurred conflicts building a good basis for resolving the conflicts collaboratively leading to a consolidated, merged model. Special focus is set on change and conflict awareness (cf. [TG04]) to better track asynchronous changes of all participants by highlighting what changes have been performed, where and when have changes been performed and who has performed them with the goal to better understand the reason behind them. In the next chapter, the model versioning system AMOR is presented in more detail.

CHAPTER 3

A Tour of AMOR

The content of this thesis has been elaborated in the context of the national funded project AMOR—Adaptable Model Versioning. In this chapter we give an overview of the model versioning system developed within AMOR, which is also called AMOR. For further details on the overall project, we kindly refer to our project homepage[1]. The overall research goal of the project AMOR is to provide an adaptable versioning framework allowing for proper versioning support while ensuring generic applicability for various Domain Specific Modeling Languages (DSML) [AKK+08]. In particular, within this project three major aspects are considered:

(a) The adaptation of the framework with language specific information allows a more precise conflict detection on the one hand and a more compact conflict report on the other hand, that is presented in the PhD thesis of Philip Langer [Lan11].

(b) Furthermore, AMOR supports the developer when resolving conflicts with the help of dedicated suggestions. In addition, not only the abstract but also the concrete syntax of a model is considered in this phase. These two points are elaborated in the PhD thesis of Petra Brosch [Bro11].

(c) Orthogonally to both aspects, appropriate support for developing models in teams is presented in this thesis. Conflicts are not seen as negative results of collaboration, but as chance for discussing ideas and design alternatives. Thus, a conflict-tolerant approach is presented including dedicated merge rules which additionally enrich the models with meta-information that supports asynchronous collaborative modeling.

Since AMOR is conducted with the (meta-)modeling framework EMF and Ecore, for the sake of completeness, an introduction of these concepts and techniques are given in the following section. In AMOR as well as in all other optimistic model versioning systems, coping with conflicts plays a central role when two or more developers work on the same artifact in parallel

[1] Project AMOR (FIT-IT No. 819584), www.modelversioning.org

and independent of each other. Thus, in the following, we present our categorization of conflicts which might occur when the same model is changed by two developers in parallel. Such a taxonomy of conflicts between model versions is indispensable for the successful establishment of dedicated model versioning systems.

Furthermore, we present the basic workflow of AMOR in Section 3.3 by outlining the major components.

3.1 Introduction to (Meta-)Modeling with EMF and Ecore

When talking about models in this thesis, we refer to models that are based on the Eclipse Modeling Framework[2] (EMF) [SBPM08]. EMF is a matured Eclipse-based framework providing a powerful metamodeling support within the Eclipse ecosystem. During the last years, EMF has found significant recognition among researchers and practitioners, which is also why we choose EMF as the underlying modeling technology. EMF offers, besides the meta-metamodeling language Ecore, which is introduced below, facilities for code generation, generation of modeling editors, reflective APIs to generically access and manipulate models, and much more. Based on EMF many very powerful technologies have been built, which allow, for instance, to persist models in relational databases, to transform models, and much more. In the following, however, we focus on introducing the metamodeling language Ecore and discuss its relationship to the well-known metamodeling stack [KÖ06].

The heart of EMF is its metamodeling language *Ecore*, a Java-based implementation of the Essential Meta Object Facility (EMOF) [Obj04] standardized by the Object Management Group (OMG). Using Ecore, developers may specify a metamodel to define the abstract syntax of a new modeling language. This metamodel may then be used to generate modeling editors for creating models, that is, instances of the developed metamodel. The relationship among meta-metamodels, metamodels, and models may best described in terms of the metamodeling stack [KÖ06]. The metamodeling stack consists of three layers called *M3*, *M2*, and *M1* whereas a model in *M2* is an instances of a model in *M3* and a model in *M1* is an instance of a model in *M2*.

M3: Meta-metamodel. In the most upper layer in the metamodeling stack, namely *M3*, the meta-metamodeling language is located (cf. Figure 3.1). In the context of EMF, this meta-metamodeling language is constituted by Ecore. The core language elements of Ecore are depicted in the upper area of Figure 3.1 in terms of a UML®class diagram. Please note that we do not present all language elements and features in this figure. Instead, we concentrate on those classes and features that are of paramount importance in the current context. Ecore allows to model `EClasses`, which may contain an arbitrary number of structural features. For structural features upper and lower multiplicities have to be defined. Additionally, structural features having an upper multiplicity greater than 1, may be defined as ordered. Structural features are divided into two distinct subsets, namely `EReferences` and `EAttributes`. Attributes as well as references must have a type. For attributes, primitive data types such as String, Boolean,

[2] `http://www.eclipse.org/modeling/emf`

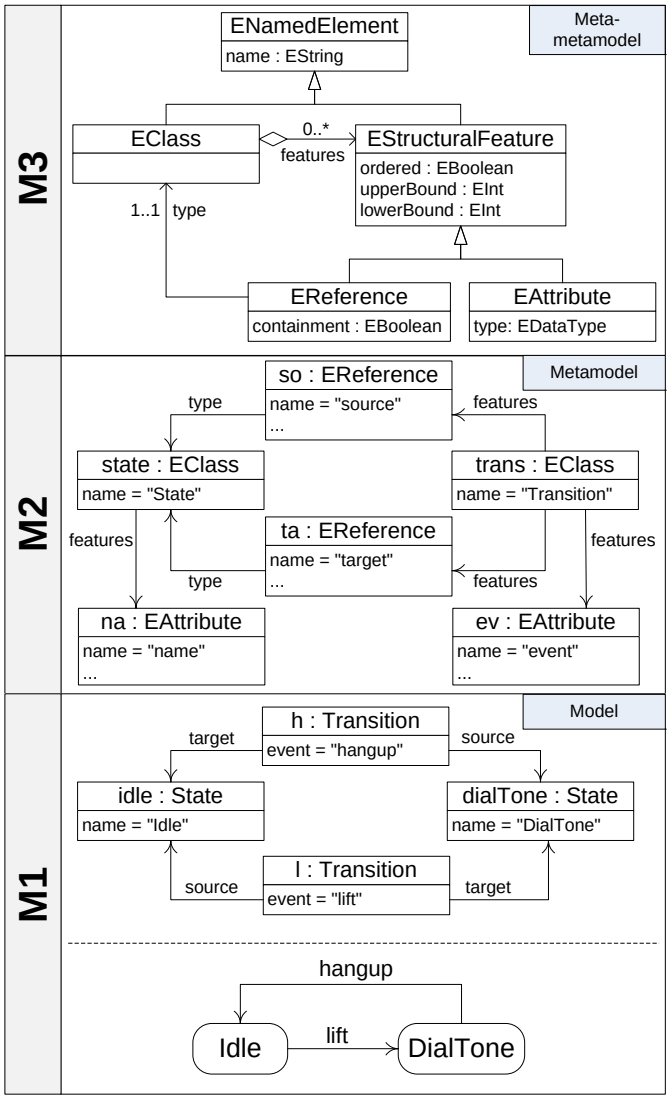

Figure 3.1: Metamodeling with Ecore

and Integer are allowed. References refer to classes for defining their types and may additionally be defined as containments. This means that referenced elements are nested inside the container element and, therefore, the deletion of a container element results in cascaded deletions of all directly and indirectly contained elements. It is worth noting that Ecore is recursively specified again by Ecore. This means that, for example, `EReference` is indeed an instance of `EClass` having the name "EReference". This class again contains, for instance, the structural feature "containment", which is an instance of `EAttribute` and so on.

M2: Metamodel. The meta-metamodeling language may now be used to create metamodels. A metamodel specifies the abstract syntax of a modeling language and is an instance of Ecore, which resides in *M3*—therefore a metamodel resides on *M2*. In Figure 3.1, we provide a small example of such a metamodel in terms of an object diagram. In particular, this metamodel is a simplified excerpt of the state machine metamodel. Basically, a state machine consists of `States` and `Transitions`. Therefore, we have two instances of Ecore's `EClass`, one for states and one for transitions. Both classes contain an attribute (i.e., an instance of Ecore's `EAttribute`): a state has a `name` and a transition has an `event`. Transitions further refer to the source state and the target state. Therefore, the metamodel for state machines contains two instances of `EReferences`, namely `source` and `target`.

M1: Model. The metamodel in *M2* may now be instantiated to specify arbitrarily many state machines on *M1*. In Figure 3.1, we illustrate a small state machine comprising two states and two transitions between those states. More precisely, the states are instances of the corresponding class `State` in the metamodel residing in *M2*. In the upper area of *M1* in Figure 3.1, the small state machine model is depicted in terms of an object diagram and in the lower area of *M1*, the same model is illustrated by the commonly used concrete syntax of state machines for the sake of readability.

3.2 Conflict Categorization

As discussed in the previous sections, one key element in model versioning is a conflict. However, the term conflict is strongly overloaded and differently co-notated. In the case of metamodel violations, the term conflict is used synonymously to the term inconsistency. Current model versioning systems mainly focus on single changes that are directly contradicting as they may be detected in an efficient and language independent way. Nevertheless, there is a multitude of further problems which could occur when merging two independently evolved models. Therefore, in this section we first present a comprehensive categorization of conflicts based on our previous work [BKL+11b, BLS+10a, BKL+11a].

The practical application of versioning systems depends on the quality of the merge component, especially on model comparison and conflict detection. In general, merge conflicts on models may occur either if one change invalidates another change, or if two changes do not commute [LvO92]. In order to better understand the notion of conflicts, different categories were set up to group specific issues. In the field of software merging, *textual*, *syntactic*, *semantic*, and *structural conflicts* were surveyed in [Men02]. While textual conflicts are detected by a line-based comparison of the program (cf. Section 2.2), syntactic merging operates on the parse tree or abstract syntax graph, and thus, ignores conflicts resulting from textual reformatting. A syntactic-aware merging component takes the programming language's syntax into account and reports conflicts causing parse errors. Semantic merging goes one step further and reflects the semantic annotation of the parse tree, as done in the semantic analysis phase of a compiler. Here, static semantic conflicts like undeclared variables or incompatible types are detected. A structural conflict occurs due to changes overlapping with restructured and refactored parts of the program. Then, it is not decidable where to integrate the changes.

First attempts to apply this conflict categorization to model versioning failed. In fact, the term semantics itself is in the field of modeling heavily overloaded [HR04], referring to language semantics and real-world semantics. The assignment to a certain category is often based on objective preferences and, consequently, ambiguous. Already in small models, a separation between syntactic conflicts and semantic conflicts quickly blurs, especially when the metamodel is enriched with additional constraints such as OCL constraints in the case of UML. For instance, two modelers are working on a UML class diagram consisting of the two classes `Circle` and `Ellipse`. Each of them adds an inheritance relationship between the two classes. Unfortunately, both modelers disagree regarding the direction of the generalization and introduce a new inheritance relationship in the opposite direction. As a result, the merged model contains an inheritance cycle. This conflict may be either referred to as syntactic since such cycles are forbidden by the UML metamodel, or as semantic, since it is not clear how to interpret the case, that a class is subclass of its own subclasses. Further, even, if such conflicts could be clearly assigned to those categories, it would not provide any insights on how to detect and resolve that kind of conflicts.

From the analysis of many conflict scenarios, we learned during the development of AMOR that the number of reasons *why* a conflict might occur is limited in regards to the existing language definitions. We identified two main groups of conflicts, namely overlapping changes and violations (cf. Figure 3.2).

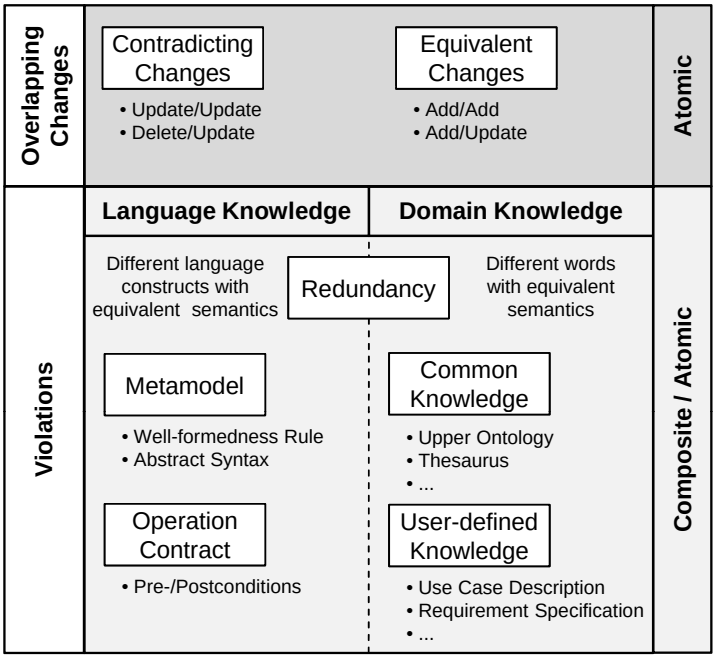

Figure 3.2: Conflict Categorization

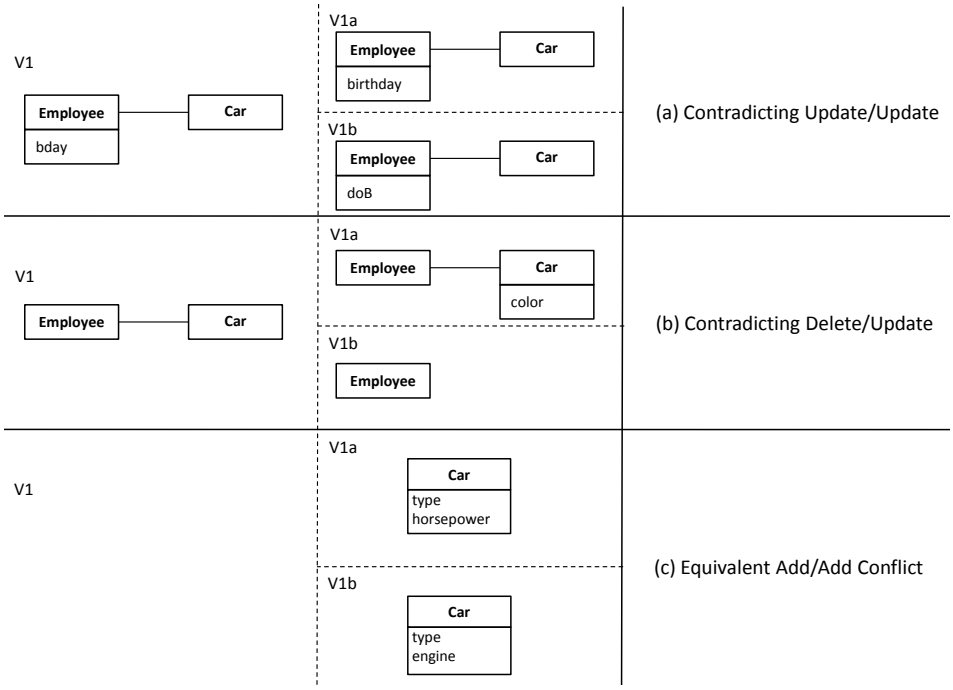

Figure 3.3: Conflict Examples: Overlapping Changes

3.2.1 Overlapping Changes

Overlapping Changes refer to two opposite atomic changes (*add, delete, update*) on an overlapping part of the model with respect to the unit of comparison, e.g., a feature of a model element or a container element. We further distinguish between two types of overlapping changes, namely *contradicting changes* and *equivalent changes*. While conflicts of the first category arise due to directly competing changes, the latter category covers parallel changes leading to an equivalent result.

Contradicting changes. Such changes find their expression in update/update and Delete/Update conflicts. Update/Update conflicts occur when an existing element of the common ancestor model is changed in both versions differently (cf. the multiplicity in Figure 3.3 (a)). Delete/Update conflicts emerge either due to the concurrent update and deletion of the same element, or due to an update of an element and the deletion of the container element, e.g., a property is added and the corresponding class is deleted, like in the example in Figure 3.3 (b).

Equivalent changes. If parallel Update/Update, Delete/Delete and Add/Add changes are the "same", they are referred to as equivalent changes and only one of the two changes has to be integrated into the merged version to completely reproduce the intention of both modelers. In the case of add/add changes, no common ancestor of the affected model element is available, but redundant elements are added to the merged versions if all changes are naively merged. If the

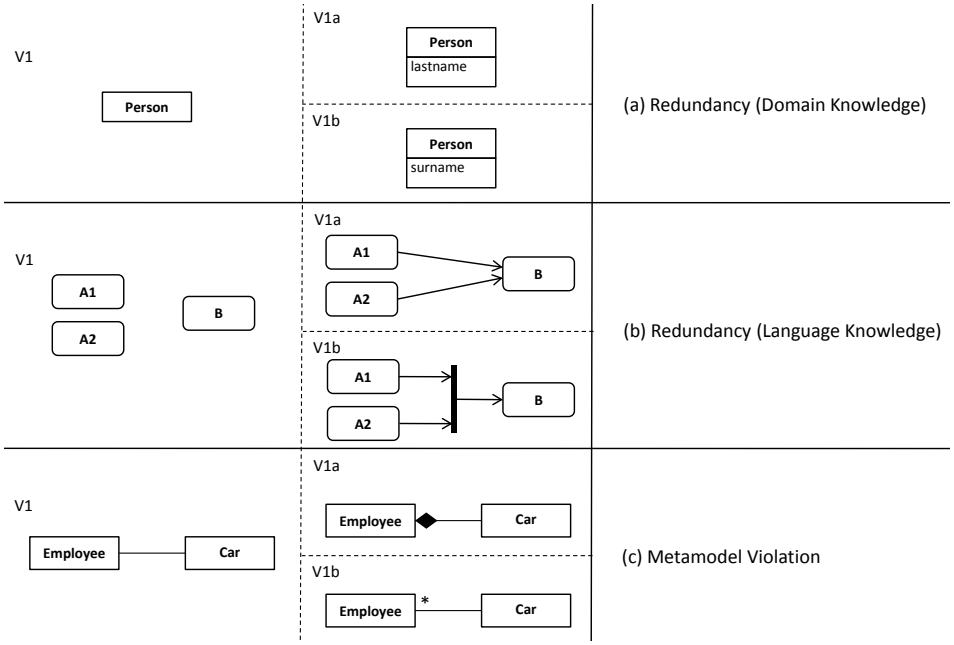

Figure 3.4: Conflict Examples: Redundancies and Metamodel Violation

duplicate elements are deep equal, i.e., all features and containments have the same values, only one of the elements should be inserted in the merged model and no conflict should be reported. However, if there are slight differences like the properties `horsepower` and `engine` in the classes `Car` of Figure 3.3 (c), an add/add conflict should be raised.

3.2.2 Violations

Besides the directly competing changes which are overlapping in terms of editing the same element in the "materialized" model, also combinations of changes in different elements may lead to an inconsistent model. This kind of conflict is harder to detect, as additional knowledge is necessary. This knowledge regards the underlying modeling language and the modeled domain.

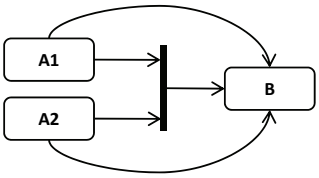

Figure 3.5: Naive Merge of Example (b) of Figure 3.4

Redundancy may be introduced to a model either by equivalent modeling concepts or variations in natural language expressing equivalent facts. Figure 3.4 (a) shows a redundancy conflict, as both modelers add the properties `lastname` and `surname` to the class `Person` in parallel, which are synonyms. Different modeling concepts may also express equivalent semantics. For example, the actions `A1` and `A2` of the UML activity diagram depicted in Figure 3.4 (b) may trigger the execution of action `B` using an implicit join by modeling a simple control flow, or by modeling the synchronization explicitly using a join node. As defined in the UML 2.3 Standard [Obj10], an action may by

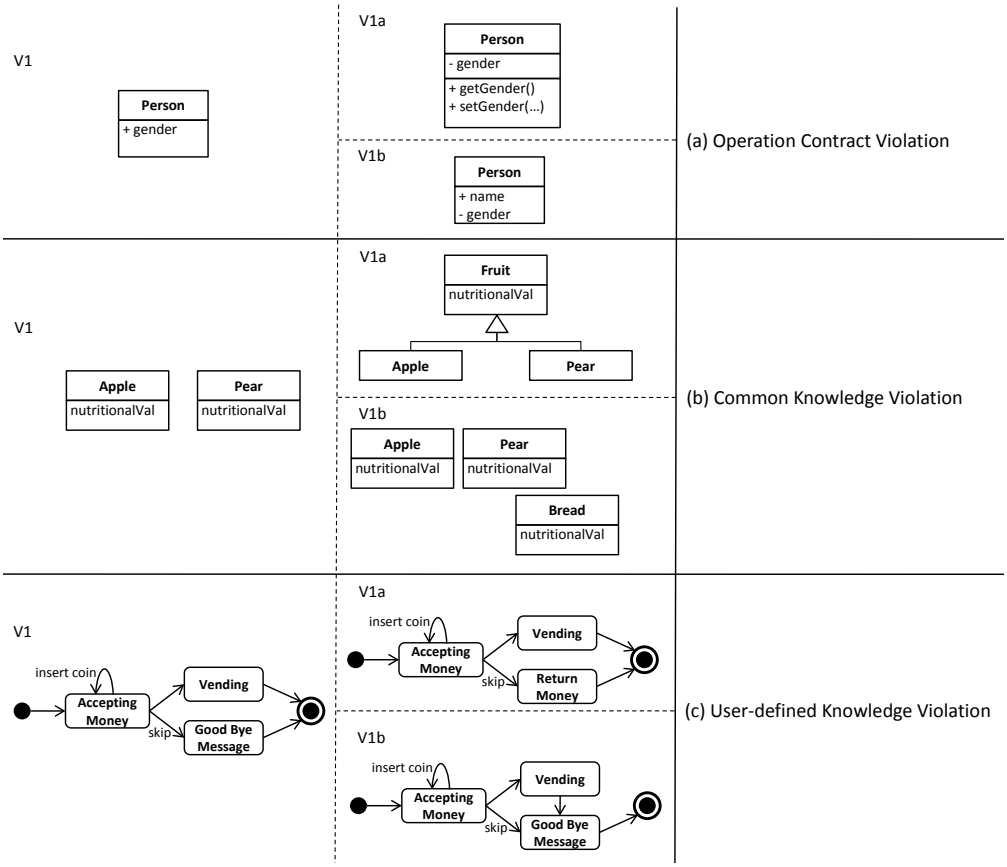

Figure 3.6: Conflict Examples: Operation Contract Violation and Domain Knowledge Violation

default only start its execution when all incoming control flows offer a control token. Thus, Version 1 and Version 2 have equal semantics, i.e., the execution of `A1` and `A2` is required to execute B. However, naively merging these two variants would not only result in a redundant model, further, redundant parts in the model might lead to unexpected traces. This is also the case for our example depicted in Figure 3.5. The merge leads to multiple outgoing control flows and, according to the UML 2.3 Standard [Obj10], introduces race conditions, as the token is offered to all outgoing edges, but may be accepted by only one target. B requires one token on each incoming control flow, which is never fulfilled. Thus, a deadlock occurs.

Metamodel violations may also be caused by concurrent changes, which do not overlap, but lead to an inconsistent model with respect to formal language constraints like the metamodel itself or additional OCL constraints. For an example of a metamodel violation, consider a UML class diagram consisting of the two classes `Employee` and `Car`, and an association between them (cf. Figure 3.4 (c)). One modeler change

Figure 3.7: Naive Merge of Example (c) of Figure 3.4

the association to be composite and the other sets the association end to unbound (*). They do not syntactically overlap, thus, when merging naively as depicted in Figure 3.7), a car would be part of multiple employees which is restricted by the UML metamodel.

Operation contract violations denote conflicts where a composite operation applied on one version is invalidated by a change of the other version. A composite operation is a set of associated atomic changes necessary to perform a larger change like a refactoring. Each composite operation formulates a contract in terms of pre- and postconditions, e.g., requiring the existence or non-existence of specific elements, or specific values for features. Only the union of all atomic changes reflects the intention of the change and therefore, if a change of the opposite version violates the contract, a composite operation should not be divided and partially applied. Figure 3.6 (a) shows an example for an operation contract violation. One modeler applies an `encapsulate field` refactoring to all public properties, i.e., `name` and `gender`, of class `Person`. The refactoring sets the properties to private and generates public getter and setter methods. A parallel change setting the property `gender` to private invalidates the operation contract and a conflict should be reported.

Common knowledge available in upper ontologies, thesauri, or other kinds of knowledge bases may be used to detect violations of real-world semantics. Consider two classes `Apple` and `Pear` with a common property `nutritionalVal` (cf. Figure 3.6 (b)). While one modeler applies a `pull up field` refactoring to shift the common property to a newly created superclass named `Fruit`, the other modeler adds a new class `Bread` with the same property to the class diagram. A refactoring aware merge would include the newly introduced class into the `pull up field` operation, resulting in a `Bread` of type `Fruit`, which does not reflect reality.

User-defined knowledge from use case descriptions, requirement specifications and other models restricts the modeled domain and therefore may be used to detect further violations. With such information available,

conflicts like the one depicted in Figure 3.6 (c) may be detected. A simple vending machine, expressed as UML state machine, containing the three states `accepting money`, `vending`, and `Good Bye Message` is refined by two modelers in parallel. One modeler changes the state `Good Bye Message` to `Return Money`, to express the fact, that in case of aborting the vending process, the inserted money should be returned. Concurrently, the other modeler changes the flow going from `Vending` to the final state to show the `Good Bye Message` in between. In the merged version depicted in Figure 3.8, the vending machine would return the money after vending, which may contradict requirement specifications.

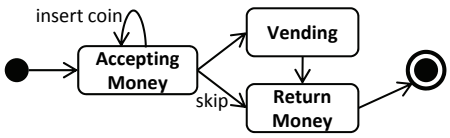

Figure 3.8: Naive Merge of Example (c) of Figure 3.6

3.3 The AMOR Workflow

In this section, we show how the model versioning system AMOR supports the developers to merge the different versions of the same model. Basically, AMOR's conflict detection component reports conflicts due to overlapping changes and violations concerning language knowledge (cf. Figure 3.2).

In the following, we outline how the different components of AMOR operate. An overview of the basic workflow of AMOR is presented in Figure 3.9.

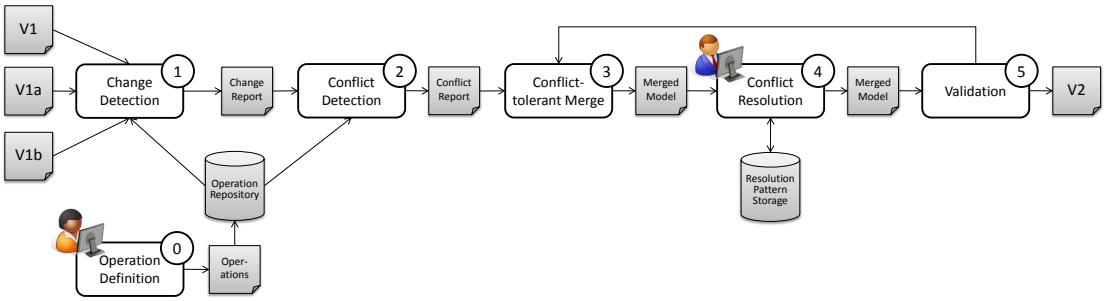

Figure 3.9: The AMOR Workflow

0: Operation Definition. In step 1 the merge process comprises the detection of changes. Beside atomic changes like *insert*, *update*, and *delete*, modelers may also have performed composite operations like refactorings. Detecting and regarding these composite operations enhances the preservation of both modelers' intentions, as reported in [DJ06]. Of course, composite operations are always specific to a certain modeling language. The AMOR conflict detection component may be enhanced with user-defined composite operation specifications, which describe composite operations in terms of a set of atomic operations and necessary pre- and postconditions. AMOR provides a tool called EMF Modeling Operations[3] to easily specify composite operations for specific languages by example [BLS+09]. These operation specifications are then either interpreted by the AMOR system or used to derive executable representations like graph transformations.

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

[3] http://www.modelversioning.org/emf-modeling-operations

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

3: Conflict-tolerant Merge. All detected changes are now incorporated within a new merged version of the model with the help of dedicated merge rules. In addition, conflicts are marked by a language independent model annotation mechanism, which do not destroy the syntax of the model and, thus, the dependence to its metamodel is ensured. Furthermore, the detected conflicts are enriched with meta-information such as the responsible developers and do not have to be resolved immediately. This, approach gives the developers the chance to assign developers to conflicts, to discuss them and, finally, to find a consolidated version of model, which reflects the intentions and requirements of all developers.

4: Conflict Resolution. The afore assigned users can now prioritize one of the two conflicting operations or propose a custom resolution of a concrete conflict. After this resolution proposal is reviewed by the others and finally accepted, the conflict annotation is hidden but still available in the central model repository for provenance reasons. In addition, the developers are also supported by resolution suggestions, which may be either manually defined or they are learned from the situations when the modelers resolve the conflicts.

5: Validation. In the validation phase, the merged model is validated. Validation means to reveal violations of rules and constraints defined by the modeling language. Whenever a violation is detected diagnostics are returned which describe the severity of the constraint violation and provide an error message. These violations are then treated like other conflicts and, thus, they also have to be resolved.

CHAPTER 4

Survey on Versioning in Practice

In the heterogeneous field of model versioning, a plethora of research directions exist trying to meet the technical challenges of model versioning systems, mostly concerned with precise conflict detection and supportive conflict resolution [ASW09]. However, there is currently a lack of empirical studies trying to derive the "real" needs of software developers in practice concerning the collaborative development of software systems [Men02]. Such studies are highly needed, because several possibilities exist for how software can be developed collaboratively as presented in Section 2.1. Furthermore, lessons learned of current best practices in collaborative software development for the various development artifacts may be inferred from such studies. To the best of our knowledge, only few investigations have been carried out in order to find answers to these questions. For example, several issues arising from practice when merging different versions of a model are identified in [BE09]. However, these findings are based on informal interviews within one company and they do "*not pretend to be general*" applicable. Furthermore, the premise of [BE09] is that "*model-centric development and its problems do not vary much from company to company*" which has not been proven so far. However, models can be used in different ways, namely as sketch to discuss ideas and design alternative, as blueprint for implementation, or for direct code generation [Fow03] and, thus, the collaborative development of models varies from company to company.

To tackle these mentioned deficiencies, this chapter provides a comprehensive empirical study, including on the one hand a survey and on the other hand in-depth qualitative interviews. The overall goal of the empirical study is to gain insights of how versioning is currently used in practice. By conducting an online survey, we wanted to find out the state-of-the-art habits and processes of versioning in general. On the basis of these results, we further conducted interviews with experts, to get insights how models are handled in their companies when developing them in team. By using this comprehensive approach, we are able to reason more objectively about the influence of certain team characteristics on the collaborative software development. Furthermore, these studies gave as valuable insights on collaborative software development, which point out many interesting research issues, of which some of them are tackled in this thesis. The next section presents the results of an online survey with 90 participants to get an overview of

general versioning habits and processes in practice. Subsequently, to get deeper insights into status quo, experiences, and requirements of software *and* model versioning, we additionally conducted 10 expert interviews to understand best-practices in industry. Talking directly with software developers and IT managers of different domains has led to interesting and important findings, which are presented in Section 4.2. To connect both empirical studies, we present in Section 4.3 lessons learned from the questionnaire and expert interviews that will positively impact the development of the features offered by current model versioning systems and give hints how to improve collaborations in software engineering projects.

4.1 Questionnaire

In 2010, we conducted a survey on versioning in software development, with a special focus on model versioning. The questionnaire is depicted in Appendix A. The data gained by the online survey was analyzed with the help of the tool "R"[1]. MS Excel[2] was used for the visualization. Furthermore, χ^2 tests have been done in "R" to show if the answers of two questions statistically significantly influence each other. In particular, a χ^2 test proves whether two properties are independent of each other with a certain probability of error (α).

4.1.1 Selection of Participants

In this survey, 90 people participated. The participants were recruited through the help of social networks and public as well as private mailing lists. Our project partner, who is selling a widely adopted modeling tool, sent out invitations for participating in this survey. As a result, we could gather participants who were working in the area of software development and who probably had knowledge in modeling. Furthermore, we sent out invitations via mailing lists of research communities in the area of computer science, especially software and model engineering.

4.1.2 Elaboration of Questionnaire

The questions in the survey aimed at how versioning systems are employed and which patterns and behaviors are applied when developing software in teams. Since we assumed that MDSE is not so widely adopted in practice yet to find enough participants, we formulated the questions in such a manner that we could expect a broad spectrum of feedback. Thus, we basically did not distinguish which artifact is set under version control. To find out if and how models are treated in the versioning process, we conducted expert interviews as presented in Section 4.2.

On the one hand, we elaborated questions, which aimed on meta information on the participant's projects. We asked, which roles did they play, how many people participated, was the team geographically distributed, etc. On the other hand, we elaborated questions, which tried to find out the used versioning strategy, the versioning habits and processes, and how occurred conflicts are handled. The results of this survey are presented in detail in the following.

[1] `http://www.r-project.org`
[2] `http://office.microsoft.com/de-at/excel`

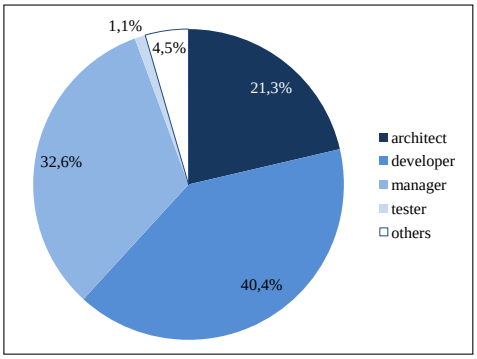

Figure 4.1: Roles of Survey Participants

4.1.3 Results

Meta Information on Projects

Overall, 40% of all participants were software developers, 33% managers, 21% software architects, and only 1% stated to work as tester as depicted in Figure 4.1. The respondents mainly work within small and middle-sized teams. More than half (57%) work in a team with up to 20 persons and 37% work with 5 or less persons. 6% of all persons work with large teams of more than 20 or even 100 participants. Approximately half of all respondents work with colleagues who are situated in the same building. Another 15% work in the same town, 22% in the same time zone and 11% are distributed all over the world. If we combine the geographical distribution with the team size, we can see that 50% of the survey participants work with their team in the same building and with up to 20 persons or less. The small group of interviewees working within big teams is distributed all over the world. Figure 4.2 depicts the combination of the team size and geographical distribution. Since the number of those people, who said they were distributed all over the world or working in the same time zone, is too small for statistical tests, we only distinguish in this chapter whether the team works inside the same building (53%) or if it is distributed (47%).

Versioning Strategy

As already discussed in the previous chapters, basically two different versioning strategies exist. Pessimistic versioning locks a specific artifact when it is changed, whereas optimistic versioning allows for concurrent development on the same artifact. According to the survey results, 79% of all interviewees use the optimistic versioning paradigm when developing software and 18% set their software artifacts under pessimistic version control. Only 3% of all interviewees do not use any version control system. Furthermore, 73% of all participants are convinced that locking of one resource causes delays in the project progress. Furthermore, Figure 4.3 depicts the relation between geographical distribution of a development team and the used versioning strategy within this team. 87.5% of the teams which are developing distributed use optimistic versioning and only 12.5% use pessimistic versioning. When working in the same building

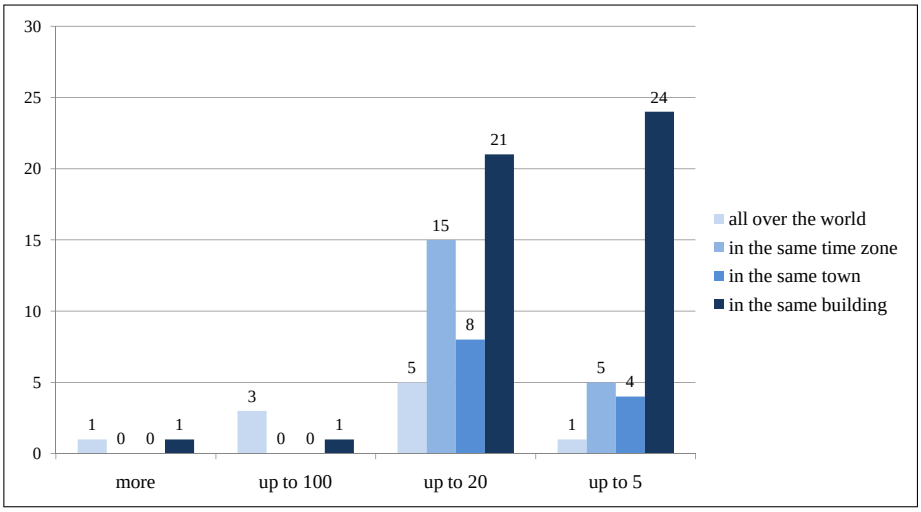

Figure 4.2: Geographical Distribution and Team Size

76.1% are versioning in an optimistic way and 23.9% in a pessimistic way. The number of those people, who are versioning optimistically or pessimistically does not significantly differ between the different kinds of geographical distribution (χ^2=1.84; α=0.05). However, the team

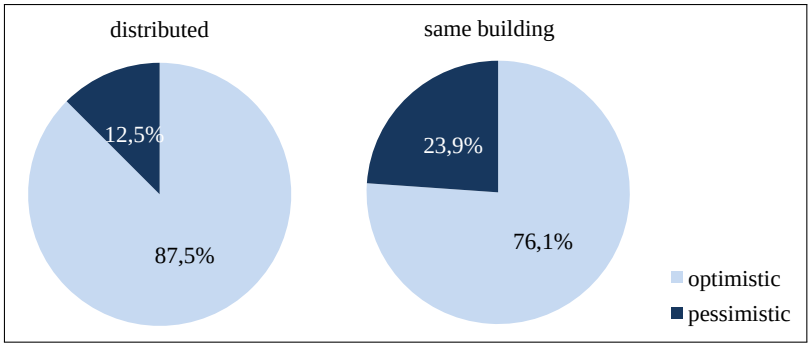

Figure 4.3: Geographical Distribution and Used Versioning Strategy

size significantly influence the applied versioning strategy (χ^2=12.016; α=0.05). In teams with up to 5 persons the number of those who apply pessimistic versioning is remarkably high. 35% of small teams use pessimistic versioning, whereas in larger teams this strategy is only used in 1-2% of cases. However, as mentioned above, overall only 18% used pessimistic versioning and, thus, the level of use in total is not very high.

Two main reasons exist as to why a versioning system is used: 58% of all respondents state that they use a version control system for concurrent development of multiple developers on one artifact. Since version control systems do not discard old versions of an artifact, it is possible to trace the changes between these versions, and this is given as the main reason for 34% of all respondents. Finally, 5% of respondents use a VCS to enable explicit branches of the

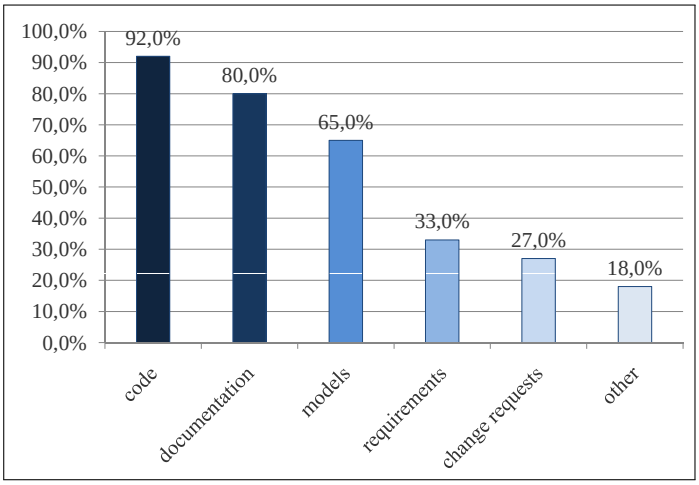

Figure 4.4: Artifacts under Version Control

developed software. With the help of branches different versions of the software are available in the repository in parallel. As depicted in Figure 4.4, over 90% of the survey participants are versioning source code and 80% are also versioning documentation. 65% use versioning facilities for software models and 33% state that they also set requirement specifications under version control. Furthermore, 27% are also versioning change requests and 18% other artifacts.

Versioning Process

In the following, answers to questions about the versioning processes and habits of the survey participants are presented. Here we were looking to see if the team size and the geographical distribution within a team influence these processes. We found that the team size does not significantly influence the versioning processes no matter whether the team consists of more or less than 5 people. In the following we discuss the relation between the geographical distribution and the versioning processes. 62% of all survey participants do not have a standard process, either on an enterprise level or on a project level, as to how versioning has to be performed. 36% do have a standard process and 2% plan to develop such a standard process. Interestingly though, as depicted in Figure 4.5, the teams which are not situated in the same building use a standard process in 75% of all cases. Within teams working together in the same building, this amount is considerably smaller (53%). The usage of a predefined process differs significantly on the basis of the geographical distribution of the team (χ^2=4.756; α=0.05). When asked about authorization policies, 71% state that they do not have different levels of authorization for changing artifacts within a project. That means that they do not distinguish in advance whether developers may change or only read a resource. Another 23% use different levels of authorization within the development process and 6% plan to implement them. In this case, the geographical distribution does not influence the decision about whether different levels of authorization exist or not (χ^2=1.747; α=0.05).

For 34% of the interviewees a changed artifact should be committed into the central repos-

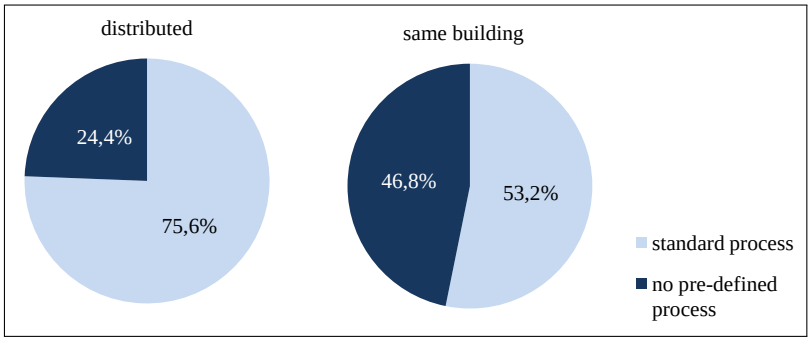

Figure 4.5: Geographical Distribution and Standard Process

itory whenever certain functionality is finished. Others (27%) believe that a developer should commit her changes when she leaves the workplace and 26% think that it should not be regulated at all. 13% guess that a commit should be performed after testing. In relation to the geographical distribution of the respective team members, most (39%) of the teams working in the same building do not use a systematic approach specifying when a commit is performed (cf. Figure 4.6). In contrast, most of the distributed teams have processes where changes have to pass certain tests or certain functionality has to be finished. The answers of both questions are not stochastically independent (χ^2=19.963; α=0.05).

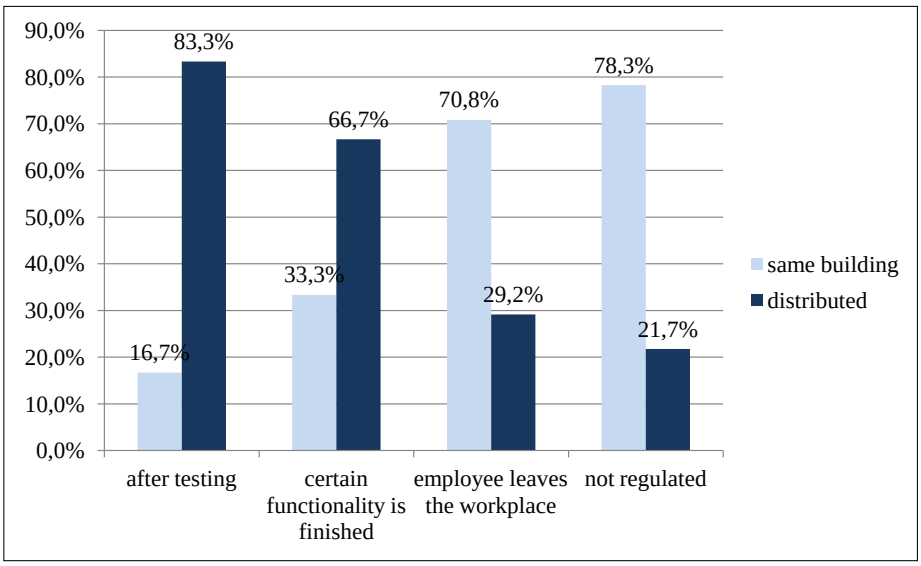

Figure 4.6: Geographical Distribution and Commit Cycles

Furthermore, we asked for criteria based by which a changed version is finally accepted. The survey participants were allowed to give more than one answer. In most cases (40%) the acceptance of changes is not based on any criteria. In 35% of cases the confirmation by a person in charge is necessary and in 20% of cases unit tests have to be successfully passed. As depicted

in Figure 4.7, if no criteria exist, it can be deduced that the team is mostly (74%) working in the same building and, in contrast, teams which are distributed mostly define one person who may accept the changes (78%). It exists a highly significant dependence between geographical distribution within team and the change acceptance creteria (χ^2=19.8; α=0.05).

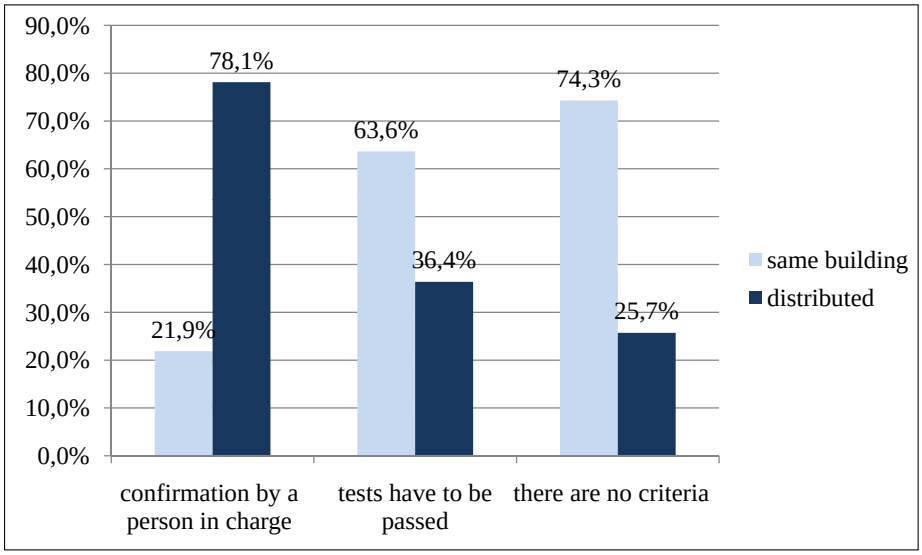

Figure 4.7: Geographical Distribution and Change Acceptance Criteria

According to the survey results, 56% of the interviewees state that big changes like refactorings, which change the structure of the system, may be performed by every developer in the team. In contrast, in 44% of all cases only one person in charge may perform such big changes. As depicted in Figure 4.8, we can see that teams which are not situated in the same building choose a more systematic approach: 79.5% of these distributed teams have a person in charge who is allowed to perform big changes. In contrast, only 21% of those teams working in the same building assign one person who is responsible for performing big changes like refactorings. A highly stochastic dependence exists whether the team is distributed or not (χ^2=29.055; α=0.05).

Coping with Conflicts

When asked about effort related to conflict resolution, 75% of all interviewees stated that the most effort occurred when using a versioning system. 12.5% state that "updates" of the local working copies cause the most effort. Other answers like "conflict detection" or "unlocking artifacts" are only given in 1-3% in each case.

As depicted in Figure 4.9, in 47% of all cases, the developer who is committing her changes is alone responsible to resolve the conflicts. 31% state that all developers who caused the conflicts by parallel modifications have to resolve them. In 11% of all cases a person in charge resolves the conflicts and in 9% the whole team resolves them. When resolving conflicts, most respondents (74%) state that quality is the most important criteria. 24% state that effort should

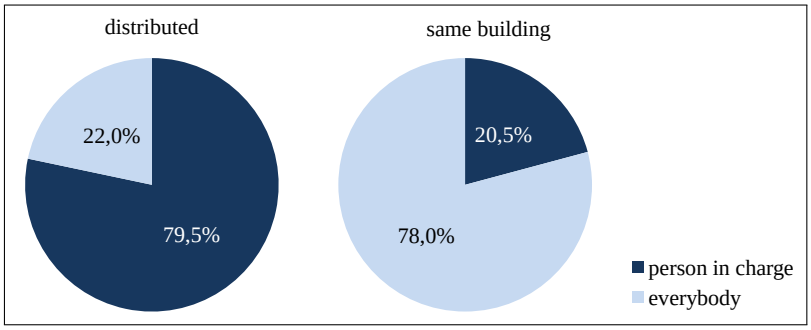

Figure 4.8: Geographical Distribution and Permission for Refactorings

be as minimal as possible. Only 2% were concerned that the conflicts be resolved in the shortest time.

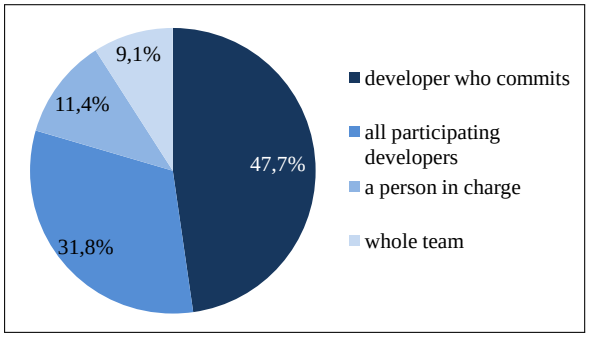

Figure 4.9: Responsibilities for Conflict Resolution

Finally, 61% of the interviewees consider visual conflict resolution support as the most important requirement for a model versioning system whether they had already used it or not (cf. Figure 4.10). For 25% the conflict detection phase is most important and 14% consider customizable automatic conflict resolution support as most important requirement.

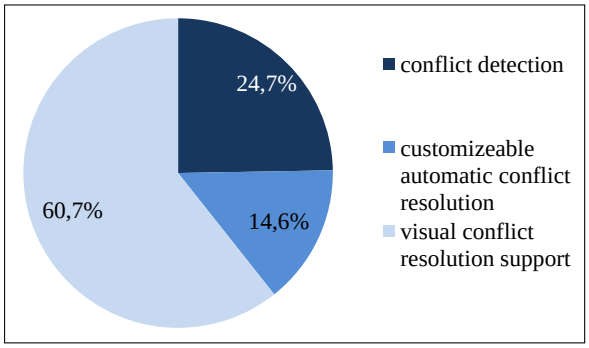

Figure 4.10: Most Important Requirements

Overall Findings

The results of the survey show that versioning is an integral component of the developers' work since nearly 100% of all respondents state to use it. It indicated that, in general, they want to develop their artifacts in an optimistic way independently of the other team members to avoid unnecessary delays in the development process. Most survey participants use textual versioning systems like SVN, which work well for textual artifacts like code. However, as indicated by the numbers of respondents also using models under version control, there is evidence that models have gained momentum in developing IT systems.

Conflict resolution causes the most effort in the development process. When merging conflicting versions the main objective is that the result should by of high quality. High quality could be reached through adequate versioning systems and well-defined collaboration patterns and processes that support the developers to find a consolidated model. Especially when the development is distributed (not in the same building) well-defined and systematic processes are important.

To get more insight into the versioning processes and habits in real-world projects, we have additionally interviewed experts of different domains. In the first place, we tried to find out, what the versioning strategies look like, when models are under version control. The most important answers/findings of these interviews are presented in the following section.

4.2 Expert Interviews

In 2010, we have conducted 10 expert interviews to identify the state of the art when developing models collaboratively supported by any SCM tool. Our interviewees gave us insights how concurrent development works in different areas. We succeeded in finding representative people of different domains working in different areas of software development.

To understand the experience of real software practices from their perspective, we decided to take a qualitative approach and to conduct semi-structured narrative interviews [Wen01]. In preparation, we identified a number of broad themes that we hoped to cover with the participants and wrote these up as an interview guide. These included: a discussion of their role, a presentation of their projects, stories of their collaborative work, reviews of their versioning habits, etc. The actual interviews were then conducted as open-ended conversations, enabling us to follow leads as brought up by the interviewees, with the guide playing a background shaping role to how we engaged in the conversation. The duration of the interviews varied between half an hour and two hours and took place usually in the offices of the interview partners. The interviews were audio recorded and later transcribed. Since most of the interviews were conducted in German, the quotes presented below were translated to English as exactly as possible. A qualitative thematic analysis was conducted with the transcripts to draw out emergent themes. In the following, we shortly describe the roles of our interview partners and how their working environments look like before going on to discuss these themes.

4.2.1 Selection of Participants

We recruited our participants with the help of our academic and industrial network. The set of interview partners included software developers working in big companies and in very small institutions, requirement engineers, and project managers. They have one thing in common: in their companies or institutions software is developed in a team.

The roles of our interview partners vary from IT managers in big international concerns such as IBM or SAP to research tool developers working in a small team. All interviewees are working in the area of software development and use models either for documentation purposes, for defining requirements or for generating code. In the interviews we have discussed the experiences of the interview partners gained in one or two concrete projects. In Table 4.1, an overview of the background of the interview partners is presented. As depicted in this table, our interview partners are assigned to different letters (from A to J). In the second column the role of the interviewee s/he plays in the company or project is depicted. The third column notes the size of the project team(s) and the forth column the geographical distribution of this team on which the interview is based. To better understand or interpret the responses, the fifth column depicts how the interviewee usually uses models. We distinguish—according to [Fow03]—between the usage as sketches (Sk), as blueprints (Bl), or as programming language (Pr), whereby it has to be noted, that the borders between models used as sketches and models used as blueprints are blurry.

	Role	**Team Size**	**Geo. Distribution**	**Usage of Models**
A	Chief Development Architect	Up to 100	All over the world	Sk, Bl, Pr
B	Software Engineer	(a) Up to 10, (b) more than 20	(a) Same building, (b) all over the world	Sk, Bl Sk, Bl
C	Developer	More than 20	Same building	Sk, Bl
D	Scientific Project Manager (Requirement Engineering)	Up to 10	Same room	Sk
E	Researcher and Developer	Up to 5	Same town	Sk, Bl, Pr
F	Researcher and Developer	Up to 10	Same town	Sk,Bl
G	Chief Information Officer (CIO)	Up to 100	All over the world	Sk, Bl, Pr
H	Scientific Project Manager (Requirement Engineering)	Up to 10	Same room	Sk
I	Independent Consultant (Open Source Software)	More than 20	All over the world	Sk, Bl
J	Developer (Freelancer)	Up to 3	Same town	Sk, Bl, Pr

"Sk" ... as sketches
"Bl" ... as blueprints
"Pr" ... as programming language

Table 4.1: Overview of the Interview Partners

Interview partner A is chief development architect of a big, international concern and he reported of large projects with development teams spread all over the world, for which he was responsible in the past.

Interviewee B is working as a software engineer and has participated in two very different projects. One project was also a large international project in which different developer teams were spread all over the world and the other project was, in contrast, a local, national project, in which approximately 10 developers were involved.

The next interviewee (C) worked as a developer in big international company and he reported his experiences of middle-sized projects of which all developers were situated in the same building.

Interview partner D is a well-known expert in the area of requirement engineering, who gave us deep insights in the collaborative development of requirement specifications. Usually, up to 10 people participated in his projects and all team members were situated in the same room.

Interviewee F is developing tools used for model-driven engineering. He reported about his projects at his university, where the challenge was to coordinate his students when supporting him in developing further his tools. They worked on new features in parallel and independently of each other and, thus, his experiences in optimistic versioning were also very interesting and important, although they do not stem from an industrial context.

A quite similar situation appropriate for interview partner E: fairly small group of developers situated in the same building. In contrast to F, he mainly worked in his project alone. However, he also got support from students leading to big challenges when working collaboratively.

Interview partner G is Chief Information Officer (CIO) at a national company with one of the largest IT departments. He is responsible for the development of different IT solutions, which are developed by different teams situated all over the world. His management view on the collaborative development of software artifacts gave us also interesting inputs from another perspective.

Interview partner H is also working in the area of requirement engineering like interview partner D. H is researcher and is developing group support systems to support the collaborative development of requirements.

Interviewee I reported in the interview about his numerous open source projects. His experiences gave us deep insights into the collaboration processes in the open source community. There, software is also developed within a team with more than 20 participants spread all over the world. Especially in the open source community, optimistic versioning place a crucial role.

Finally, we have also interviewed a software developer (J) working as freelancer. For his projects, he usually managed to get support from one or two other developers in later phases of his projects. This challenging situation gave as also very absorbing inputs.

This variety of different domains, team sizes, geographical distribution of the team members, and different kinds of development processes leads to interesting findings that we describe here. These let us derive lessons learned of best practices for model versioning.

4.2.2 Elaboration of Questions

Based on the questionnaire, we elaborated questions for our interview partners. Since our interviews varied due to the different roles of the interview partners, the elaborated questions represented only a guideline for the open-ended conversations. The overall goal of the questions was to find the strategies and habits when setting models under version control. Of course, the interviews also cover experiences with other software artifacts.

4.2.3 Results

In the following, the most important results of the interview are presented. After presenting the big picture, we distinguish between the versioning processes in big and in small teams. This distinction is only used to better group the individual statements of the interviewees. Special focus is set on the resolution of conflicts. Finally, we present the main requirements for versioning of our interview partner. As code-centric development is still the standard way to develop software, most experiences and answers are closely related to code. Nevertheless, the wish list for model versioning is long. In fact, it can be seen as indispensable tool for putting MDSE into practice.

Big Picture

Most of the interviewees have worked or are working with a traditional versioning system like SVN. Most of the interviewees put every software artifact, including models under optimistic version control. However, three interviewees (F, H, and I) mentioned in the interview, that they lock models when changing them.

Developer F stated: *"In general, tool support for developing models is nowadays insufficient especially when modifying them in parallel."* Moreover, Consultant I stated: *"Versioning models with SVN is everything but not recommendable, because merging two versions is far too complex."* And IT Manager G is convinced: *"...that optimistic model versioning can be used in industry in 10 years."*

In addition, in all projects in which B and F have been involved, parallel modifications of the same artifact were allowed. However, also both developers complain about the tool support when merging the parallel modifications. For instance B claims: *"Merging branches or parallel versions costs at least two working days with state-of-the-art tools—no matter if we have merged models or code."*

When developing requirements, asynchronous development is not possible. Both, D and H stated that requirements can only be defined in workshops where all participants develop requirements collaboratively and synchronously. D states: *"I only see disadvantages when requirements are not defined synchronously within the team."* The requirement engineers that we have interviewed have their own group support system, which allow finding easily a consolidated model reflecting the requirements. Both are convinced that only in this way misunderstandings can be eliminated in time. H states: *"Misunderstandings or conflicts have to be eliminated as soon as possible in a software development project. The earlier the better. [..] In later project phases it often leads to expansive delays."*

In the following, we present the versioning processes and habits which have been described by our interview partners according to their personal experiences. We distinguish between big and small development teams. According to the table presented above, we can distinguish between small teams with less than 10 developers and big teams with more than 10 and up to 100 developers. To better group the results of the interviews, we had to discard the classification of the team size used in Section 4.1.

Versioning Processes in Big Teams

Manager G has successfully managed very large IT projects in central Europe, in which up to 100 people have been involved. According to the interview, parallel development only works through the exact assignment of packages and modules to dedicated developers and it is of huge significance that the interfaces between all participants are well defined to avoid conflicts: *"If conflicts arise, we have a big problem—we have done something wrong. [..] We then have to organize workshops where the development teams have to discuss the occurred problems."* Since the developers are usually distributed in different countries and also in different time zones these workshops are also conducted via virtual online meetings. Also for Manager I, who has a lot experience in developing open source software, the avoidance of conflicts when developing in big teams, which can be distributed all over the world, is important: *"I would not go so far to say that every artifact has to be locked when changing it. However, it is important to separate exactly the responsibilities in advance."* For him it is also very useful to allow branches in the development process, but he knows that *"merging of two branches is very difficult and error-prone, because it happens very often that many conflicts arise."*

Developer B has also worked in a large project in which the developers were distributed all over the world all artifacts (also documentation) have been put under optimistic version control. The developers were usually allowed to modify all artifacts, but critical artifacts were locked when changing them. After this project experience he recommends to *"...commit or update changes to the central repository in as short as possible cycles to avoid the resolution of really big conflicts. [..] Due to different time zones of the responsible developers it was very hard to communicate the different intentions"*. In contrast to manager I, he cannot recommend branching, because, as already mentioned above, it would cost too much time merging the different branches. According to his experiences, it usually took at least one whole working day to include all changes in the main branch.

Versioning Processes in Small Teams

All interviewees, who participated in projects with not more than 10 developers, reported that all artifacts have been put under optimistic version control.

Developers B, C, and F stated in the interviews that a useful separation of the artifacts under development and their assignment to the dedicated developers are of significant importance. Despite the small number of participants they try to avoid conflicts due to error-prone and time-consuming resolution of these conflicts. Developer C states: *"In our projects, we had less conflicts, because we have split up all the packages in such a way that it was nearly impossible to change the same class in parallel"*. The participants used many different strategies for separating code or models. For instance, Developer F uses the following strategy: *"Our students support us in implementing new features of our MDE tools. Thus, we separate our artifacts according to these features trying to avoid conflicts."* Developer B reported that in his projects they have *"...tried to hold the files as small as possible..."*, also with the objective to have as less conflicts as possible.

We have also asked the interviewees how often a commit of the changes should be performed. Most of the interviewees mentioned that they do not have any when developing in small

teams. *"We have committed our changes only after big changes. [..] I don't want to say that this is good or bad, but in all projects it was common."* Developer E has mostly developed his tools alone and, thus, no conflicts have arisen. He has used versioning systems for traceability reasons. He said: *"I tried to have as small commits as possible, because in my opinion, it is most important that the code in the repository compiles."*

Conflict Resolution

When it comes to a conflict due to concurrent changes, conflicts have to be resolved before a merged version can be produced. First of all, Manager I and Developer J are convinced that the automatic resolution of the occurred conflicts is not a good strategy to have one consolidated version at the end. Developer J states: *"Automation would lead to even more problems. [..] It is not ensured that the result is correct."* This points to the importance of issues such as trust and control over the quality of the process. In the projects, which were managed by Developer F or Manager I, one person in charge was responsible to review the changes of the other developers and to resolve occurred conflict. Developer F states: *"In our small university projects I was responsible to resolve the conflicts, because I wanted to make sure that the merged version doesn't include several errors."* Conflict resolution is not always performed by one single person. In the large projects of architect A he has *"assigned developers to specific problems. Not only the two developers who were responsible for the conflict. [..] They have to discuss the occurred conflicts."* Developer B also reported that they resolved conflicts collaboratively: *"We have chatted or even organized workshops when serious conflicts occurred."*

When defining requirements H and D reported in the interview that conflicting requirement specifications are always resolved within the team. In addition, a facilitator mediates this conflict resolution session to have consolidated requirements at the end.

The Ideal Versioning Scenario

Finally, we asked all interview partners about what their ideal versioning system would look like and which features they miss in current systems. As already discussed above, all interview partners agree that, in general, versioning support for models is insufficient to use it in practice. In addition, according to the experiences all interview partners had gained in the past, we asked, whether they would change the versioning processes and habits in future projects. In the following, the most important and interesting statements are discussed.

First of all, for Manager D the most important requirement a model versioning system should fulfill is *"that it is a simple, easy to use tool. Nobody would use it if it is not intuitive and really simple."* Models can be expressed using their abstract syntax or concrete syntax. The abstract syntax of a model is defined by so-called metamodels [KÖ06] and the concrete syntax represents the notation - graphical or textual - of the modeling language. Concerning the conflict resolution in current model versioning systems, Chief Development Architect A criticizes the way in which the separation of the concrete and abstract syntax impairs the usability of the used versioning system: *"When modeling in the concrete syntax, I want to resolve occurred conflicts also in the concrete syntax. [..] There are many open research challenges. For instance, how to visualize conflicts of the domain model if they are not available in the concrete syntax?"* Some

language concepts, which have a particular representation in the concrete syntax, are not even explicitly represented in the metamodel. Instead, these concepts are hidden in the metamodel and may only be derived by using specific combinations of attribute values and links between objects [KKK+06]. Moreover, for Manager I and Developer C, it also important that not only the conflicts but also the changes can be visualized in a dedicated way. Changes, conflicts and how these conflicts are resolved should be represented explicitly enabling *"better support to see the evolution of a model. [..] Browsing between the different versions of a model would be very beneficial"*, claims Manager I.

In traditional versioning systems, the developer who checks in later is left alone with the error-prone task of resolving conflicts. Thus, Architect A wants to *"assign people to specific conflicts, who are together responsible to resolve them."*

For Developer B, it would be a beneficial that versioning systems allow for automatic commits and updates to reduce the amount of conflicts when developing in parallel. *"[..] But that would only work when the versioned artifacts are well distributed."* The possibility of a useful separation or modularization of a model is also a requirement of Developer E to a model versioning system. He states: *"When we could divide a model into different modules, conflicts could be better avoided. I think that there are some approaches in different research areas that could be implemented for models."* In addition, his ideal versioning system would also consider the fact that different artifacts can depend on each other: *"When generating code out of models, and I change the model, it would be great if the dedicated code is synchronized by the versioning system."* Furthermore, developer E would like to have the integration of a bug report system especially adapted for models.

Finally, Developer E and J would like to have a versioning system, with which occurred conflicts can be resolved later on. E states: *"I often just want to commit my changes and do not want to resolve the conflicts immediately. [..] My work is saved and on the next working day I can discuss the conflicts with the other developer."*

Overall Findings

Confirming the survey results, optimistic versioning of code is used within every software engineering project that our participants discussed. For models, the situation is different: tool support is more unsatisfactory. Thus, models are often locked when changing them. No matter which artifacts are under version control, in general, developers try to avoid conflicts due to missing adequate tool support when resolving these conflicts. However, not only technical issues such as precise conflict detection have to be tackled. Aspects of collaborative work cannot be ignored when trying to resolve conflicts. Especially from the area of requirement engineering, we can learn that conflicts are not always negative results of collaboration. Several interview partners claimed that conflicts have to be discussed and have to be resolved collaboratively. Furthermore, it can be deduced, that the constant evolution of the software artifacts have to be more comprehensible. Change-awareness is crucial when developing in parallel and independently of each other.

Due to the presented interview and survey results, we present in the following lessons learned to improve collaboration when using a versioning system dedicated to models.

4.3 Lessons Learned

In this chapter, we have been concerned to understand the realities of collaborative development supported with versioning systems. To address this we have conducted both a survey and in-depth interviews. Across both of these we see common themes emerging which we present here as lessons learned, along with a discussion of the implications of these lessons for future research directions if appropriate.

Model Versioning Tools are needed. 65% of our survey respondents set also models under version control. All respondents use traditional versioning systems such as SVN, which makes collaboration worse because models are usually serialized and stored as XMI in versioning systems. Traditional VCSs perform their comparison and conflict detection on the XMI serialization and, thus, detected conflicts have to be resolved also on XMI level. XMI is not intended to be edited by humans and, moreover, models are usually developed graphically or at least with a tree-based editor. Through these barriers developers usually refrain from modeling in parallel and independent of each other. Instead, they often lock the complete models to avoid conflicts.

Stick to known notation. Current text-based versioning systems match with textual programming languages in the sense that the familiar notation of programmers used in programming environments is also supported by the versioning systems when visualizing changes and conflicts. In our interviews, a recurring practical need has been mentioned, namely to have versioning support within the concrete syntax of the models defined in graphical modeling languages. Of course, this result is not unexpected, because modelers are usually familiar with the concrete graphical syntax, but not with computer internal representations like XMI or abstract syntax graphs. In addition, 60% of the survey participants find visual conflict resolution as most important requirement. Some dedicated approaches have been proposed for visualizing differences of models [MGH05,OWK03]. They construct a dedicated view using the concrete syntax, which combines and highlights changes of both models using coloring techniques. However, these approaches require special extensions of the modeling editors being a barrier for adoption. In [BKL+11b], the visualization of changes and conflicts with the help of UML Profiles[3] is presented which do not require for heavy-weight editor extensions. However, none of the respondents and none of the interview partners are using some dedicated versioning tools for models due to the dominance of text-based versioning systems which are well-known by the developers.

Division of work as key for parallel development. Most interview partners stated that, in general, they try to avoid conflicts, but do not want to lock the artifacts. According to the survey, most participants claim that locking of artifacts lead to delays in the project. Thus, appropriate mechanisms are necessary which enable an efficient division of work. In particular, several interview partners pointed out that the exclusive access to an artifact is realized by establishing clear structures and well-defined interfaces. Therefore, in the requirement phase of a project,

[3] `http://www.omg.org/technology/documents/profile_catalog.htm`

where the goal is to find the structure for the system to implement, special partitioning techniques and design approaches are needed for models. Whereas for object-oriented programming languages, structuring the software in packages and classes is a well-established design technique [Dij97, Par72], in domain-specific languages such structuring mechanism might not be available.

The more distributed teams are, the more discipline is needed. The results of the questionnaire clearly show that teams which are geographically distributed, i.e., not in the same building, work in a more systematic and disciplined way concerning their versioning habits and processes. Predefined collaboration patterns exist that orchestrate the versioning behavior of the team members. In most cases, a commit into the central repository is only allowed when the changes pass predefined tests or certain functionality is finished. In addition, in 60% of all cases these changes need to be confirmed by a person in charge. Furthermore, big changes like refactorings may not be performed by every team member. Although optimistic versioning is used, conflicts have to be avoided by a good separation of responsibilities and well defined interfaces between them, because conflict resolution is too time-consuming especially when the teams are distributed all over the world.

Commit cycles are kept short but systematic. What we also can learn from our interview partners is that commit cycles should be as short as possible to avoid the resolution of a bunch of conflicts at once. One interviewee would actually like to configure the VCSs to enable automatic updates and commits. However, most interview partners agree that a systematic approach is necessary: tests have to be passed or a concrete feature has been completely finished. In addition, the survey results show that most respondents commit after a certain functionality is finished. Concerning the need for frequent commits, one important aspect is that the developer has the possibility to switch to different development states. However, propagating the changes directly to the global repository seems to be problematic, because inconsistent states may be checked out in parallel. Therefore, distributed versioning systems are gaining some popularity which allow to have a local repository for each developer. However, only 3% of our respondents use distributed versioning systems, the rest is using SVN and CVS.

Branching is expensive. According to two interview partners branching is often used when complete features of a system are developed independently of each other. For example, at the university of one interview partner different students support him to further develop his tool. They have to work on an own branch to avoid conflicts in the main branch. However, the price of branching is time-consuming merging of two branches. The other interview partner also reported similar experiences in the open source community. It usually took at least one whole working day to include all changes correctly into the main branch and to ensure that the system works correctly afterwards.

Define requirements synchronously. From our interviews with experts in the field of requirement engineering, we can learn that defining requirements has to be done together and synchronously within the team. It would be too expensive later on in the project phase, to resolve

conflicts due to misinterpretations of requirements. This is in accordance with studies concentrating on the costs of bugs which are introduced in different phases of the software development lifecycle. Therefore, it is not surprising that dedicated synchronous development environments have been proposed. For example, [DH07] also propose a synchronous environment for software development. Here the hypothesis is that conflicts can be avoided through awareness, as is similarly provided by the event notification service associated with CVS use in [FMP06]. Consider for example that when one modeler is working on a certain model fragment and another modeler is also opening this model fragment, both modelers are informed that they are concurrently working on a certain part of the model. Then they have the possibility to consolidate their planned changes with the help of collaborative tool features such as chats.

Critical cross-cutting changes require locking. According to the survey most respondents are versioning their artifacts in an optimistic way. However, interview partners claimed that certain artifacts are locked when performing critical cross-cutting changes like refactorings to avoid several conflicts, which would be hard to resolve. However, the question arises which parts or files have to be locked. Since most survey respondents claim that locking leads to delays in the projects, the design of the artifacts enabling reasonable locking plays an important role. At this point, we want to refer to the above mentioned challenge concerning the separation of artifacts as key for parallel work and to the systematic versioning in case of distributed teams.

Conflict as first-class citizens. When different developers work in parallel and independently, it might easily happen that the modifications are not comprehensible any more. In traditional VCSs it is nearly impossible to get to know in which way a conflict is resolved, because this provenance information is missing. According to the interviews, it is often very important to see which part of an artifact is modified by which developer and in which way it is changed. Also when a conflict occurs it is beneficial to know how a conflict was resolved and who was responsible for the resolution decision. This information about the resolution process should be explicitly available in versioning systems. When making the participants aware of changes, conflicts as well as the resolution process, the evolution of a system is becoming more comprehensible. Similar to the handling of bugs in nowadays software development, the representation and persistence of conflicts between various developers as well as the definition of lifecycles for conflicts is of paramount importance.

Conflicts are resolved collaboratively. For the majority of the survey participants quality is the most important property when resolving conflicts. This is also confirmed by several interview partners. The developer who is committing her changes later than the other developers is responsible to resolve the occurred conflicts. In traditional VCSs, the developer who commits her changes bears the full responsibility when resolving conflicts, which may lead to undesired results. The developer does not always fully grasp the intentions behind the changes of the other developers. Thus, it is very difficult or impossible to finally have a consolidated version of the model reflecting the intentions of all developers. In the interviews, it is often mentioned, that conflict resolution has to be done by the responsible developers, not by a single person, who

has checked in her changes. Several interview partners reported that they meet each other for face-to-face meetings or they organize workshops in which they discuss the occurred conflicts.

Allow for conflict tolerance. When talking about the ideal versioning system with our interview partners, it is requested to commit the changes without worrying about potential conflicts. In this context, it is desirable to keep all or at least many of the model changes, even if they are conflicting. The interview partners want to resolve the conflict at a later point in time and, in addition, if all changes are committed and the conflicts are marked it forms a good basis for discussion. Thus, a versioning system for tolerating conflicts should allow to explicitly representing and persisting conflicts as well as to allow tracking conflicts to resolve them later on. A first idea of tolerating conflicts in the context of model versioning is proposed by [BLS+10b].

CHAPTER 5

Turning Collaborations into Annotations

Modeling activities express the personal view of the modeler on the system under development and the fact that each team member perceives the world or the system differently, is directly reflected by the models. Especially in early phases of the software development cycle, when the team did not establish a consolidated view on the system, the subjective understanding and the intention of the different team members might diverge. According to the survey presented in Section 4.3, this is of crucial importance when defining requirements to avoid misinterpretations at the beginning of a software development project.

One challenge within the modeling process is to create a common knowledge base. Tool support is needed allowing the creation of a common representation of individual assumptions within the team in order to avoid misunderstandings. As we have learned from the survey presented in the chapter before, models are one of the most used artifact in software design, and, thus, tools for making the developers aware about conflicting modeling activities might be valuable in order to find a common solution by learning about the reasons that lead to the conflicts (cf. [RKdV08b]).

The traditional tool support for collaborative software development usually follows the paradigm of either avoiding conflicts or of resolving conflicts as soon as possible. According to the survey results, project participants try to avoid conflicts in general. Conflict avoidance is for example realized by the means of *pessimistic version control systems* which lock artifacts for exclusive modification by exactly one developer. However, according to the survey, locking of artifacts leads to undesired delays in the development project. In contrast, *optimistic versioning approaches* support distributed, parallel team-work. This comes along with the price of conflict resolution when concurrently evolved versions of one model are merged. Figure 5.1(a) shows a typical versioning scenario with conflicting modifications. The modeler Alice creates a new version of a model (V1) and commits this model to the central model repository. The modelers Harry and Sally check out the current version and perform their changes in parallel. Harry deletes an element which is extended by Sally at the same time. Assume that Harry is the first

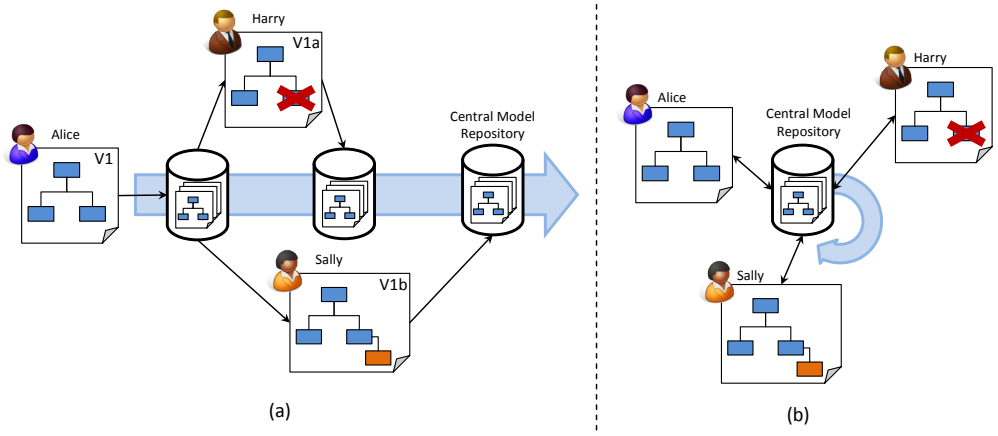

Figure 5.1: (a) Continuous Conflict Resolution and (b) Conflict Tolerance

who finishes his work and he is also the first who checks in his version into the repository. Afterwards, Sally tries the same, but the VCS rejects her version V1b, because her changes are conflicting with Harry's changes. In standard VCS, Sally is responsible for resolving the conflict immediately. Even when assuming that Sally does not want to corrupt Harry's work, still information might get lost, because she might not be aware of all intentions Harry had when doing his work. Also Harry is not aware how Sally is doing the merge. Finally one conflict-free version is checked in with the consequence that some modifications may be inevitably lost as they are removed in an undocumented manner. In contrast to code, models are often used in an informal manner for collecting ideas and discussing design alternatives in brainstorming periods [Fow03]. Models are then used to manage and improve communication among the team members by establishing common domain knowledge. Then this loss of information is especially problematic. According to the survey presented in Section 4.3, the discussion of the conflicts might have helped to eliminate flaws in the design which are probably harder to eliminate at the later point in time of the software development life cycle. In the interviews, it is often mentioned, that conflict resolution has to be done by the responsible developers or by the whole team collaboratively, not by a single person, who has checked in her changes.

In this context, a new paradigm shift as depicted in Figure 5.1(b), namely to tolerate conflicts, is requested explicitly in our survey (cf. Section 4.3), because it is desirable to keep all (or at least many) of the modifications, even if they are conflicting. The conflicts may help to develop a common understanding of the requirements on the new system. To support versioning in early project phases, we propose a versioning system which temporarily tolerates conflicts enabling a creative design process without destroying the model's structure to use common modeling tools for editing. The developers do not have to worry about possible conflicts when checking in as requested by the survey participants. We also learned from the survey that in traditional VCSs it is nearly impossible to get to know in which way a conflict is resolved, because this provenance information is missing. This information about the resolution process should be explicitly available in the versioning system. To make the participants aware of changes, conflicts as well as the resolution process, the evolution of a system is getting more comprehensible. In addition, the

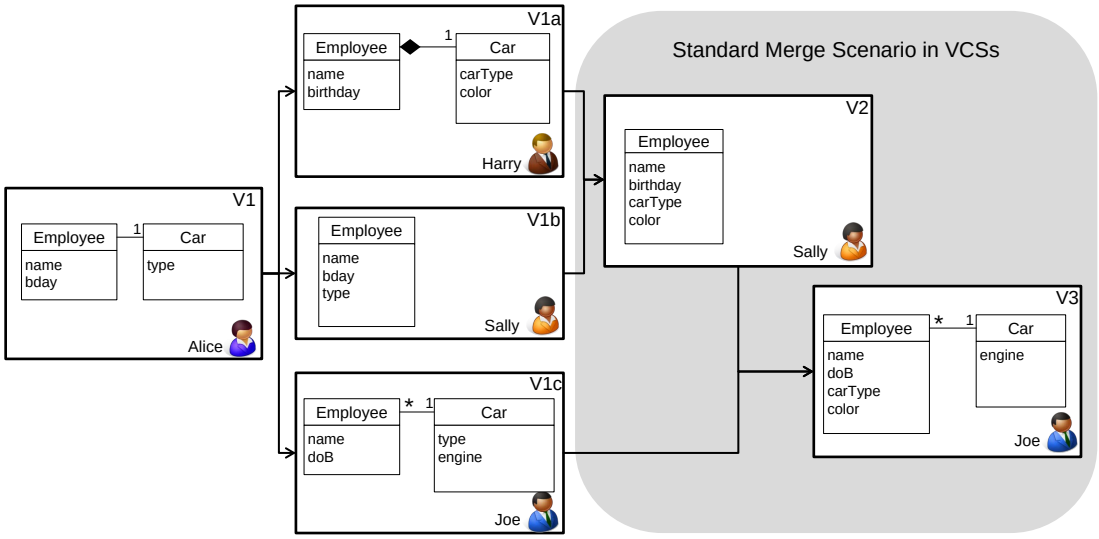

Figure 5.2: Running Example: Merging in Current VCSs

survey shows that merging models should be possible in the concrete syntax of a model defined in a graphical modeling language. Thus, we present a system with which conflicts are visualized on top of the concrete syntax of a model. The model is enriched with meta information about the performed changes and resolutions.

5.1 Running Example

The example depicted in Figure 5.2 describes a merge scenario which demonstrates typical problems when developing models in a distributed team following the standard versioning process. All team members may check-out and modify the model in parallel and independent of each other. And when a conflict arises, the developer who is committing her changes is solely responsible to resolve the conflict immediately.

V1. Alice creates a new model in terms of a UML Class Diagram[1] containing the two classes `Employee` and `Car`. She adds the attributes `name` and `bday`—representing the birthday of a person—to the class `Employee` and `type` to the class `Car`. Finally, she defines an association between these two classes and sets the multiplicity to 1 to define that one employee is assigned to exactly one car. After she has finished, she checks in the new model (called V1) into the central repository of the VCS. Now, the modelers Harry, Sally, and Joe want to continue working on this model and therefore, they check out the current version V1 of the model from the central repository to perform their changes.

V1a. In Harry's opinion, a car is always owned by exactly one employee. Therefore, he modifies the association to express a containment relationship (noted by the black diamond). In addition, he changes the attribute `bday` of the class `Employee` to `birthday` and adds a new

[1] http://www.omg.org/spec/UML/2.2

attribute, namely `color`, to the class `Car`. In addition, he changes the name of the attribute `type` to `carType`.

Resolution. Because Harry is the first to check in, he has no conflicts to resolve. His version is now the latest version within the repository.

V1b. In Sally's opinion one car always belongs to one employee and, thus, she inlines the class `Car` in order to ensure efficient querying. In particular, she moves the attribute `type` from class `Car` to `Employee` and subsequently deletes the class `Car`.

When Sally commits her changes, the VCS rejects her modifications because they are conflicting with the already performed changes of Harry. A so called Delete/Update conflict occurs since the deleted class `Car` has been changed by another modeler (Harry). Consequently, Sally is responsible to resolve this conflict according to the process of standard VCS.

Resolution. In Sally's opinion, the class `Car` is still unnecessary in the model, although Harry has updated this class by adding a new attribute. Therefore, she decides to inline the added attribute of Harry, namely `color`, into the class `Employee` and to still abandon the class `Car`. Version V2 now consists of the class `Employee` containing the attributes added or updated by Alice, Harry, and Sally.

V1c. Joe has also checked out the version of Alice (V1) and performs his changes in parallel to Harry and Sally. He renames the attribute `bday` to `doB` (*d*ate *o*f *b*irth) in the class `Employee` and adds to the class `Car` a new attribute called `engine`. Furthermore, Joe is of the opinion that one car may belong to several employees and therefore, he sets the multiplicity of the association to unbound indicated by the asterisk in the model.

Resolution. Now, Joe tries to check in his version V1c into the central repository but different conflicts are reported by the VCS, because Joe's version is now conflicting with the current version V2 in the repository. Recall that V2 is the version of the model covering Harry's and Sally's modifications consolidated by Sally. Now an Update/Update conflict is reported because both, Harry and Joe, have updated the attribute `bday` in different ways. Joe decides to take his version namely `doB`. In addition, also a Delete/Update conflict is reported, because Joe has updated the class `Car` by adding a new attribute—namely `engine`—to this class. He decides not to delete the class `Car` in contrast to the decision of Sally.

The new version V3 of the model is put into the repository after Joe has resolved the conflicts. Although version V3 is a valid model, it contains several flaws resulting from the conflict resolutions.

The final version of the example model does not reflect all intentions of the participating modelers, because only one single modeler was responsible for the merge of two versions. Also when assuming that all modelers do their best to resolve occurred conflicts, it is particularly difficult for them to understand the motivation behind the changes of the others, especially, if more than two modelers performed changes in parallel.

In particular, the version V3 of the example described above does not reflect the idea of Harry to consider a car as *part of* one employee as expressed by the composition. It does also not reflect the idea of Sally to have only one single class for efficiency reasons. Furthermore, in V3 the attributes `carType` and `color` have become part of the class `Employee` that is obviously undesired assuming that the class `Car` is retained. Over the evolution of the model the information that these attributes are referring to the car get lost since no indications for that

exist anymore.

To summarize, the final version of the model contains several flaws and does not reflect several of the modelers' original intentions. The reason for that undesired situation is mainly that only one modeler was responsible to resolve all conflicts on her/his own. Furthermore, currently only limited tool support is available for such tasks. As an ultimate result, changes, opinions, and potentials for obviously important discussion are destroyed by this versioning paradigm.

5.2 Conflict-Tolerant Merging of Models

The overall goal of our Conflict-tolerant Merge is to incorporate all changes concurrently performed by two modelers into a new version of a model. The merge implements our major premise that no information on conflicts gets lost and that irrespectively of any occurring conflicts the merge process is never interfered by forcing users to immediately resolve conflicts. To realize this goal, we have to focus on overlapping changes (cf. Section 3.2), because they are responsible for hindering the production of a merged version.

5.2.1 Overlapping Changes Revisited

When two changes interfere with each other, i.e., they either result in different models when applied in different orders or one change hinders the other change, then conflicts due to overlapping changes shall be reported. Table 5.1 shows which change combination leads to a conflict, if they are performed on the same model element in parallel. Conflicts are marked with $\times$, and the combination of non-conflicting changes, which might potentially result in an undesired situation, of which the users should be notified, are marked with $\sim$. In the following, we shortly describe the table and in the next section a detailed presentation of the conflicting situations is presented. One important characteristic of overlapping changes is that no unique and complete merged version can be produced without defining priorities for changes.

	Insert	**Delete**	**Update**	**Use**	**Move**
Insert					
Delete			$\times$	$\times$	$\times$
Update		$\times$	$\times$	$\sim$	$\sim$
Use		$\times$	$\sim$		$\sim$
Move		$\times$	$\sim$	$\sim$	$\times$

$\times$ ... *Conflict*

$\sim$ ... *Warning*

Table 5.1: Overlapping Changes

Inserts are never considered as source for conflicts in the scope of this thesis, because when an element is introduced by one modeler, the other modeler cannot apply a contradicting deletion, update, and move on the same element. However, some conflict detection approaches report

insert/insert conflicts, when the "same" model element is introduced by multiple modelers. This is usually the case, when the comparison is not based on artificial identifiers, but on matching heuristics such as string comparisons. In this thesis, we do not consider such situations, because we use only artificial identifiers and a merged model comprising both model elements may be produced anyway.

When two modelers delete the same model element in parallel, no conflict has to be reported, as the intention as well as the result of both changes is the same. All other change combinations applied to the same model element lead to conflicts, e.g., when two modelers update the same feature (such as the name) of a model element, in a different way. In the running example (cf. Figure 5.2) several Update/Update conflicts caused by the parallel modifications of one element, such as of the attribute `bday` in the class `Employee`, occur. In such situations, it is not possible to produce one unique merged version including both changes without extending the modeling language. The same is true for Delete/Update conflicts, e.g., consider our running example where Sally deletes the class `Car` which is modified by Harry and Joe.

If a model element is deleted and, concurrently, a reference value using the same element as target is inserted, a Delete/Use conflict is reported. Consider a model of two classes `A` and `B`. One modeler deletes the class `B` whereas the other modeler concurrently set a reference to the same class. The deleted class `B` is used as a new reference value and, thus, a Delete/Use conflict occurs.

Special kinds of Delete/Update conflicts are Delete/Move conflicts where the update operation is actually a move operation. Of course, when one modeler moves a model element and the other modeler deletes it, a conflict has to be reported. Finally, another source for conflicts is when one element is moved to different containers by two modelers in parallel. For instance, assume that an attribute is moved from class A to class B by modeler 1 and modeler 2 moved the attribute from class A to class C. The container has to be unique for each model element; therefore this situation must result in a Move/Move conflict. In fact, a move operation is a special kind of update operation, but drawing the distinction between these operations give the user more detailed information.

Another special combination of changes exists when an element is moved to a different container and exactly this element is updated in parallel by another modeler. When both versions are merged, no conflict is reported by the system, because both operations can be performed independently of each other. For instance, an element is moved to the new container and, in parallel, it is updated. The same result is produced when the changes are executed in the reverse order. Now one may conclude that these situations are not problematic. However, the modelers should be aware of them, because the new context of the moved element may lead to undesired situations. Although coming from another background, [DA00] define context as "any information that can be used to characterize the situation of an entity. [..]". Furthermore, they define a system as context-aware "if it uses context to provide relevant information [..] to the user, where relevancy depends on the user's task." We also want to make the modelers aware of the new context, i.e., new container, of the moved element when it is changed in parallel by another modeler. For instance, in the running example Sally moves the attribute `type` from the class `Car` to the class `Employee` and deletes the class `Car`. In parallel it is renamed to `carType`. The attribute `carType` is now contained by the class `Employee` and, in addition,

the class `Car` is not deleted in the final version. This example shows that the context of an element is very important to reason about the meaning of an element. Thus, in Table 5.1, $\sim$ is used which means that not a conflict is reported but a Move/Update warning is raised. In such cases the modeler who updates the element should verify if her update makes sense also in the new context after applying the move.

Similar situations occur, when a use operation is performed and, in parallel, the used element is updated or moved. If an element is used by another element and the used element is updated or moved, both operation can be performed, but we report an Use/Update or Use/Move warning to notify the user that an undesired situation might occur which can be checked by the modeler.

5.2.2 Conflict Tolerant Merge Rules

In the following, we present the different types of conflicts and introduce for each conflict type a dedicated rule specifying how a merged model is produced when two or more conflicting operations are at hand. The merged model is produced with the help of annotations and priorities for changes. Thus, these two aspects are presented in detail.

In order to allow for generic definitions of the merge rules, we make use of the UML Object Diagram notation to represent the model fragments. This means that each model element is represented by an object. An object has links to other objects as well as attribute values for possessing simple data values. Please note that there is a specific kind of link called `containment link` which expresses container/containee relationships. In the following figures, such containment links are depicted by a black diamond at the container side. The containment links are especially used in these figures if moves of an element, i.e., changes of the container, are involved. For the sake of clarity, we distinguish between 3-way merge rules and N-way merge rules with N greater than 3 in the following.

3-Way Merge Rules

Update/Update Conflict. As depicted in Figure 5.3, an Update/Update conflict arises if the same feature `f1` of object `o1` is changed in two different ways in parallel. Since both updates cannot be performed when merging without overriding each other, the original value namely `x` of the feature `f1` is preserved in `o1`.

To mark this conflict, the modified object is annotated with an `UpdateUpdate` annotation containing values of both concurrent changes. The annotation includes the involved feature of the object. Finally, meta-information is added to the annotation such as user-related and time-related information. This meta-information is provided for each conflict annotation.

Delete/Update Conflict. In Figure 5.4 the merge rule for Delete/Update conflicts is depicted. A Delete/Update conflict is reported, when the same object `o1` is changed by updating one of its features while it is deleted in parallel.

In this case, we prioritize the update and annotate the object `o1` with a Delete/Update annotation. In this annotation, the old value and the update value of the changed feature is preserved. Please note that in order to represent this kind of conflict in a model, the deletion is not executed but only marked by the annotation. The actual deletion is deferred as long as the modelers reach the conclusion that the deletion has a higher priority as the update.

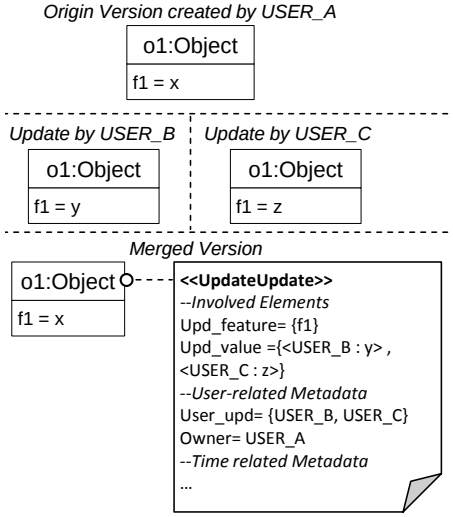

Figure 5.3: Merge Rule for Update/Update Conflict

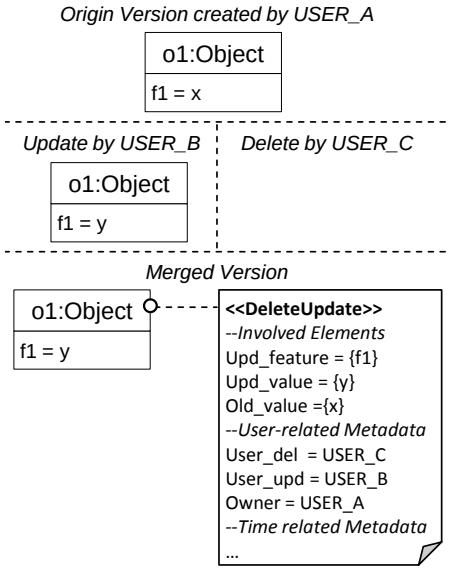

Figure 5.4: Merge Rule for Delete/Update Conflict

Delete/Use Conflict. The merge rule for Delete/Use conflicts is depicted in Figure 5.5. The object `o2` is used as reference value of feature `f1` of the object `o1`. In parallel, the used object `o2` is deleted.

This conflict is handled similar to a Delete/Update conflict. The deletion of object `o2` is not performed and the reference to this object is still available in the merged version. In this version, the deleted object is annotated and the old value of the changed feature `f1` is preserved in this

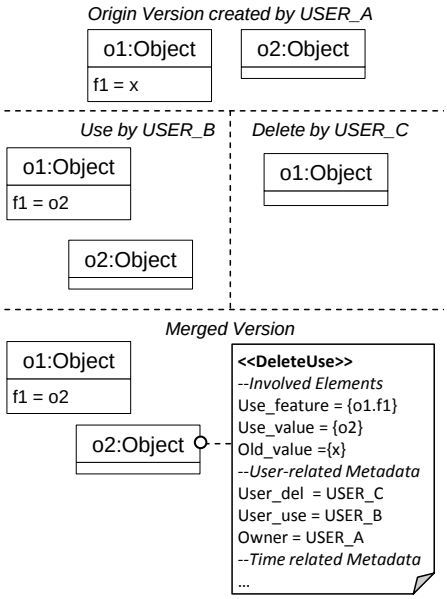

Figure 5.5: Merge Rule for Delete/Use Conflict

annotation.

Move/Move Conflict. When an object `o1` is concurrently moved into different container (`c2` and `c3`) as depicted in Figure 5.6 a so-called Move/Move conflict arises due to the fact that the container of an object always has to be unique.

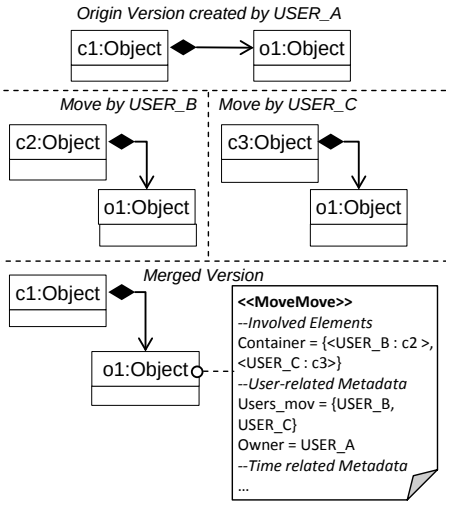

Figure 5.6: Merge Rule for Move/Move Conflict

This conflict is handled similarly to an Update/Update conflict, i.e., the original structure is retained and the contradicting changes are not executed but only annotated. Since both move

operations cannot be executed together, the object `o1` is still contained by `c1`, the original container, after merging and is marked with a Move/Move annotation. This annotation contains both container objects, i.e., `c2` and `c3`, to which object `o1` has been concurrently moved.

Delete/Move Conflict. A special kind of Delete/Update conflicts are Delete/Move conflicts. Such conflicts arise when the object, which is moved into another container, is concurrently deleted. Figure 5.7 illustrates such a situation in which the object `o1` is moved into the container `c2` while `o1` is concurrently deleted.

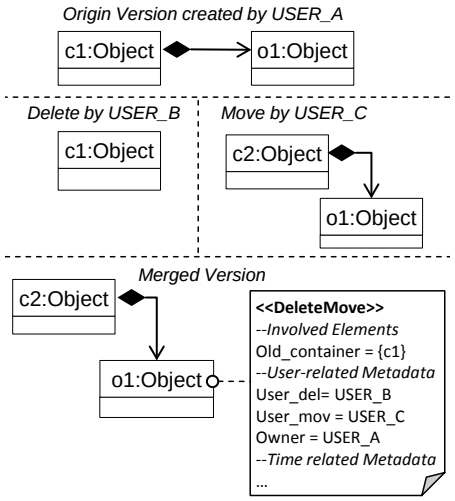

Figure 5.7: Merge Rule for Delete/Move Conflict

As already mentioned, this conflict is similar to the Delete/Update conflict. Therefore it is handled analogously during merging. The delete operation is not executed, the object is moved into the new container, and it is appropriately annotated with a Delete/Move annotation as depicted in Figure 5.7.

Move/Update Warning. Figure 5.8 depicts a situation, where the object `o1` is moved into a new container `c2` and a feature `f1` of object `o1` is concurrently updated. Basically, both operations can be executed, and therefore no conflict should be reported. However, when changing the context of a model element by moving it into another container while having concurrent updates may lead to undesired situations.

To give the modelers the chance to review such situations, the moved object is annotated with a dedicated warning to make the modeler aware of a potential problem. In particular, a Move/Update annotation is attached to the moved object which also comprises the old value of the updated feature. Please note that warnings are visualized in another way than conflicts.

Other kinds of warning are the **Use/Update Warning** and the **Use/Move Warning**. Similar to the Move/Update Warning, both concurrent operations are executed during the merge and the updated or moved element, which is referenced ("used") is annotated with a dedicated warning.

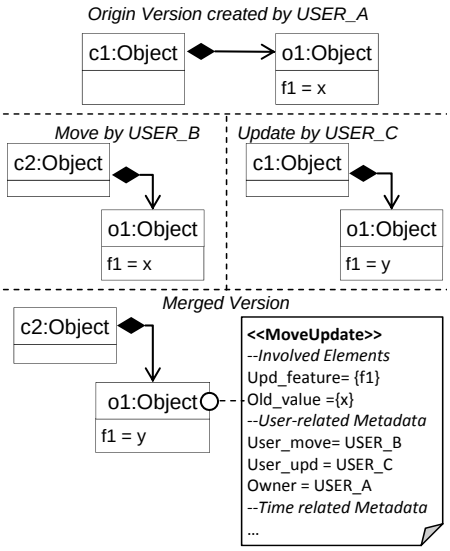

Figure 5.8: Merge Rule for Move/Update Warning

N-Way Merge Rules

In the previous subsection, we have presented different merge rules, which are applied if a conflict arises between two users. This so-called three-way merging is insufficient when three or more users perform their changes to the same model in parallel following the conflict-tolerant merge approach. For instance, when two users rename the same model element differently, the original value remains in the merged model and both update values are kept in the dedicated conflict annotation. If a third user also renames the same element differently, no conflict would be reported. Thus, in the following, we present merge rules, which are applied, when three or more modelers change the same model element in parallel. When merging in a conflict-tolerant way and two or more users participate, it is important that also the already reported conflicts in terms of annotations are checked. For instance, when a Delete/Update conflict occurs between two versions, the update is executed, the deletion is omitted, and the dedicated model element is marked as conflict. If a third user has also updated the same model element in parallel, the merge rules presented in 5.2.2 would not match. Therefore, the conflict detection must also check the applied annotations of the changed attribute.

Furthermore, we want to present the modelers with the conflicts in an aggregated way to avoid an overload of similar conflict annotations applied on the same element. For instance, if a model element is already annotated by a Delete/Update conflict and a third modeler also changes the same element in terms of an update, a Delete/Update and an Update/Update conflict would additionally be reported. Thus, we *extend* the first conflict annotation resulting in only one Delete/Update annotation containing information about one deletion and two updates. Each change type can be instantiated as often as necessary. This means, that in the case described before, any number of deletes or updates might be performed on one element, but only one conflict annotation is applied.

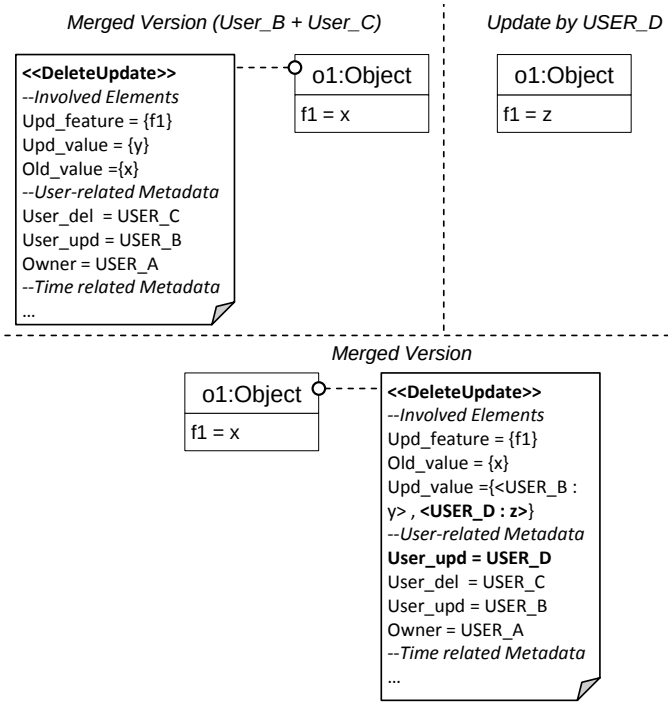

Figure 5.9: Merge Rule for Delete/Update/Update Conflict

In the following, we present two example merge rules where (i) a Delete/Update conflict and (ii) an Update/Update conflict are extended by a further update of the same model element performed by a third user.

In Figure 5.9, a merge rule is depicted for the following scenario: User B has updated the feature `f1` of object `o1` from the value `x` to `y`, which was deleted by User C, leading to a Delete/Update conflict as already discussed in Section 5.2.2 and depicted in Figure 5.4. However, User D has also updated the feature `f1` to the value `z` in parallel. Thus, with this rule the already applied Delete/Update conflict annotation is extended by including the update value and additional information of User D. This rule is needed, because it has to be checked, whether the updated object has already been deleted by analyzing all already applied annotations of this object. Since the object `o1` has already been deleted as indicated by the first Delete/Update conflict, the update of User D has to be seen as a concurrent change. The Delete/Update conflict annotation is extended to present the users the conflicts in a minimal and more comprehensible way as depicted at the bottom of Figure 5.9.

As mentioned before, the Conflict-tolerant Merge ensures that the changes of n users are considered when merging. For example, if a fourth user has also concurrently updated the same feature, the procedure is repeated and the applied annotation is extended again. In the following, we present another merge rule as depicted in Figure 5.10. In this scenario, three users perform three concurrent updates of the same model element.

User B and C has concurrently updated the feature `f1` of object `o1`. The merge rule for

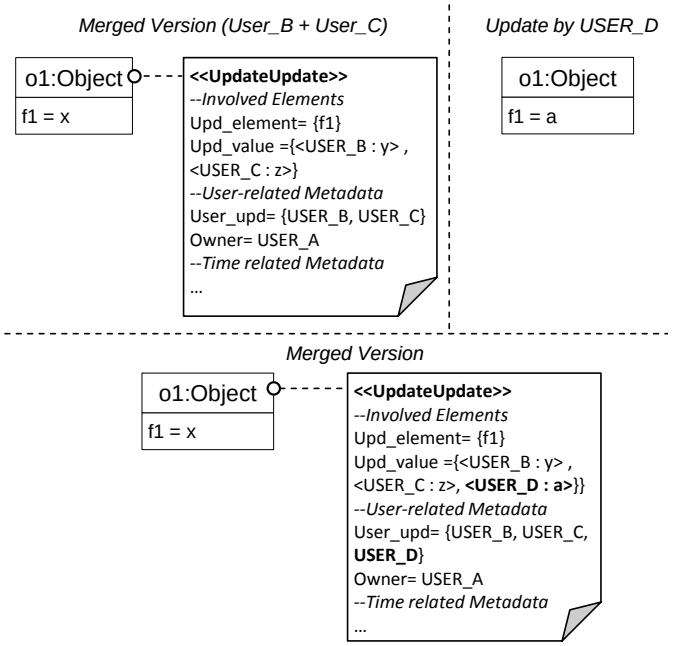

Figure 5.10: Merge Rule for Update/Update/Update Conflict

this Update/Update conflict is depicted in Figure 5.3. The original value `x` of `f1` remains in the merged model and the update values `y` and `z` are included in the conflict annotation. Now, User D has also updated the `f1` to the value `a` in parallel to User B and C. Although two more Update/Update conflicts are reported, the already applied Update/Update conflict annotation is extended by the third update. This annotation now contains all three updates.

Furthermore, it might happen that three or more different change types are performed on the same model element. Thus, beside the conflict pairs presented before, the following aggregated conflicts and warnings have to be introduced. The order of the individual change types of each combination does not matter and, for each change type, any number of changes can be included in the conflict annotation as discussed above.

- Delete - Update - Use Conflict
- Delete - Move - Use Conflict
- Delete - Move - Update Conflict
- Delete - Update - Use - Move Conflict
- Update - Move - Use Warning

For instance, if a Delete/Update conflict annotation is already applied due to the changes of User A and B, and User C now “uses” this Element by setting a reference, the conflict annotation is extended to indicate a Delete/Update/Use Conflict. If a User D would also perform a use

operation in parallel, the Delete/Update/Use conflict annotation would be extended by including the second use operation as presented by the examples before. Similar to this scenario, all other combinations may be built to ensure that N parallel versions of a model may be merged in a conflict-tolerant way and that the conflicts are presented to the users in an aggregated view.

5.2.3 The Merge Algorithm at a Glance

In this subsection, we present an overview on the conflict-tolerant merge algorithm. Figure 5.11 depicts a UML Class Diagram including all classes used in this algorithm. The diagram contains a class `Model` representing a versioned software model. A `Model` contains one root `ModelElement`, which again might contain several child elements. Each model element has an `ID` and knows its container model element. Model elements may be enriched with several `Annotations` and can be further described by `MetaInfos` which contains information about the users who recently changed the model elements, the `Changes` themselves, and the base version. The aggregated annotations presented in the previous subsection are not depicted for reasons of clarity.

The algorithm consists of two phases, in particular, the *fusion* phase and the *validation* phase. In the *fusion* phase, the algorithm iterates over all changes and checks for conflicts by using the afore presented conflict rules. Once the conflicts are identified, a copy of the common origin model of the modified models is created and all non-conflicting changes are applied to this copy. Next, conflicting changes are regarded and merged as presented in the previous section. By this, all conflicts are resolved by prioritizing updates and moves over deletions and contradictory updates are omitted. Furthermore, annotations indicating the conflict and providing all relevant meta-data are immediately added to the involved model elements. This allows us to fully automatically produce a merged model in any case which is additionally enhanced with conflict information based on annotations.

In the *validation* phase, the merged model is validated. Validation means to reveal violations of rules and constraints defined by the modeling language. Thus, we reuse model validation frameworks which are able to detect language constraint violations in the merged model. As in the fusion phase, we annotate all model elements which are subject to violations in this phase. Therefore, we iterate through all revealed violations and annotate the elements involved in the violation by so-called `Violation` annotations.

5.2.4 Merging the Running Example

In this subsection, the afore presented merge algorithm is applied to the running example (cf. Figure 5.2) in order to illustrate its function in more detail. In contrast to the standard merge depicted in Figure 5.2, where conflicts are resolved immediately comparing changed models pairwise, Figure 5.12 illustrates the merge results obtained using the Conflict-tolerant Merge and the annotations of the elements marked with the corresponding number.

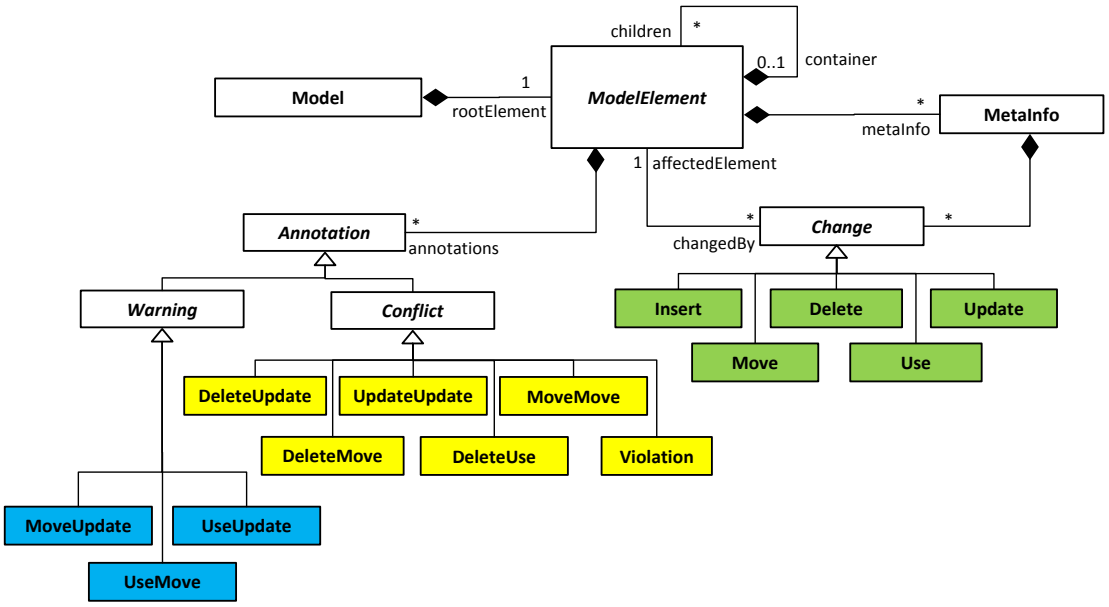

Figure 5.11: Changes and Annotations at a Glance

Merging V1a with V1b into V2

In the example introduced in Section 1.1, Sally checks in after Harry has committed his changes to the common repository. His changes comprise the insertion of the attribute `color` in class `Car`, and three updates. He updated the name of attribute `bday` to `birthday` in class `Employee`, also the name of the attribute `type` to `carType`, and changed the aggregation type of the reference connecting `Employee` and `Car` from `unspecified` to `composition`. Consequently, the classes `Car` and `Employee` as well as the reference have to be considered as updated when Sally's version V1b is merged with the latest version V1a of Harry.

Fusion Phase. Sally's changes comprise a move of the attribute `type`, a deletion of class `Car`, and a deletion of the reference connecting `Employee` and `Car`. Subsequently, the merge algorithm iterates over all changes of Sally and checks for each changed element if an afore presented merge rule has to be applied. For the first change, i.e., the move of the attribute `type`, the conditions of the "Move/Update rule" match, because this element has been concurrently updated by Harry. Since the rule executes both operations and annotates the element with a `Move/Update Warning`, in the merged version V2 the attribute `carType` is contained by the class `Employee` and annotated with the according conflict. The second change of Sally is the deletion of the class `Car` leading to a Delete/Update conflict, because the same class has been updated by Harry by inserting the attribute `color`. The "Delete/Update" rule ignores the deletion of the class `Car`, executes the the insertion of the attribute, and attaches the class with a `Delete/Update Conflict` annotation. The same is true for the third change of Sally, i.e., the deletion of the reference between the classes `Car` and `Employee`: the deletion of the reference is ignored but is updated to a composition and a `Delete/Update Conflict`

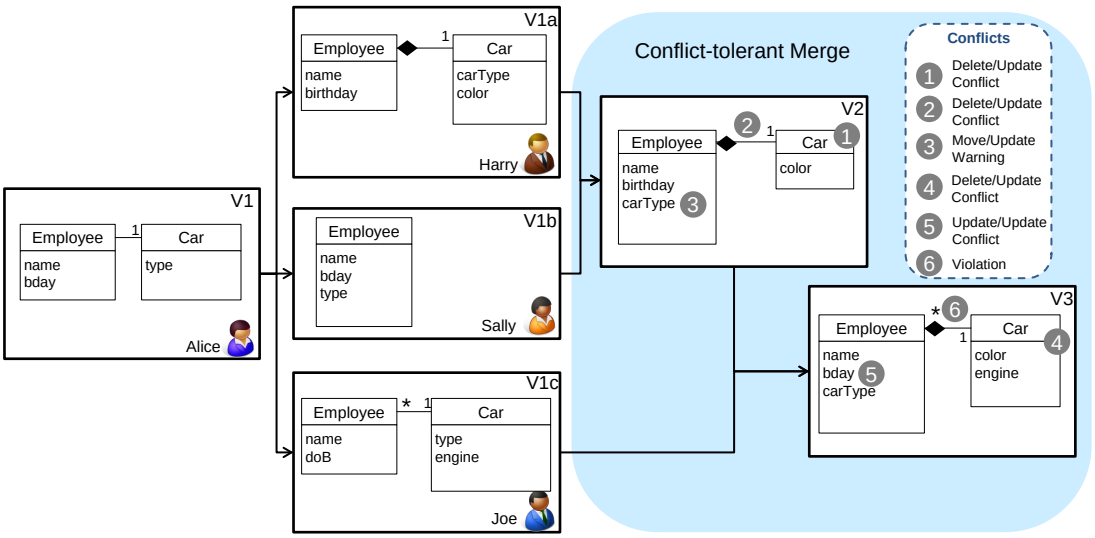

Figure 5.12: Conflict-tolerant Merge of the Running Example and Annotated Conflicts

annotation is applied to the reference. Since Harry has updated the name of the attribute `bday` to `birthday` and Sally did not perform any operation on this element, no special merge rule matches and, thus, the update of Harry is performed without annotating the attribute.

Validation Phase. Now the merged version is validated but since there are no language violations in this model, the merge algorithm terminates and the merged version is published into the central repository (cf. V2 in Figure 5.12).

Merging V2 with V1c into V3

According to Figure 5.2, Joe checks in his version V1c, which has to be merged with the head version V2 in the repository.

Between V1 and V2, Harry and Sally performed several changes, which are encompassed in V2. Recall that Harry updated the attribute `bday` as well as the type of the reference from `Employee` to `Car`. Moreover, he added an attribute to `Car`. Sally removed the class `Car` and also the reference between `Employee` and `Car`.

Fusion Phase. The first change of Joe comprises the renaming of the attribute `bday` to `doB`. Since this attribute has also been updated concurrently by Harry, the condition of "Update/Update" merge rule matches. Consequently, the rename of Harry is reverted to its original name `bday` and an `Update/Update Conflict` annotation is applied. Recall that the new values of this element, i.e., `birthday` and `doB`, are not deleted. Instead, they are saved as meta information in the annotation. The next change of Joe concerns the insertion of the attribute `engine` into the class `Car`. Since the class `Car` is already attached with a `Delete/Update Conflict` annotation, this annotation is extended containing also the information about the update operation of Joe. The final change of Joe concerns the update of the reference multiplicity which, in particular, has been set to unbounded (notated as $\star$). Since no merge rule matches,

this operation can be performed without causing a conflict.

Validation Phase. After all changes have been handled, we may move on to the validation phase. When the merged version of the model V3 is validated, a language violation is reported. References of type `Composition` may not have an unbound multiplicity. Consequently, each element involved in this violation is marked with a `Violation` annotation. In our example, only the reference is involved and annotated.

5.3 Consolidation

After all developers have finished to contribute changes to the repository, the all-encompassing head revision in the repository might contain several conflicts and inconsistencies. At a certain point in time during the software development project, a consolidated model version which reflects a unified view on the modeled domain has to be found by all participants before the model is further used. This point in time may be defined for each development project depending on the applied development process. Thus, the versioning process does not dictate the development process. In contrast, developers have the freedom to choose when resolving the conflicts according to the project phase and the purpose of the used model.

5.3.1 Conflict Resolution

For supporting the consolidation phase, we have developed a dedicated structural model defining the relevant information about a conflict resolution as well as a dedicated behavioral model for formalizing the life cycle of conflicts. The resulting *Conflict Resolution Model* is depicted in Figure 5.13. A `ModelElement` may be annotated by a conflict as already discussed in the previous section. A conflict may be assigned to different users, which are responsible to resolve the conflict. These users may propose different resolutions, but exactly one of these resolutions has to be finally accepted in order to resolve the conflict. Two possible kinds of resolution strategies exist: (1) either select *one* out of the conflicting changes, or (2) discard both and perform a custom resolution, which may contain several changes. In the latter case, the modeled resolution is stored as its own `Diff` to comprehend afterwards what happened to the conflict in the resolution process.

Furthermore, a conflict may *depend* on another conflict: Coming back to the example depicted in Figure 5.12, a Delete/Update conflict is already annotated on class `Car`, because Sally has deleted this class which is updated by Harry by introducing the new attribute `color`. Since Joe has also introduced in parallel the attribute `engine` in the same class, a further Delete/Update conflict is reported, but, as described before, the conflict annotation is integrated in the Delete/Update conflict annotation already applied on the class `Car`. The resolution of one conflict is now dependent on the resolution of the other conflict. In our example, if the class `Car` should remain in the model, both independent updates, i.e., insertion of attribute `color` and `engine` are prioritized if they are not contradicting. More investigations on how to automatically resolve dependent conflicts will be conducted in future work and, thus, we do not go into more detail in this thesis.

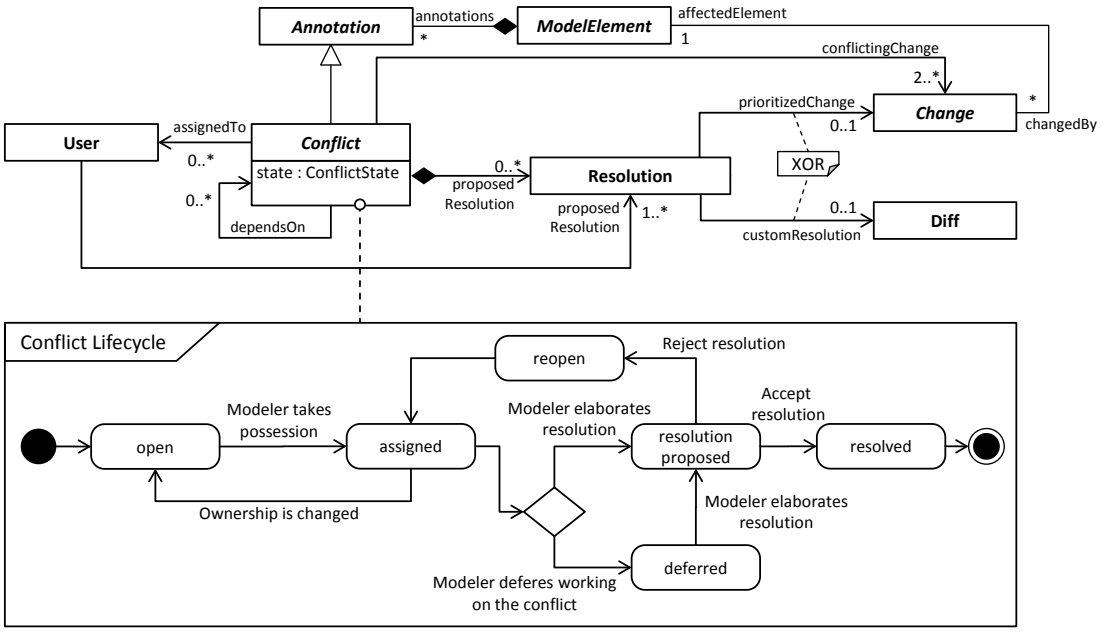

Figure 5.13: Conflict Resolution Model with Conflict Lifecycle

In the conflict resolution process, a conflict passes through different states which is comparable with the lifecycle of a bug in a dedicated tracking system like Bugzilla[2]. At the bottom of Figure 5.13 these states are depicted with the help of a UML State Diagram: When a conflict is reported it is in the state `open`. Now, modelers can be `assigned` to the dedicated conflict. These modelers are not necessarily the same like those who have caused the conflict, but may also include the modeler who has created the model element. After that, modelers may now elaborate the resolution or explicitly defer working on the conflict. Therefore, there is the additional state `deferred` that is important to make the other modelers aware of the deferral of the conflict resolution if, for example, more time is needed to resolve the conflict. As soon as a resolution has been elaborated by the assigned modelers, a conflict is in the state `resolution proposed` and can be reviewed by the others. If this resolution is rejected, the conflict can be `reopened` to start a new resolution phase. In the opposite case, the conflict is in the state `resolved` which directly leads to the end state of its lifecycle.

In Figure 5.14 the resolution process is depicted for a concrete Delete/Update conflict. In this example, the deletion of the class `Car` is conflicting with the insertion of the attribute `color` leading to a Delete/Update conflict. Since our conflict tolerant merge does not execute conflicting delete operations, but only marks the involved elements as deleted, the class `Car` remains in the merged version of the model and the attribute `color` is also included. In the first step, the class is annotated with a dedicated annotation and the state of the conflict is `open`. In the second step, the user `Harry` is assigned to the conflict, who is now responsible to resolve it. He proposes a new resolution, by selecting the update of the class `Car`. This is represented in

[2] `http://www.bugzilla.org/docs/2.18/html/lifecycle.html`

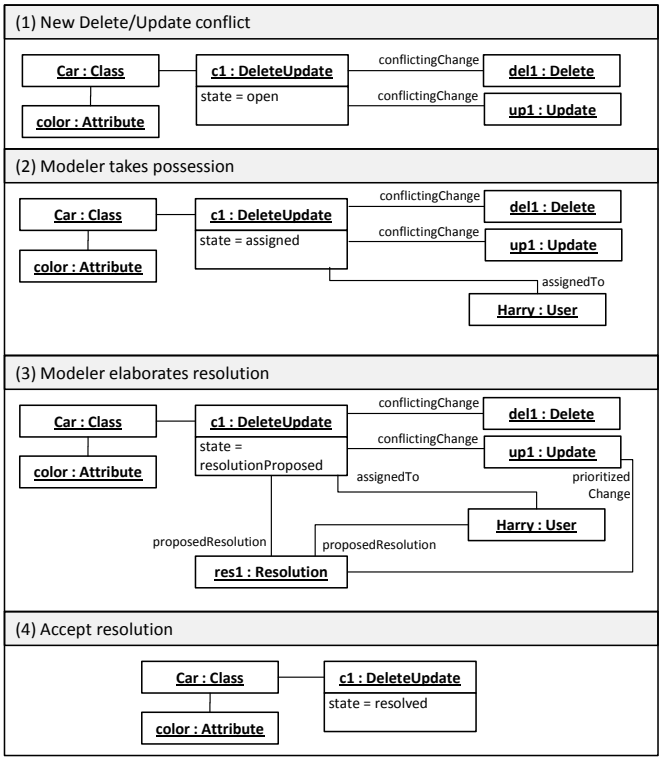

Figure 5.14: Delete/Update Conflict Resolution Example

the model by having a link from the resolution object to the prioritized change, i.e., the update object in our example. After this resolution proposal is reviewed by others and finally accepted, the merged version still contains the class `Car` with its attribute `color`. Finally, the conflict annotation is hidden but still available in the central model repository for provenance reasons.

In the following, a second example depicted in Figure 5.15 is presented where the resolution of an Update/Update conflict is conducted. Harry and Joe have concurrently changed the name of the attribute `bday`, which was created by Alice, to `birthday` and `doB`, respectively. Thus, in the merged version the attribute has its origin name `bday` and it annotated with an Update/Update conflict. The state of the conflict is `open`. In a second step, not only Harry and Joe are assigned to this conflict to resolve it, but in this concrete example also Alice, because she is the creator of the model element. Thirdly, since `birthday` is a more proper name for an `Employee`'s birthday, they decide to prioritize the update of Harry, i.e., `Up2`. Note that more than one modeler may propose more than one resolution proposal. However, in this example only Alice explicitly proposes the resolution of the conflict. Finally, the resolution proposal is accepted and the update is performed. Now, the attribute is named `birthday`. The state of the conflict is changed to `resolved` and, as mentioned before, it is hidden but kept in the repository.

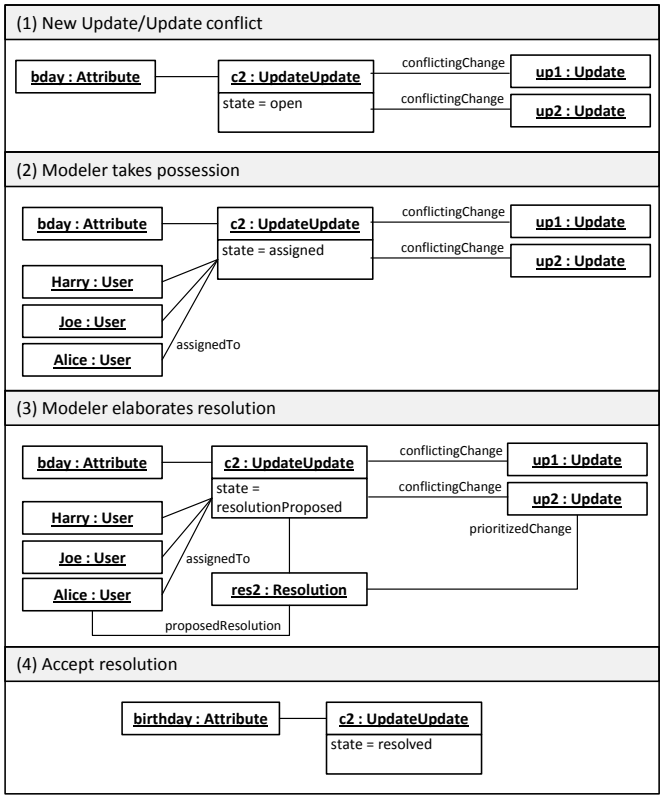

Figure 5.15: Update/Update Conflict Resolution Example

5.3.2 Consolidating the Running Example

The *consolidation phase* is supported by an adequate visualization of all conflicts in the unconsolidated model. This view serves as a basis to discuss existing issues and different points of view.

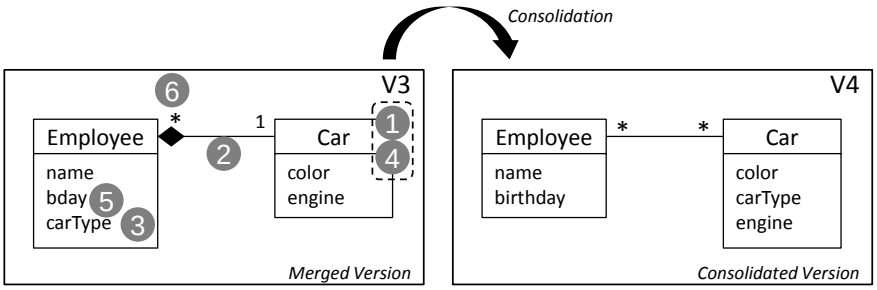

Figure 5.16: Consolidated Version by Turning Conflicts into Collaborations

Coming back to our running example, Harry, Sally, Joe, and Alice collaborate to resolve

all existing conflicts. Different possibilities exist how the developers are able resolve the occurred conflicts collaboratively. In this chapter, we presented the possibility of asynchronous conflict resolution, where the assigned modelers can prioritize one of the conflicting operations. In [BSW+09] we proposed synchronous conflict resolution, where the modelers may remotely discuss and resolve the conflicts together. Also for this kind of resolution, the conflict annotations build a good basis for discussion. When evaluating the Conflict-tolerant Merge as presented in Chapter 7, we used face-to-face session for resolving the occurred conflicts. In the following, we discuss a potential collaborative resolution for each conflict in the running example in more detail. Supported by the conflict report and the provided metadata, they find a consolidated version which is depicted in Figure 5.16 by resolving the following six conflicts:

1. *Delete/Update: Class Car.* First of all, the modelers have to decide whether the class `Car` is needed. Since Sally was of the opinion that exactly one car is assigned to one employee, she has deleted the class `Car` and has inlined its attributes to the class `Employee` to ensure the efficient querying of information. However, after Harry and Joe communicate their opinions, all participants are convinced to keep the class `Car` in the model. Due to this decision, both attributes, i.e., `color` and `engine`, remains in the class `Car`.

2. *Delete/Update: Reference employee-to-car.* Since class `Car` is not deleted, the reference between both classes is also retained.

3. *Move/Update: Attribute carType.* The assigned modelers also check the occurred warning and notice that the attribute `carType` is now in the "wrong" class. Therefore, it is moved back to its origin class `Car`.

4. *Update/Update: Attribute bday.* Harry and Joe have concurrently renamed the attribute `bday`, which has been introduced by Alice. Both agree, that `bday` and `doB` may lead to misunderstandings and, therefore, decide to use `birthday`.

5. *Violation: Reference cardinality.* Harry has introduced the containment relationship between the two classes and Joe has set the multiplicity to "unbound" leading to a metamodel violation. They discuss the different possibilities and decide to resolve it manually as depicted in Figure 5.17. In the final version, a company car can be assigned to more than one employees and one employee can be assigned to more than one car. Thus, the containment relationship is converted back to a non-containment relationship and both multiplicities are set to "unbound".

After Harry, Sally, Joe, and Alice have finished the resolution of all conflicts and, furthermore, all resolution proposals are accepted by the responsible modelers, the consolidated version of the model is then saved in the repository as new version *V4*.

The presented example illustrates that it is highly beneficial to conjointly discuss each conflict because there are different ways to resolve them due to the different viewpoints of the involved modelers. Conflict resolution is an error-prone task when only one modeler has the full responsibility to resolve conflicts. Our presented approach counteracts this problem. We retain all information necessary to make reasonable resolution decisions is kept and collaboration

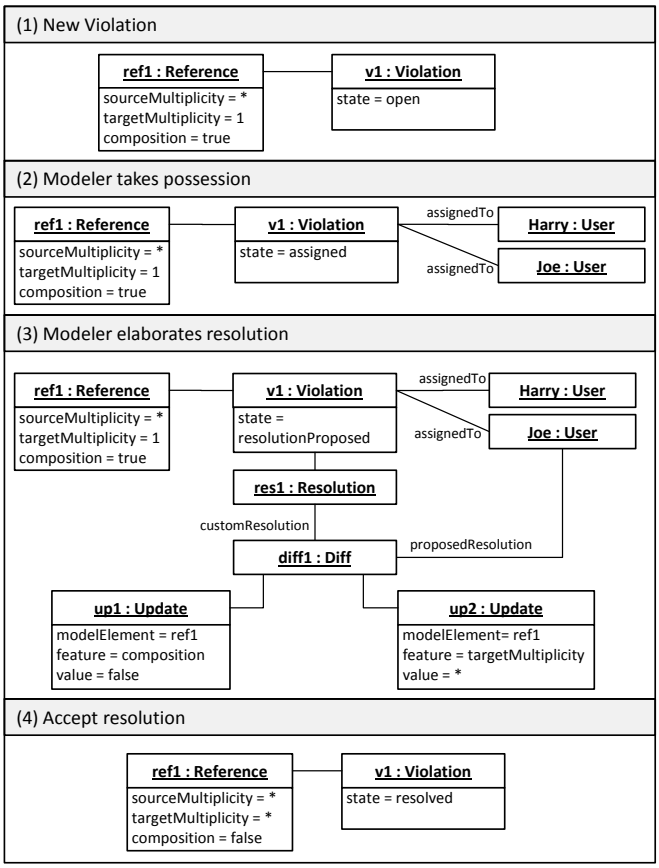

Figure 5.17: Violation Resolution Example

and discussion is fostered. The resulting final version (cf. Figure 5.16) reflects all intentions much better than this could have been achieved by one modeler on her own. In summary, such a consolidation leads to a unification of the different viewpoints and finally to a model of higher quality accepted by all team members.

5.4 Summary

In this chapter, we presented a novel approach for optimistic model versioning. Instead of forcing the modelers to resolve merge conflicts immediately, our system supports deferring the resolution decision until a consolidated decision of the involved parties has been elaborated. Well-defined merge rules are used by our algorithm to incorporate all changes of the modelers, also when overlapping changes happen, and to mark conflicts by dedicated annotations. The resolution process itself is supported by a conflict resolution model, with which the participants can be assigned to specific conflicts to propose resolutions and find collaboratively a consolidated version. By this method, we transform conflicts into collaborations which results in a higher

acceptance of the consolidated model. In the next chapter, we present the realization of this approach in the context of our model versioning system AMOR.

CHAPTER 6

Making AMOR Collaboration-Aware

We created a prototypical implementation of the concepts presented before which is presented in this chapter and evaluated in Chapter 7. Our implementation is realized as Eclipse[1] plug-in and is built upon the Eclipse Modeling Framework (EMF) [BSM+03], one of the most adopted (meta-)modeling frameworks in practice. Due to this widespread adoption, a plethora of modeling languages are specified in EMF—among them UML 2.0. Using the powerful reflection mechanisms of EMF, we designed our implementation to support every EMF-based modeling language. In the following, we outline the architecture of the Conflict-tolerant Merge and the implementation of all components required to realize the presented concepts. One of the main components is the light-weight annotation mechanism called EMF Profiles, which we present in detail in Section 6.3 after discussing other model annotation mechanisms in Section 6.2. The design rationale for the Conflict-tolerant Merge in conjunction with EMF Profiles is recalled in the following:

- *User friendly visualization:* Merge conflicts as well as the information on performed changes are presented in the concrete syntax.
- *Integrated view:* All information is visualized within a single diagram to provide a complete overview of conflicts.
- *Preventing metamodel pollution:* The models incorporating the conflict information are still compliant to their metamodel.
- *No editor modifications:* The visualization of conflicts in Ecore-based models are possible without modifying the graphical Ecore editors.
- *Model-based representation:* If models are exchanged between different Ecore tools, the conflict information is not lost, because conflicts are explicitly represented as model elements.

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

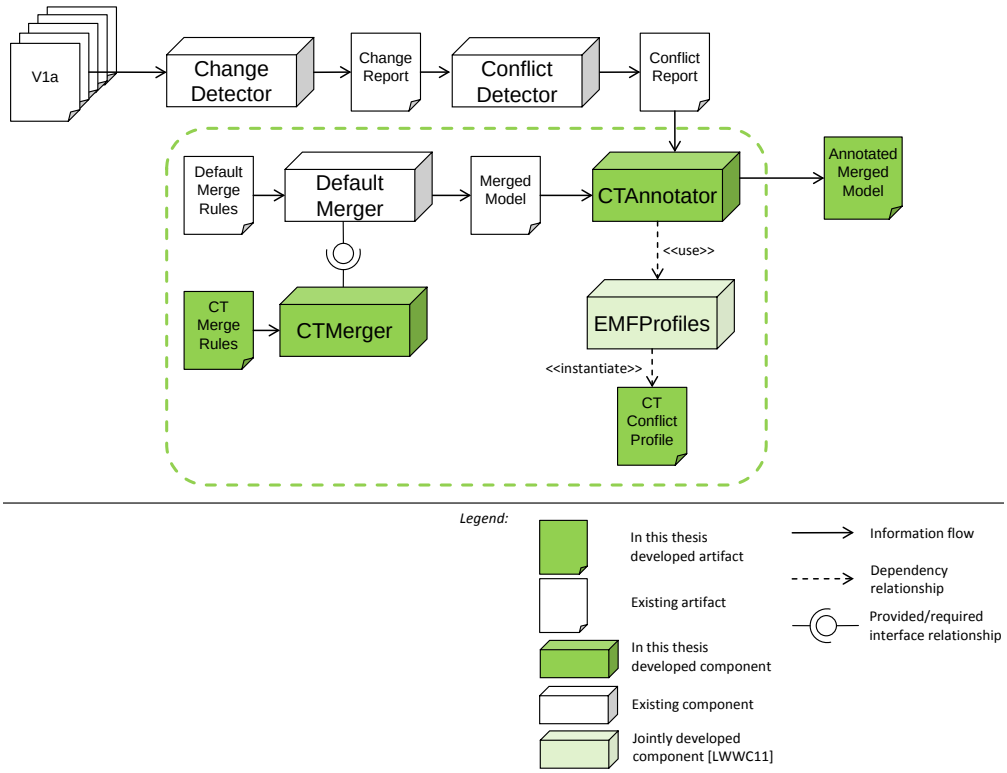

Figure 6.1: Architecture of Conflict-tolerant Merge Tool

6.1 Architecture and Implementation

To better understand our design decisions, we want to recall in the following the requirements for the approach presented in this thesis: Overall, we want to mark conflicts to resolve them later on. Thus, the author of this thesis developed a model merge tool following the Conflict-tolerant Merge rules (cf. Section 5.2.2 and implemented conjointly an annotation mechanism (cf. [LWWC11]), which annotates conflicts or changes, as light-weighty as possible to reuse existing modeling editors. In addition, the Conflict-tolerant Merge including the conflict annotations should not be restricted to only certain modeling languages. Following the principles of AMOR any Ecore-based modeling language should be supported. In contrast to traditional/state-of-the-art merge tools, our system should provide the possibility to merge any number of parallel versions of a model while considering the difference and conflicts between all versions. Finally, for supporting the modelers to better understand the changes of the others and to resolve the occurred conflicts, we also want to visualize the changes and conflicts in the concrete syntax, e.g., graphical notation, of the models.

In Figure 6.1, the most important components of the Conflict-tolerant Merge are visualized and described in the following.

Change Detector

To identify the changes applied between an origin model and a revision of it, two different approaches exist. On the one hand, *state-based* approaches take two versions of a model as input and compute the model differences by comparing these two model states [Men02]. On the other hand, as introduced by [LvO92], *operation-based* approaches obtain the changes by directly recording them in the modeling environment as they are performed by the user. Both approaches have their advantages and disadvantages and are already discussed in Chapter 2. In comparison to state-based approaches, change recording is, in general, more precise and potentially enables to gather more information (e.g., the order in which the changes have been applied) than state-based approaches. However, these advantages come at the price of inherently strong editor-dependence because the editor used for modifying the model has to be capable of recording changes and must represent them in a common format. To avoid depending on the used editor, we conduct in AMOR state-based model differencing based on the extensible model comparison framework EMF Compare [BP08]. To increase the precision of EMF Compare, we implemented an extension to EMF Compare exploiting *universally unique identifiers* (UUIDs) which are attached to each model element and which are, once assigned, never changed anymore. Thus, using UUIDs also moved and intensively changed model elements can be recognized across two versions of the same model which allows for a more precise computation of applied changes. Once, the changes between an original model and two revisions of it are identified, they are saved to two so-called *difference models* and passed to the conflict detection component.

For more details on the `Change Detector` as well as the `Conflict Detector`, which is presented in the following, we kindly refer to [Lan11].

Conflict Detector

Having identified the precise changes concurrently applied by two users, we may now proceed with detecting conflicts between them. In Section 5.2, we introduced rules for detecting, annotating, and merging conflicts. These rules involve a change pattern which specifies two changes concurrently applied to the same origin model element (cf. upper left area of Figures 5.3–5.8). Hence, as soon as such a pattern exists in the difference models, a corresponding conflict is at hand. For instance, having two *updates* changing the same model element, a conflict shall be raised (cf. Figure 5.3). Having these rules, implementing the conflict detection component is largely straightforward. Generally speaking, for all change combinations of both difference models it has to be checked whether one of the aforementioned conflict patterns matches and raises, in case a match is at hand, a proper conflict. However, for the sake of efficiency, we refrain from checking the complete cross product of all change combinations between all changes of both difference models. In contrast, both difference models are translated in a first step into an optimized view which sorts all changes based on their type into potentially conflicting combinations. Secondly, these combinations are triaged whether they are spatially affected overlapping parts of the original model. Finally, all remaining combinations are checked in detail by evaluating the rules presented before. Once, all conflicting combinations of changes are identified, they are marked accordingly to be regarded in the following step.

Besides conflicting changes, the merged model may incorporate violations of the confor-

mance rules of the respective modeling language. For detecting such violations, the EMF Validation Framework[2] can be used. By this, each EMF-based model may be validated to detect violations of constraints arising directly from the metamodel as well as those coming from additionally defined constraints expressed in OCL or Java. Whenever a violation is detected, diagnostics are returned which describe the severity of the constraint violation and provide an error message as well as the model elements involved in the respective violation. This information is used to mark violations besides overlapping changes in the merged model in the next step. The integration of the EMF Validation Framework has not been conducted yet. However, it is subject to future work.

CTMerger

Instead of the `Default Merger`, we implemented the so-called `CTMerger`, which overwrite the `Default Merge Rules` by the `CT Merge Rules` presented in Chapter 5. Although the Conflict-tolerant Merge allows for incorporating the changes of any number of parallel versions of a model, in fact two versions are compared, i.e. the head version in the repository and the newly committed version. However, compared to a standard versioning system, this system is designed to tolerate conflicts as long as the developers want to resolve them. Thus, from the user's point of view all parallel versions are incorporated to one merged versions and the conflicts between all versions are annotated. The general process of the `CTMerger` is depicted in Figure 6.2.

An `Origin Model` (V0) is in the `Central Repository` and Modeler A and B performs their changes in parallel (cf. Mark 1).

When Modeler A checks in her version, the changes of $diff_1$ are applied in step 2 leading to a new `Head Model` in the repository. When Modeler B checks in, the `Conflict Detector` identifies all conflicting changes between the changes of $diff_1$ and $diff_2$. Afterwards, we may create a merged model according to the merge rules specified in Section 5.2. In the third step, a copy of the common origin model is created. Of course, in this copy the UUIDs of the origin model must be retained. Next, all non-conflicting changes contained by both difference models are reapplied to the created copy (cf. Mark 4). Therefore, we implemented a dedicated model transformation engine based on the *merging framework* of EMF Compare [BP08] which is able to transform a model according to the afore detected model differences. The model elements to be transformed are identified using the aforementioned UUIDs. In the next step, we handle conflicting changes according to the conflict-tolerant merge rules presented in Section 5.2.2. To recall, if a Delete/Update, Delete/Use or Delete/Move conflict occur, the change operation is prioritized over the deletion by the `CTMerger` and applied to the merged model. When an Update/Update or Move/Move conflict occur, no change is applied to the merged model.

The occurred conflicts are marked by means of annotations leading to an `Annotated Merged Model` in the central repository. If a Modeler C has additionally checked out the `Origin Model`, the `Origin Model`, is copied again and the non-conflicting changes of all three diffs are applied to this model. Finally, the occurred conflicts are handled as described

[2] `http://www.eclipse.org/modeling/emf/?project=validation#validation`

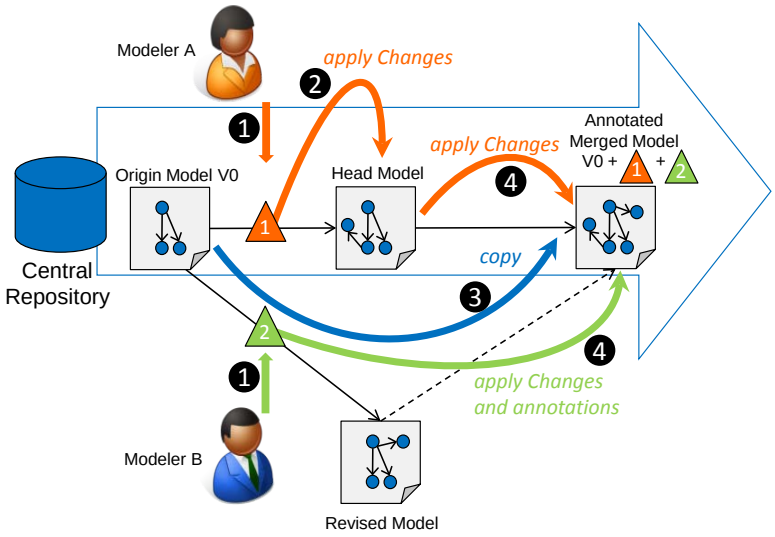

Figure 6.2: Overview of CTMerger Process

above. But now, the already applied annotations of the conflicts between Modeler A and B have to be considered when detecting and reporting conflicts between the version of Modeler C and the `Annotated Merged Model` in the repository as described in Chapter 5.

CTAnnotator

Finally, the merged model is enriched with the conflict and violation annotations as specified by the merge rules in Figures 5.3–5.8 conforming to the annotation metamodel depicted in Figure 5.11. Unfortunately, EMF does not inherently provide a common annotation mechanism for enriching a model with additional information. Therefore, we ported the lightweight extension mechanism known from UML Profiles [FFVM04] to the realm of EMF models as presented by [LWWC11]. By this, every EMF-based model may be annotated with stereotypes containing tagged values. If, for instance, an *update/update* conflict appeared (cf. Figure 5.3), the corresponding stereotype `UpdateUpdate` is applied to the object which was concurrently modified. This stereotype contains information—in terms of tagged values—on the contradictory updated values as well as the users who performed the conflicting changes. Stereotype applications may be visualized on top of the graphical representation of a model to support the user in resolving all annotated conflicts directly in the model. Annotations are saved in a separate model to avoid polluting the merged model. However, when checking-in the merged model comprising conflicting changes, the annotation model is saved alongside the merged model to also allow other modelers to investigate and resolve existing conflicts. More details about this annotation mechanism is presented in Section 6.3.

As discussed in Section 5.3, users may resolve conflicts by applying either one of the two changes or by performing a new custom set of changes. All of these options may be performed directly in the modeling editor. We exploit the `SelectionListener` interface which is im-

plemented by all EMF-based editors to allow for displaying all annotated conflicts in a dedicated view whenever a model element is selected by the user in the modeling editor. In this dedicated view, the user may now choose how to resolve the displayed conflict. Once this is done, the tagged value representing the state of the conflict is changed accordingly. After all or a subset of all conflicts are resolved, the model alongside the updated annotation model is again checked into the repository. As argued earlier, it is often desired by developers to collaboratively resolve certain conflicts in a synchronous manner in contrast to resolve a conflict by oneself [BSW+09]. Although this has not been integrated in our implementation yet, users may use *Connected Data Objects*[3] (CDO), a distributed shared model framework which allows for simultaneously editing EMF-based models over a network. Since the actual model as well as the annotation model is based on EMF, also a distributed collaborative resolution of existing conflicts is possible by using the CDO technology. In the evaluation of the approach, which is presented in Chapter 7, the participants had to consolidate their models in face-to-face sessions and they resolved the conflicts manually.

6.2 Annotation of Models

In the following, we outline different approaches for annotating models in general and, additionally, discuss approaches focusing on annotating and visualizing conflicts.

6.2.1 General Model Annotation Mechanisms

One alternative to profiles as an annotation mechanism is to use weaving models (e.g., by using Modelink[4] or the Atlas Model Weaver[5] [FBJ+05]). Model weaving enables to compose different separated models, and thus, could be used to compose a core model with a concern-specific information model in a non-invasive manner. However, although weaving models are a powerful mechanism, annotating models with weaving models is counter-intuitive. Since this is not the intended purpose of weaving models, users cannot annotate models using their familiar environment such as a diagramming editor which graphically visualizes the core model. Current approaches only allow for creating weaving models with specific tree-based editors in which there is no different visualization of the core model and the annotated information. Not least because of this, weaving models may quickly become very complex and challenging to manage.

Recently, Kolovos et al. presented an approach called *Model Decorations* [KRDM+10] tackling a very similar goal as EMF Profiles. Kolovos et al. proposed to attach (or "decorate") the additional information in terms of text fragments in GMF's *diagram notes*. To extract or inject the decorations from or into a model, hand-crafted model transformations are employed which translate the text fragments in the notes into a separate model and vice versa. Although their approach is very related to ours, there also are major differences. First, for enabling the decoration of a model, an extractor and injector transformation has to be manually developed which is not necessary with EMF Profiles. Second, since Kolovos et al. exploit GMF notes,

[3] http://wiki.eclipse.org/CDO

[4] http://www.eclipse.org/gmt/epsilon/doc/modelink

[5] http://www.eclipse.org/gmt/amw

only decorating GMF-based diagrams is possible. In contrast to our approach, models for which no GMF editor is available cannot be annotated. Third, the annotations are encoded in a textual format within the GMF notes. Consequently, typos or errors in these textual annotations cannot be automatically identified and reported while they are created by the user. Furthermore, users must be familiar with the textual syntax as well as the decoration's target metamodel (to which the extractor translates the decorations) to correctly annotate a model. In EMF Profiles, stereotypes may only be applied if they are actually applicable according to the profile definition and editing the tagged values is guided by a form-based property sheet. Consequently, invalid stereotype applications and tagged values can be largely avoided.

EMF Facet[6], a spin-off of the MoDisco subproject [BCJM10] of Eclipse, is another approach for non-intrusive extensions of Ecore-based metamodels. In particular, EMF Facet allows to define additional derived classes and features which are computed from already existing model elements by model queries expressed, e.g., in Java or OCL. Compared to EMF Profiles, EMF Facet targets on complementary extension direction, namely the dynamic extension of models with additional *transient information* derived from queries. In contrast, EMF Profiles allow to add new (not only derived) information and is able to persist this additional information in separate files. Nevertheless, the combination of both complementary approaches is worth to be subject for future work. For example, this would allow to automatically extend or complete models based on EMF Facet queries and persist this information with EMF Profiles.

The concept of meta-packages has been proposed in [CESW04] for the lightweight extension of the structural modeling language XCore which is based on packages, classes, and attributes. New modeling concepts are defined by extending the base elements of XCore and can be instantly used in the standard XCore editor. Compared to meta-packages, EMF Profiles are more generic, because not only one modeling language may be extended, but any Ecore-based modeling language.

6.2.2 Visualization and Annotation of Conflicts

In the following, we review approaches which support the visualization of changes and conflicts. Therefore, we consider not only the literature on software modeling, but also the literature on ontology engineering. Ontologies are usually established by communities, and therefore sophisticated mechanisms are required to enable collaborative work. As the ontology engineering process is highly interactive, mechanisms are required to keep conflicts in order to establish a consolidated opinion of the community.

Software Modeling. The visualization of differences between model versions by using different coloring and highlighting techniques are proposed by [MGH05] and [OWK03]. The modifications are shown in *unified diagrams* which incorporate the changes of both users. The approach of Ohst et al. has been implemented in [Nie04], but due to a restricted merging approach only conflicts concerning contradicting updates of attribute values as well as element moves are marked explicitly in the unified diagrams. Furthermore, tool-specific extensions have to be implemented for modeling editors in order to use this approach. Mehra et al. [MGH05] also

[6] `http://www.eclipse.org/modeling/emft/facet`

report on changes concerning the concrete syntax. For each movement of the shape of a model element, the original as well as the new position of the shape connected by a line is shown. Many overlapping highlighted model elements are generated when a large number of changes has occurred. The approach has been implemented for the meta-CASE tool Pounamu [ZGH+07] for providing generic visualization support for modeling languages defined in Pounamu, but for UML modeling environments there is no support available.

Ontology Development. Ontologies are kind of conceptual models. Thus, there are many similarities between building an ontology and a structural model such as an UML class diagram. Furthermore, several works are geared towards integrating models and ontologies such as [WRK+06]. Ontologies cover common knowledge of a certain domain, and usually the building of an ontology is a community activity in order to collect common domain knowledge and in order to establish a common terminology. Hence, the ontology engineering community needs tools for collaboratively developing ontologies [SNTM08]. Often, ontologies are constructed by the means of Wikis like LexWiki[7] intended to define terminologies. Therefore, the participating engineers can comment the current status and propose changes in a text-based manner by annotations. These annotations are later examined by curators which are editing the ontologies in standard ontology editors separated from LexWiki. OntoWiki [ADR06] supports to change and rate ontology definitions via a Web-based interface. However, OntoWiki cannot represent conflicting changes explicitly. Collaborative Protégé [TNTM08] also allows for collaborative ontology development with annotations on ontology changes, proposals, votings, as well as discussions. The authors recognize the need for synchronous and asynchronous development as one of the main requirements for ontology engineering, but for the moment only synchronous development is supported. Consequently, conflict detection and visualization is not treated by these approaches in contrast to this thesis. Furthermore, ontologies are developed in abstract syntax using tree editor, thus no concrete syntax conflicts are considered.

As mentioned before, with our annotation mechanism, namely EMF Profiles, any EMF-based models can be extended in a light-weight way. Thus, it provides an easy way for marking conflicts and for enriching models with metadata when merging in a conflict-tolerant way. Based on previous work [LWWC11], the full potential of EMF Profiles is presented in the following:

6.3 Annotation Support with EMF Profiles

Domain-Specific Modeling Languages (DSMLs) have gained much attention in the last decade [KT08]. They considerably helped to raise the level of abstraction in software development by providing designers with modeling languages tailored to their application domain. However, as any other software artifact, DSMLs are continuously subjected to evolution in order to be adapted to the changing needs of the domain they represent. Currently, evolving DSMLs is a time-consuming and tedious task because not only its abstract and concrete syntax but also all related artifacts as well as all DSML-specific components of the modeling environment have to be re-created or adapted.

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

UML has avoided these problems by promoting the use of profiles. Indeed, the profile mechanism has been a key enabler for the success and widespread use of UML by providing a lightweight, language-inherent extension mechanism [Sel07]. Many UML tools allow the specification and usage of user-defined profiles and are often shipped with various pre-defined UML Profiles. Induced by their widespread adoption, several UML Profiles have even been standardized by the OMG[8]. In the last decade, many debates[9] on pros and cons of creating new modeling languages either by defining metamodels from scratch (with the additional burdens of creating a specific modeling environment and handling their evolution) or by extending the UML metamodel with UML Profiles (which provide only a limited language adaptation mechanism) have been going on.

However, in this chapter we propose a different solution to combine the best of both breeds. We advocate for adapting the UML Profiles concept as an annotation mechanism for *existing* DSMLs. We believe the usage of profiles in the realm of DSMLs brings several benefits:
(1) Lightweight language extension. One of the major advantages of UML Profiles is the ability to systematically introduce further language elements without having to re-create the whole modeling environment such as editors, transformations, and model APIs.
(2) Dynamic model extension. In contrast to direct metamodel extensions, also already existing models may be dynamically extended by additional profile information without recreating the extended model elements. One model element may be further annotated with several stereotypes (even contained in different profiles) at the same time which is equivalent to the model element having multiple types [AK07]. Furthermore, the additional information introduced by the profile application is kept separated from the model and, therefore, does not pollute the actual model instances.
(3) Preventing metamodel pollution. Information not coming from the modeling domain, can be represented by additional profiles without polluting the actual domain metamodels. Consider for instance annotating the results of a model review (as known from code reviewing) which shall be attached to the reviewed domain models. Metaclasses concerning model reviews do not particularly relate to the domain and, therefore, should not be introduced in the domain metamodels. Using specific profiles instead helps to separate such concerns from the domain metamodel and keeps the metamodel concise and consequently, the language complexity small.
(4) Model-based representation. Additional information, introduced to the models by profile applications, is accessible and processable like ordinary model information. Consequently, model engineers may reuse familiar model engineering technologies to process profile applications. Due to their model-based representation, profile applications may also be validated against the profile definition to ensure their consistency as it is known from metamodel/model conformance.

Until now, the notion of profiles has not been adopted in current metamodeling tools. Thus, the contribution behind the EMF Profiles project is to adapt the notion of UML profiles to arbitrary modeling languages residing in the Eclipse Modeling Framework[10] (EMF) which is currently one of the most popular metamodeling frameworks. Thanks to this, existing modeling

[8] `http://www.omg.org/technology/documents/profile_catalog.htm`

[9] Consider for instance the panel discussion "A DSL or UML Profile. Which would you use?" at MoDELS'05 (`http://www.cs.colostate.edu/models05/panels.html`)

[10] `http://www.eclipse.org/modeling/emf`

languages may easily be extended by profiles in the same way as it is known from UML tools. Besides this, we propose two novel techniques to enable the systematic reuse of profile definitions across different modeling languages. First, we introduce *generic profiles* which are created independently of the modeling language in the first place and may be bound later to *several modeling languages*. Second, we propose *meta profiles* for immediately reusing them for *all modeling languages*. Finally, we present how our prototype called EMF Profiles is integrated in EMF and how it is used for the Conflict-tolerant Merge.

6.3.1 From UML Profiles to EMF Profiles

In this section, we present the *standard profile* mechanism (as known from UML) for EMF. Firstly, we disclose our design principles. Secondly, we discuss how the profile mechanism may be integrated in EMF in a way that profiles can seamlessly be used within EMF following the previous design principles. Finally, we show how profiles as well as their applications are represented based on an example.

Design Principles

With EMF Profiles we aim at realizing the following five design principles. Firstly, annotating a model should be as *lightweight* as possible; hence, no adaptation of existing metamodels should be required. Secondly, we aim at avoiding to *pollute* existing metamodels with concerns not directly related to the modeling domain. Thirdly, we aim at *separating annotations from the base model* to allow importing only those annotations which are of current interest for a particular modeler in a particular situation. Fourthly, the annotations shall be *conforming to a formal and well-known specification* such as it is known from metamodel/model conformance. Finally, users should be enabled to intuitively attach annotations using environments and editors they are familiar with. Consequently, annotations shall be created either on top of the concrete (graphical) syntax of a model or on top of the abstract syntax using e.g., generic tree-based editors.

Integrating Profiles in the EMF Metalevel Architecture

The profile concept is foreseen as an integral part of the UML specification. Therefore, the UML package *Profiles*, which constitutes the language for specifying UML Profiles, resides, in terms of the metamodeling stack [KÖ6], at the meta-metalevel M_3 [Obj07] as depicted in Figure 6.3. A specific profile (*aProfile*), as an instance of the meta-metapackage *Profile*, is located at the metalevel M_2 and, therefore, resides on the same level as the UML metamodel itself. Thus, modelers may create profile applications (*aProfileApplication* on M_1) by instantiating *aProfile* just like any other concept in the UML metamodel.

To embed the profile mechanism into EMF, a language (equivalent to the package *Profiles* in Figure 6.3) for specifying profiles is needed as a first ingredient. This is easily achieved by creating an Ecore-based metamodel which is referred to as *Profile MM* (cf. column *Profile Definition* in Figure 6.4). Specific profiles, containing stereotypes and tagged values, may now be modeled by creating instances, referred to as *aProfile*, of this profile metamodel. Once a specific profile is at hand, users should now be enabled to apply this profile to arbitrary models

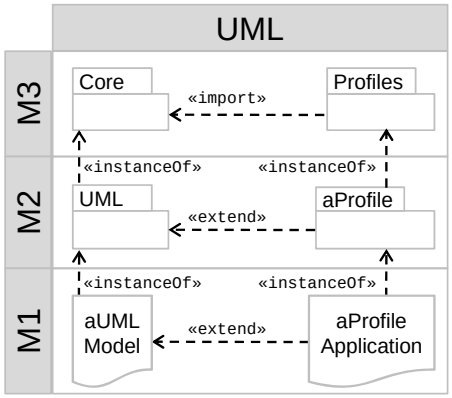

Figure 6.3: UML Architecture

by creating *stereotype applications* containing concrete values for tagged values defined in the stereotypes. In UML, a stereotype application is an instance—residing on M_1—of a stereotype specification in M_2 (cf. Figure 6.3).

Unfortunately, in contrast to the UML architecture, in EMF no profile support exists in M_3. The level M_3 in EMF is constituted only by the metamodeling language Ecore (an implementation of MOF [Obj04]) which has no foreseen profile support. Extending Ecore on level M_3 to achieve the same instantiation capabilities for profiles as in UML is not a desirable option, because this would demand for an extensive intervention with the current implementation of the standard EMF framework. Therefore, in EMF, our profile metamodel (*ProfileMM* in column *Profile Definition* of Figure 6.4) is defined at level M_2 and the user-defined profiles (*aProfile*) reside on M_1. As an unfortunate result, a defined stereotype in *aProfile* cannot be instantiated for representing stereotype applications (as in UML), because *aProfile* is already located on M_1 and EMF does not allow for *instantiating an instance* of a metamodel, i.e., EMF does not directly support multilevel modeling [AK01].

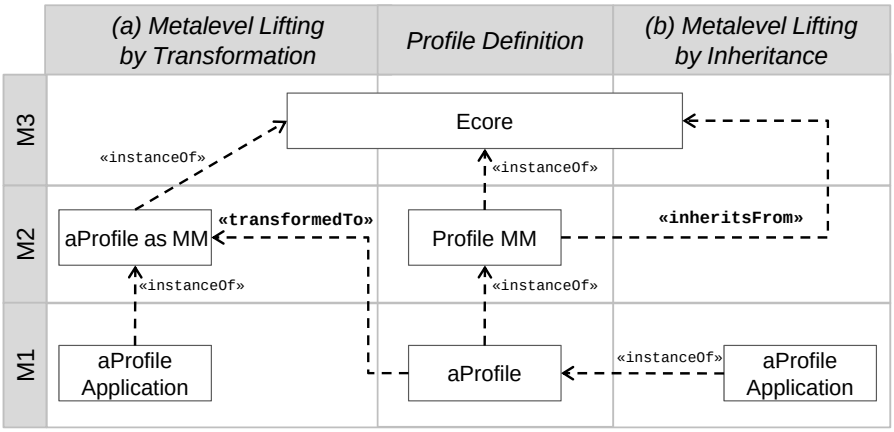

Figure 6.4: EMF Profile Architecture Strategies

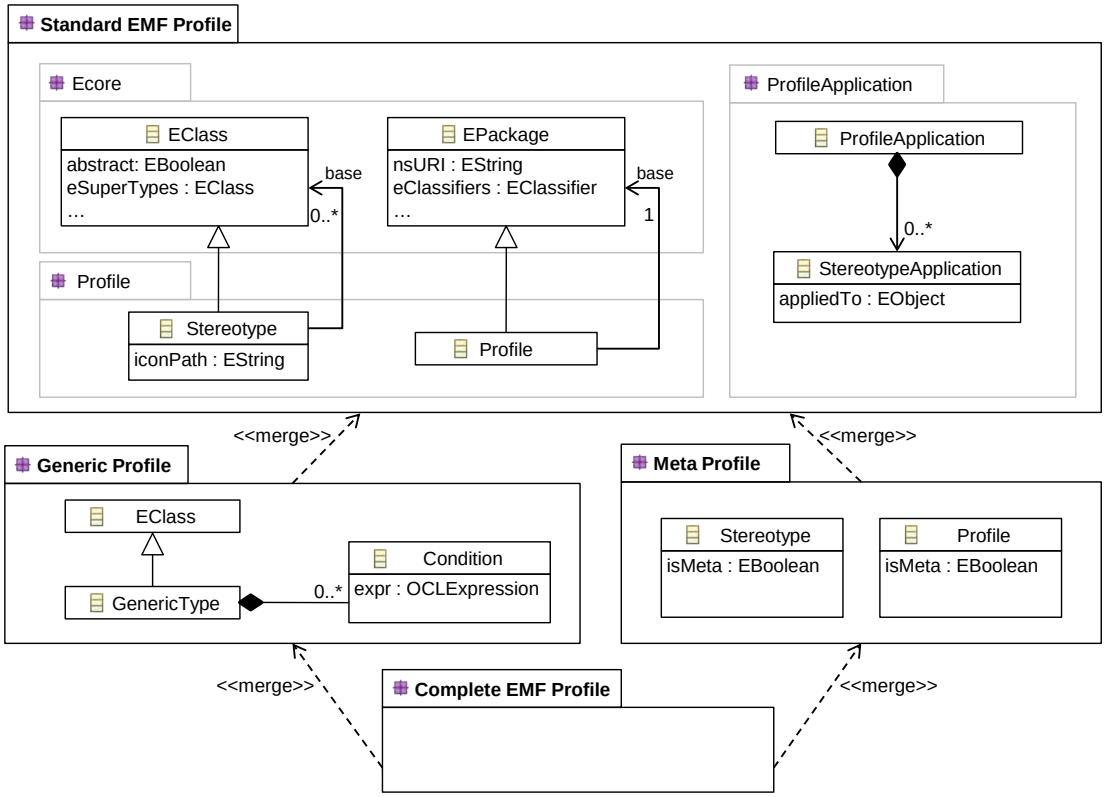

Figure 6.5: EMF Profile Metamodel

Therefore, more sophisticated techniques have to be found for representing stereotype applications in EMF. In particular, we identified two strategies for lifting *aProfile* from M_1 to M_2 in order to make it instantiable and directly applicable to EMF models.

(1) Metalevel Lifting By Transformation. The first strategy is to apply a model-to-model transformation which generates a metamodel on M_2, corresponding to the specified profile on M_1. The generated metamodel, denoted as *aProfile as MM* in the first column of Figure 6.4, is established by implementing a mapping from Profile concepts to Ecore concepts. In particular, the transformation generates for each `Stereotype` a corresponding `EClass` and for each `TaggedValue` a corresponding `EStructuralFeature`. The resulting metamodel is a direct instance of Ecore residing on M_2 and therefore, it can be instantiated to represent profile applications.

(2) Metalevel Lifting By Inheritance. The second strategy allows to *directly* instantiate profiles by *inheriting instantiation capabilities* (cf. «*inheritsFrom*» in the right column of Figure 6.4). In EMF, only instances of the meta-metaclass `EClass` residing on M_3 (e.g., the metaclass `Stereotype`) are instantiable to obtain an object on M_1 (e.g., a specific stereotype). Consequently, to allow for the direct instantiation of a defined stereotype on M_1, we specified the metaclass `Stereotype` in *Profile MM* to be a *subclass of* the meta-metaclass `EClass`. By this, a stereotype inherits EMF's capability to be instantiated and thus, a stereotype application

may be represented by a direct instance of a specific stereotype.

We decided to apply the second strategy, because of the advantage of using only one artifact for both, (1) defining the profile and (2) for its instantiation. This is possible because by this strategy, a profile is now a dual-faceted entity regarding the metalevels which is especially obvious when considering the horizontal «*instanceOf*» relationship between *aProfile* and *aProfileApplication* (cf. Figure 6.4). On the one hand, a profile is located on M_1 when considering it as an instance of the profile metamodel (*ProfileMM* on M_2)). On the other hand, the stereotypes contained in the profile are indirect instances of `EClass` and are therefore instantiable which means that a profile may also be situated on M_2. Especially, when taking the latter view-point, the horizontal «*instanceOf*» relationship between profile and profile application shown in Figure 6.4 will become the expected vertical relationship as in the UML metalevel architecture.

6.3.2 The EMF Profile Metamodel

The metamodel of the profile definition language is illustrated in package *Standard EMF Profile* of Figure 6.5. As a positive side effect of choosing the metalevel lifting strategy 2, the class `Stereotype` may contain, as being a specialization of `EClass`, also `EAttributes` and `EReferences`, which are reused to represent tagged values. Thus, no dedicated metaclasses have to be introduced to represent the concept of tagged values. Please note that stereotype applications also require having a reference to the model elements to which they are applied. Therefore, we introduced an additional metamodel package, namely *ProfileApplication* in Figure 6.5. This metamodel package contains a class `StereotypeApplication` with a reference to arbitrary `EObjects` named `appliedTo`. Whenever, a profile (instance of the *Profile* package) is saved, we automatically add `StereotypeApplication` as a superclass to each specified stereotype. To recall, this is possible because each `Stereotype` is an `EClass` which may have superclasses. Being a subclass of `StereotypeApplication`, stereotypes inherit the reference `appliedTo` automatically. In the following subsection, we further elaborate on the EMF Profile metamodel by providing a concrete example. Please note that the so far unmentioned packages *Generic Profile* and *Meta Profile* in Figure 6.5 are discussed in Section 6.3.3.

Applying the EMF Profile Metamodel

To clarify how profiles and profile applications are represented from a technical point of view, we make use of a small example. In particular, a simplified version of the well-known *EJB profile* is applied to an Entity-Relationship (ER) model [Che76]. Figure 6.6(a) depicts an excerpt of the ER metamodel and the EJB profile. The EJB profile contains the stereotypes `SessionBean` and `EntityBean`, which both extend the metaclass `Entity` of the ER metamodel. Besides, the profile introduces the stereotype `IDAttribute` extending the metaclass `Attribute` to indicate the ID of an `Entity`.

As already mentioned in the previous subsection, internally, we use the *ProfileApplication* metamodel (cf. Figure 6.6(b)) to weave the necessary concepts for a profile's application into a profile model. In particular, the class `ProfileApplication` acts as root element for all `StereotypeApplications` in a profile application model. Furthermore, all

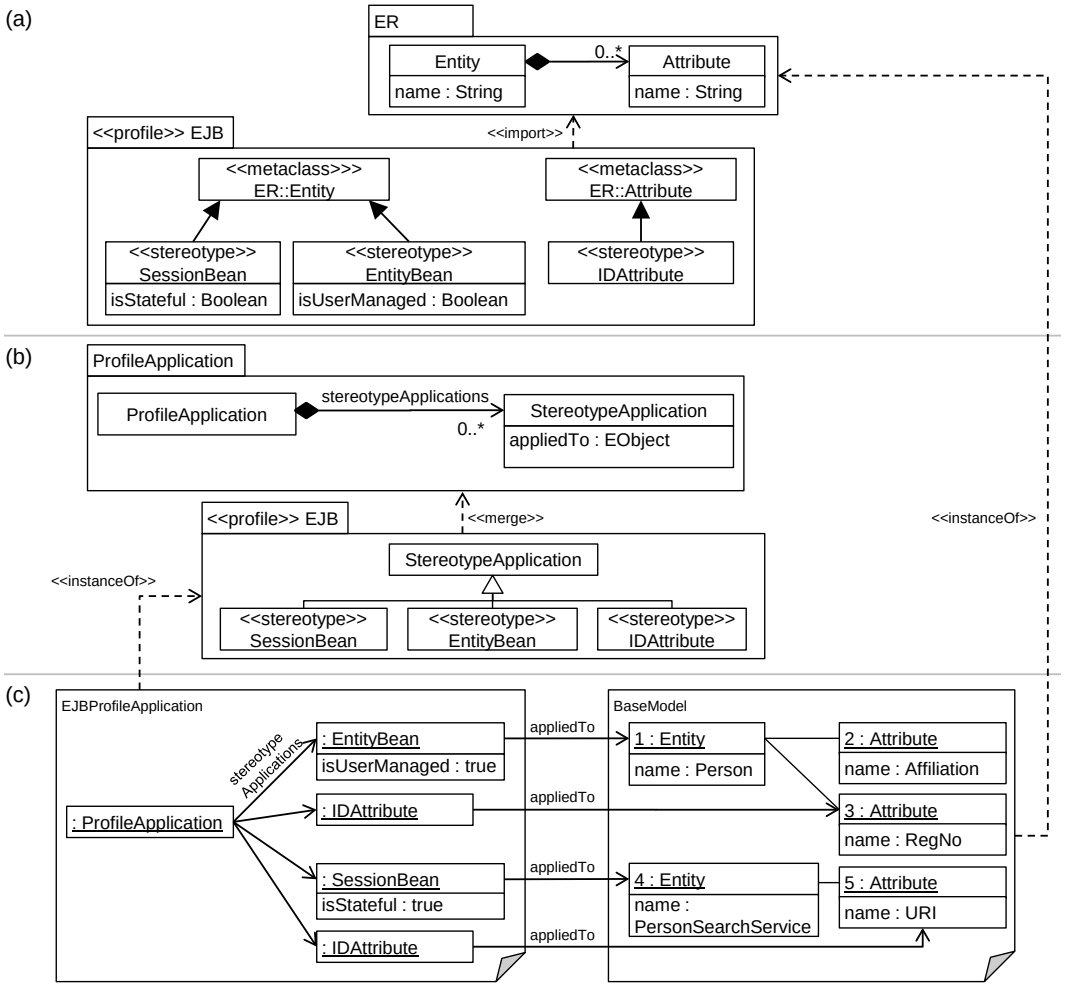

Figure 6.6: EMF Profiles by Example: (a) Profile definition user-view, (b) Internal profile representation, (c) Profile application

`Stereotypes` inherit the reference `appliedTo` from `StereotypeApplication`. When instantiating (i.e., applying) the EJB profile, a root element of the type `ProfileApplication` is created which may contain stereotype applications as depicted in Figure 6.6(c). For determining the applicability of a stereotype `s` to a particular model element `m`, it is checked whether the model element's metaclass (`m.eClass()`) is included in the list of metaclasses that are extended by the stereotype (`s.getBase()`). If so, the stereotype `s` is applicable to model element `m`. Each stereotype application is represented as a direct instance of the respective stereotype (e.g., «*EntityBean*») and refers to the model element in the *BaseModel* to which it is applied by the reference `appliedTo` (inherited from the class `StereotypeApplication`). Please note that the EJB profile application resides in a separated model file and not in the original ER model denoted with *BaseModel* in Figure 6.6.

6.3.3 Going Beyond UML Profiles

Originally, the profile mechanism has been specifically developed for UML. Hence, profiles may only extend the UML metamodel. In the previous section, we showed how this lightweight extension mechanism is ported to the realm of DSMLs. However, in this realm a whole pantheon of different DSMLs exists which are often concurrently employed in a single project. As a result, the need arises to reuse existing profiles and apply them to *several DSMLs*. Thus, we introduce two dedicated reuse mechanisms for two different scenarios:

(1) Metamodel-aware Profile Reuse. The first use case scenario is when users aim to apply a profile to a *specific set of DSMLs*. Being aware of these specific DSMLs' metamodels, the user wants to take control of the applicability of stereotypes to a manually selected set of metaclasses.

(2) Metamodel-agnostic Profile Reuse. In the second use case scenario, users intend to use a profile for *all DSMLs* without the need for further constraining the applicability of stereotypes. Therefore, a stereotype shall—agnostic of the DSMLs' metamodels—be applicable to every existing model element.

To tackle scenario (1), we introduce *generic profiles* allowing to specify stereotypes that extend so-called generic types. These generic types are independent of a concrete metamodel and may be bound to specific metaclasses in order to reuse the generic profile for several metamodels. For tackling scenario (2), we propose *meta profiles* which may immediately be applied to all DSMLs implemented by an Ecore-based metamodel.

Generic Profiles

The goal behind generic profiles is to reuse a profile specification for several "user-selected" DSMLs. Therefore, a profile should not depend on a specific metamodel. Inspired by the concepts of generic programming [MS89], we use the notion of so-called *generic types* instead. In particular, stereotypes within a generic profile do not extend concrete metaclasses as presented in the previous section, they extend generic types instead. These generic types act as placeholders for concrete metaclasses in the future. Once, a user decides to use a generic profile for a specific DSML, a binding is created which connects generic types to corresponding concrete metaclasses contained in the DSML's metamodel. For one generic profile there might exist an arbitrary number of such bindings. Consequently, this allows for reusing one generic profile for several DSMLs at the same time. Furthermore, it enables users to first focus on the development of the profile and reason about the relationship to arbitrary DSMLs in a second step.

As example, consider the same EJB profile which has been specified in terms of a concrete profile in Section 6.3.1. Now, we aim at specifying the same profile in a generic way to enable its use also for other DSMLs. In particular, we show how the EJB profile may first be specified generically and we subsequently illustrate the binding of this generic profile again for ER models. We get the same modeling expressiveness as before but now in a way that allows us to reuse the EJB profile when using other data modeling languages. The original EJB profile for ER extends two metaclasses, namely the stereotypes `SessionBean` and `EntityBean` extend the metaclass `Entity`, and the stereotype `IDAttribute` extends `Attributes` (cf. Figure 6.6). To turn this concrete profile into a generic one, we now use two generic types,

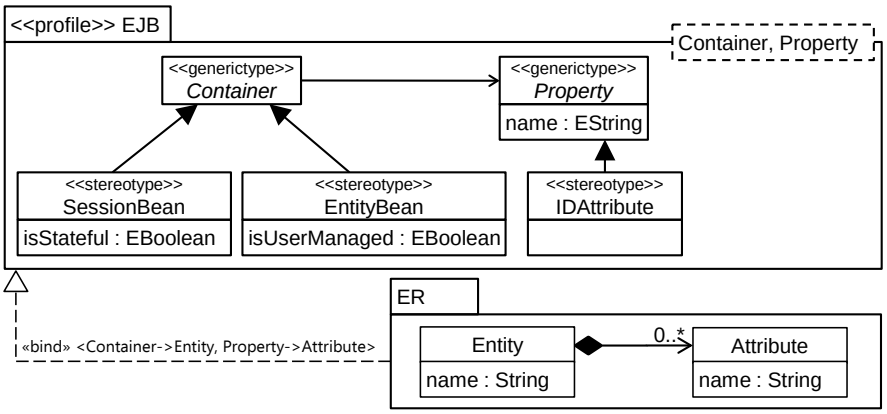

Figure 6.7: Generic EJB Profile and its Binding to the ER metamodel

named `Container` and `Property` in Figure 6.7, instead of the two concrete types `Entity` and `Attribute`.

Before we describe how generic profiles may be bound to concrete DSMLs, we first discuss conditions constraining such a binding. When developing a concrete profile, the extended DSML is known and consequently only suitable metaclasses are selected to be extended by the respective stereotypes. For instance, in the concrete EJB profile for ER, the class `Entity` may be annotated with the stereotype `EntityBean`. For marking the `Entity`'s ID attribute, the EJB profile introduces the stereotype `IDAttribute` which extends the class `Attributes`. This is reasonable, because we are aware of the fact that instances of the class `Entity` *contain* instances of the class`Attribute` in the ER metamodel, otherwise it obviously would not make any sense to extend the metaclass `Attribute` in this matter. However, generic profiles are developed without a concrete DSML in mind. Hence, profile designers possibly need to specify conditions enforcing certain characteristics to be fulfilled by the (up to this time) unknown metaclasses to which a generic type might be bound in future.

Therefore, EMF Profiles allows to attach conditions to generic profiles. Such conditions are specified by simply adding references or attributes to generic types. This is possible because, as a subclass of `EClass`, generic types may contain instances of `EReference` and `EAttribute`. By adding such a reference or attribute in a generic type, a profile designer states that there must be a corresponding reference or attribute to the metaclass which is bound to the generic type. Internally, these references and attributes are translated to OCL constraints which are evaluated in the context of the metaclass a user intends to bind.

Furthermore, the profile designer must specify which meta-features, such as the cardinality of the reference or attribute in a generic type, shall be enforced. In our example in Figure 6.7, the profile designer specified a reference from the generic type `Container` to `Property` as well as an attribute `name` in `Property`. To enforce this, the OCL constraints in Listing 6.1 are generated. These constraints must be satisfied by each metamodel on which we want to apply this profile on.

Listing 6.1: OCL Constraints generated for `Container` and `Property`

```
1 context Container inv:
2 self.eReferences->exists(r | r.eType = Property)}
3 context Property inv:
4 self.eAttributes->exists(a | a.name = "name" and a.eType = EString)
```

Once the stereotypes and generic types are created, the profile is ready to be bound to concrete DSMLs. This is simply achieved by selecting suitable metaclasses of a DSML for each generic type. In our example depicted in Figure 6.7, the generic types `Container` and `Property` are bound to the metaclasses in the ER metamodel `Entity` and `Attribute`, respectively, in order to allow the application of the generic EJB profile to ER models. When the binding is established, it can be persisted in two different ways. The first option is to generate a concrete profile out of the generic profile for a specific binding. This concrete profile may then be applied like a normal EMF profile as discussed in Section 6.3.1. Although this seems to be the most straightforward approach, the explicit trace between the original generic profile and the generated concrete profile is lost. Therefore, the second option is to persist the binding directly in the generic profile definition. Whenever a user intends to apply a generic profile to a concrete DSML, the EMF Profile framework searches for a persisted binding for the concrete DSML's metaclasses within the profile definition. If a binding exists, the user may start to apply the profile using this persisted binding. Otherwise, the user is requested to specify a new binding.

To support generic profiles, we extended the EMF Profile metamodel by the class `GenericType` (cf. Fig 6.5). Generic types inherit from `EClass` and may contain `Conditions` representing more complex constraints going beyond the aforementioned enforced references and attributes for bound metaclasses.

Meta Profiles

With meta profiles we tackle a second use case for reusing profiles for more than one DSML. Instead of supporting only a manually selected number of DSMLs, with meta profiles we aim at reusing a profile for *all* DSMLs without the need of defining an explicit extension for each DSML. This is particularly practical for profiles enabling general annotations which are suitable for every DSML. In other words, stereotypes within a meta profile must be agnostic of a specific metamodel and shall be applicable to *every model element* irrespectively of its metaclass, i.e., its type.

In EMF, every model element is an instance of a metaclass. Each metaclass is again an instance of Ecore's `EClass`. Therefore, meta-stereotypes in a meta profile do not extend metaclasses directly. Instead, they are configured to be applicable to *all instances of instances* of `EClass` and, consequently, to every model element (as an instance of an instance of `EClass`). This approach is inspired by the concept of potency known from multilevel metamodeling [AK01]. Using the notion of potency, one may control on which metamodeling level a model element may be instantiated. By default, the potency is 1 which indicates that a model element may be instantiated in the next lower metamodeling level. By a potency $p \geq 1$ on a metamodeling level n, a model element may be configured to be also instantiable on the level $n - p$ instead of the next lower level only. In terms of this notion of potency, a meta-stereotype has a potency of $p = 2$.

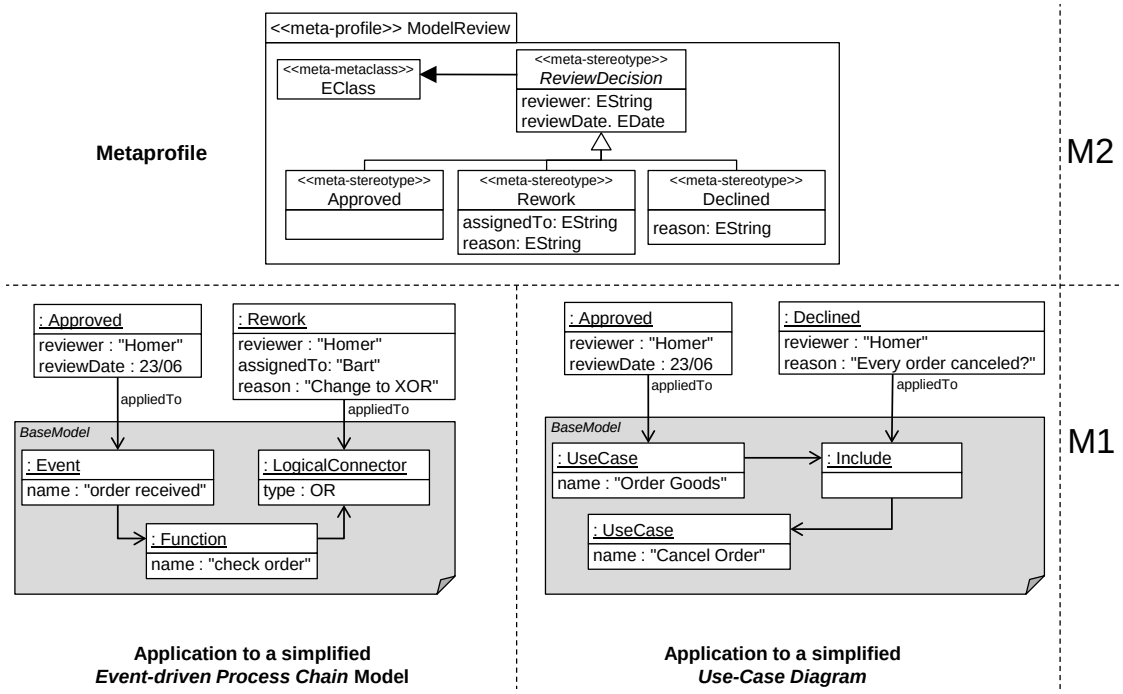

Figure 6.8: Meta Profile Example: The Model Review Profile

Meta profiles are created just like normal profiles. However, a new attribute, namely `isMeta`, is introduced to the profile metamodel for indicating whether a stereotype is a meta-stereotype (cf. Figure 6.5). The Boolean value of this attribute is regarded by EMF Profiles when evaluating the applicability of stereotypes. In particular, if `isMeta` is `true`, a stereotype is always considered to be applicable to every model element, irrespectively of its metaclass.

Our example for presenting meta profiles is a *model review profile* (cf. Figure 6.8). The goal of this profile is to allow for annotating the results of a systematic examination of a model. Since every model irrespectively of its metamodel can be subject to a review, this profile is suitable for every DSML. For simplicity, we just introduce three stereotypes in the review profile, namely `Approved`, `Rework`, and `Declined`, which shall be applicable to every kind of element in every DSML. Therefore, these three stereotypes extend the class `EClass` and are marked as meta-stereotypes (indicated by `meta-stereotype` in Figure 6.8). By this, the applicability of these stereotypes is checked by comparing the meta-metatypes of model elements with the metaclasses extended by the stereotypes. As a result, the metaprofile in our example is applicable to every element in every DSML.

In the example shown in Figure 6.8, we depicted the Object Diagram of two separate applications of the same metaprofile to two models conforming to different metamodels. In the first Object Diagram, an `Event` and one `LogicalConnector` within an Event-driven Process Chain (EPC) model have been annotated with the meta-stereotype `Approved` and `Rework`, respectively. This is possible because both instances in the EPC model are instances of a meta-

class which is again an instance of `EClass`. The same meta profile may be also applied to any other modeling language. Of course, also UML itself is supported by EMF Profiles. Therefore, the model review profile may be also applied to, for example, a UML Use Case Diagram (cf. Figure 6.8). In this figure, the stereotype `Approved` has been assigned to the `UseCase` named "`Order Goods`" and the stereotype `Declined` has been applied to the `Include` relationship.

Summary

Both techniques for enabling the reuse of profiles for several DSMLs have their advantages and disadvantages depending on the intended use case. Meta profiles are immediately applicable to all DSMLs without further user intervention. However, with meta profiles no means for restricting the use of such profiles for concrete DSMLs exist. If this is required, generic profiles are the better choice. When specifying generic profiles, explicit conditions may be used to control a profile's usage for concrete DSMLs. On the downside, this can only be done with additional efforts for specifying such conditions in the generic profile and creating manual bindings from generic profiles to concrete DSMLs.

6.3.4 A Tour on EMF Profiles

In this section, we present our prototypical implementation of EMF Profiles which is realized as Eclipse plug-in on top of the Eclipse Modeling Framework and Graphical Modeling Framework[11] (GMF). Please note that we refrained from modifying any artifact residing in EMF or GMF. EMF Profiles only uses well-defined extension points provided by these frameworks for realizing profile support within the EMF ecosystem. For a screencast of EMF Profiles, we kindly refer to our project homepage[12].

Profile Definition. To define a profile, modelers may apply either the tree editor automatically generated from the Profile Metamodel or our graphical EMF Profiles Editor, which is realized with GMF (cf. Figure 6.9 for a screenshot). The graphical notation used in this editor takes its cue from the UML Profiles syntax.

With these editors, modelers may easily create stereotypes containing tagged values and set up inheritance relationships between stereotypes and extension relationships to metaclasses of arbitrary DSML's metamodels. Metaclasses may be imported by a custom popup menu entries when right-clicking the canvas of the editor and are visualized using the graphical notation from Ecore.

Profile Application. Defined profiles may also be applied using any EMF-generated tree-based editor or any GMF-based diagramm editor. The screenshot depicted in Figure 6.10, shows the afore presented EJB profile applied to an example Ecore diagram. To apply profiles, our plugin contributes a popup menu entry (cf. Figure 6.10 (1)) which appears whenever a model element is right-clicked. By this menu, users may apply defined profiles (i.e., creating new profile application) or import already existing profile applications. Once a profile application is created

[11] `http://www.eclipse.org/gmf`
[12] `http://www.modelversioning.org/emf-profiles`

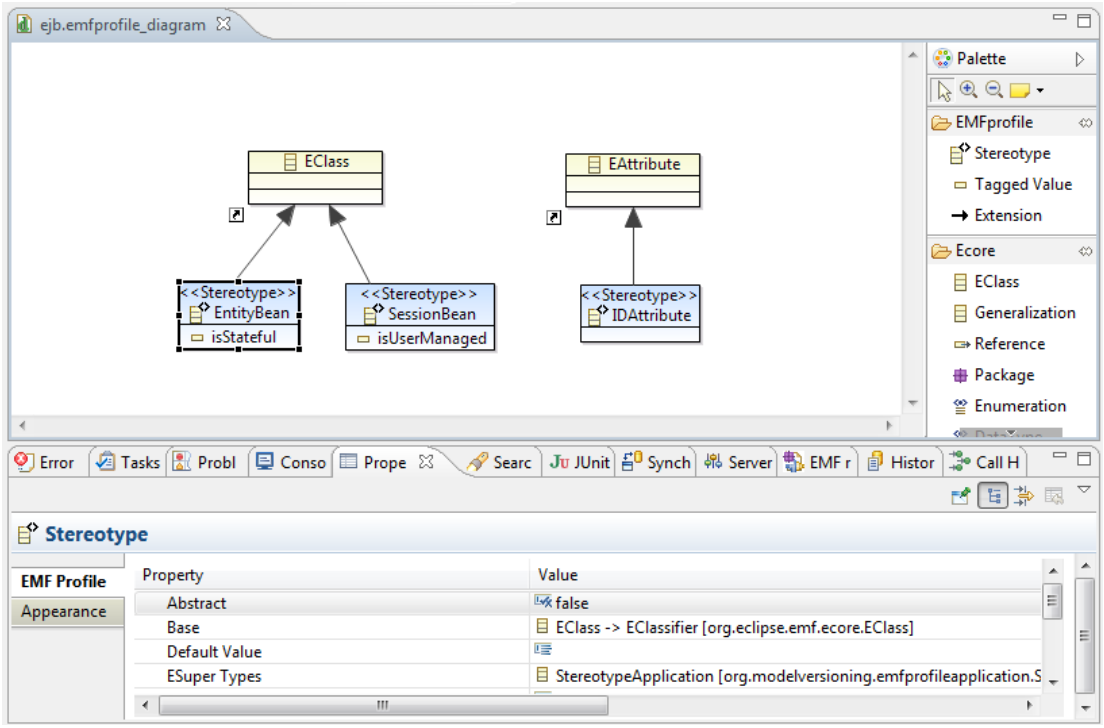

Figure 6.9: EJB Profile Defined on Ecore with Graphical EMF Profiles Editor

or imported, stereotypes may be applied using the same popup menu. When a stereotype is applied, the defined stereotype icon is attached to the model element (cf. Figure 6.10 (2)). For this purpose we used the GMF Decoration Service, which allows to annotate any existing shapes by adding an image at a pre-defined location. Furthermore, we created a Profile Applications view, which shows all applied stereotypes of the currently selected model element (cf. Figure 6.10 (3)). The currently selected model element is retrieved using the `ISelectionProvider` interface which is implemented by every EMF or GMF-based editor. For assigning the tagged values of an applied stereotype, we leverage the `PropertyView` (cf. Figure 6.10 (4)) which generically derives all defined tagged values from the loaded profile's metamodel. The separate file resource which contains the profile applications is added to the `EditingDomain` of the modeling editor. Hence, as soon as the model is saved, all profile applications are saved as well. Finally, profile applications can be unloaded and reloaded at any time without loosing the application information.

6.4 Merging Models in AMOR

We adapted the notion of UML Profiles to the realm of DSMLs residing in the Eclipse Modeling Framework. Using our prototype EMF Profiles, DSMLs may be easily extended in a non-invasive manner by defining profiles in the same way as done in UML tools. Moreover, we

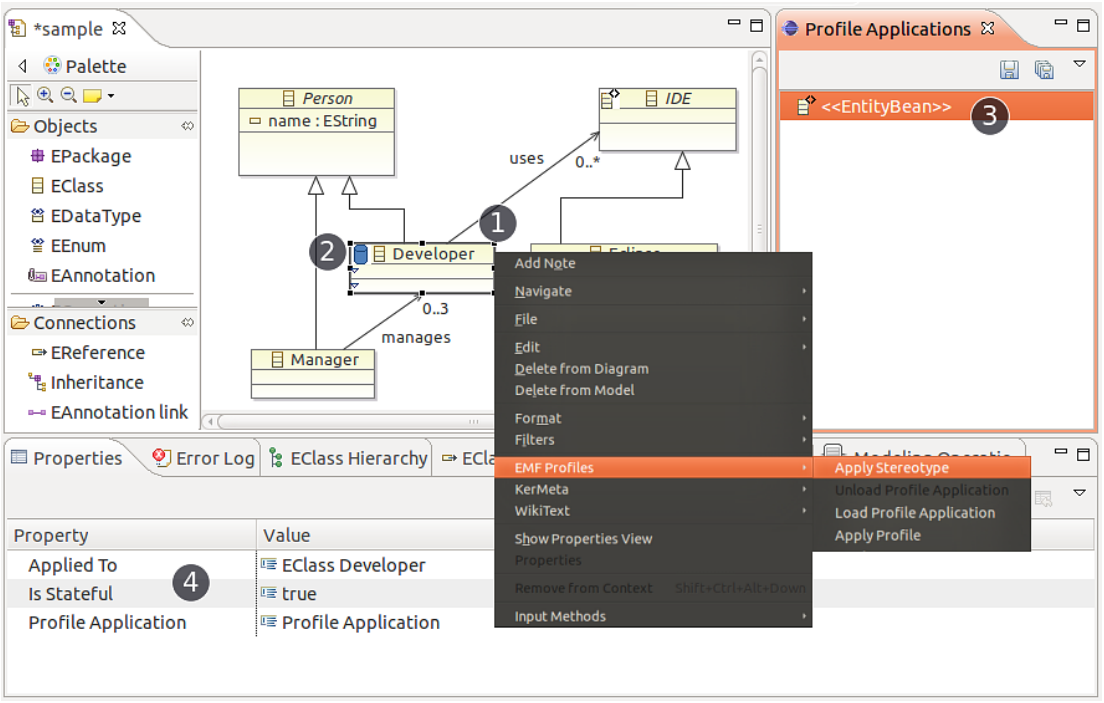

Figure 6.10: EJB Profile Applied to Ecore Instance

introduced two novel mechanisms, namely *Generic Profiles* and *Meta Profiles*, for reusing defined profiles with several DSMLs. Although, the presented approach has been presented based on EMF, the general procedure is also applicable for other metamodeling frameworks which comprise a similar metalevel architecture as EMF. Furthermore, the presented metalevel lifting strategies may also be adopted for other scenarios in which model elements on M_1 need to be instantiated.

We are applying EMF Profiles in the context of our model versioning system AMOR to support the Conflict-tolerant Merge as presented in this thesis. We use EMF Profiles for marking conflicts caused by concurrent changes of the same model artifact using a *conflict profile* and for adding metadata to the involved model elements. The profile has been defined as *meta profile*, which is generically applicable, i.e., independent of the used modeling languages. Furthermore, it is based on the conflict annotation model as depicted in Figure 5.11 and the resolution model as depicted in Figure 5.13. Coming back to the running example introduced in Chapter 5, a screenshot of Eclipse consisting of the merged model is depicted in Figure 6.11, in which the parallel versions of Harry, Sally, and Joe are merged into one version. The occurred conflicts are automatically annotated using EMF Profiles and are visualized with dedicated conflict icons. As already mentioned in the previous section, the icons can be defined for each stereotype in the profile model. If a user selects an annotated model element, the applied conflict annotations are presented in the "Profiles Applications" view. In the screenshot, the Update/Update conflict applied on the attribute `bday` of the class `employee` is selected. At the bottom of this screenshot,

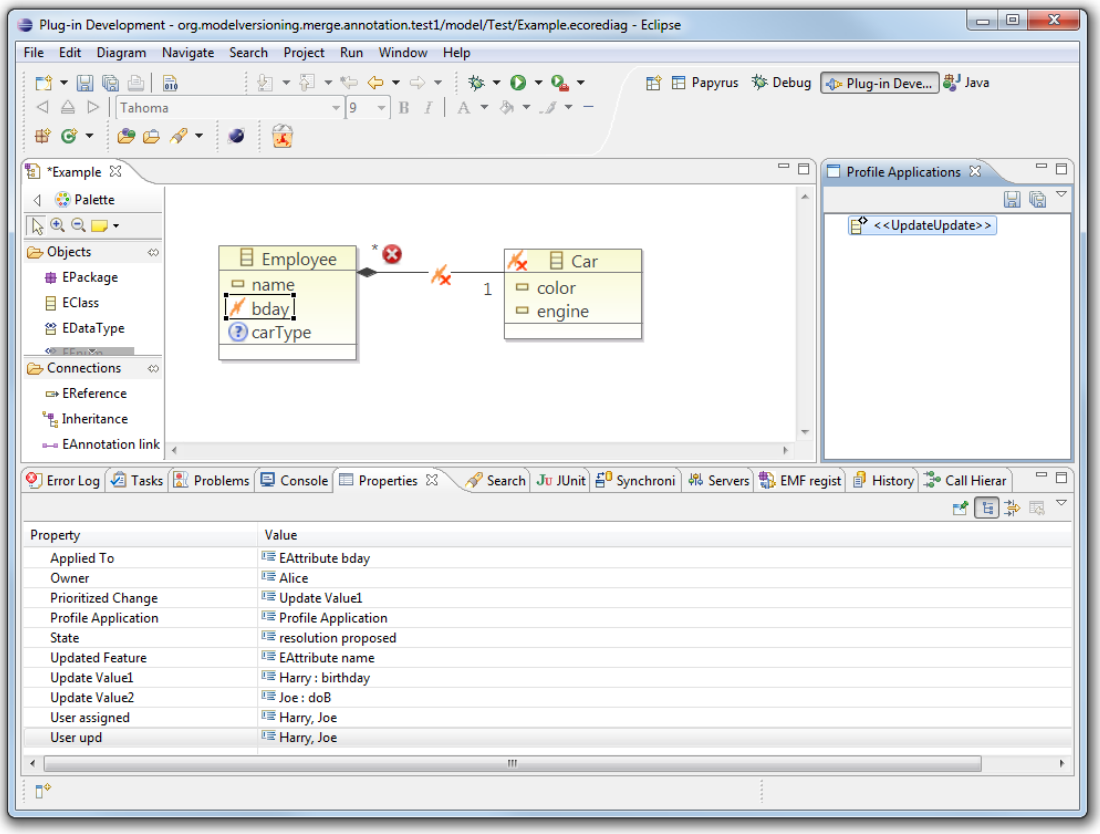

Figure 6.11: Conflict-tolerant Merge as Eclipse Plug-in

the tagged values of this stereotype are presented in terms of properties. The users may now see that Harry and Joe have concurrently renamed the attribute to `birthday` and `doB`, respectively. Both users are also assigned to this conflict and have chosen to prioritize the update of Harry. Thus, the status of the conflict is set on `resolution proposed`.

Using the EMF Profiles API for Annotating Conflicts

In contrast to the EJB Profile example presented in the section before, the stereotypes defined in the conflict profile are automatically applied by the CTAnnotator of the Conflict-tolerant Merge using the API of EMF Profiles. EMF Profiles is shipped with a dedicated API, alongside the graphical user interfaces presented above to ease the programmatic application of profiles. In the following, we briefly present the usage of this API in terms of an example. In this example, we show how the Conflict-tolerant Merge annotates an Update/Update conflict after applying the dedicated merge rule presented in Section 5.2.2. The corresponding Java code is depicted in Listing 6.2. In line 2 of this listing, the resource containing the model to be annotated is loaded. Before we may apply the *conflict profile* to specific conflicts of the conflict report, we first have to load the resource containing the corresponding profile definition (cf. line 8) and retrieve the

instance of `Profile` representing the *conflict profile* from this resource (cf. line 9). To ease the application of stereotypes, EMF Profiles provides the interface `IProfileFacade` and an implementation of this interface called `ProfileFacadeImpl`, which is instantiated in line 15. Next, we specify a resource to which the profile application shall be saved and load the profile to be applied using the facade (cf. line 16–17). Please note that users may load multiple profiles to be applied at the same time with one profile facade instance. In the next step, we assign stereotypes to variables. In our case, the stereotype "`UpdateUpdate`" of the conflict profile is assigned to `updUpdStereo` (cf. line 20). Now, we are set up to apply stereotypes to conflicts. Therefore, the conflict report is iterated through and when an instance of an `UpdateUpdate` conflict is found (cf. line 20-22), we may apply a dedicated stereotype and its tagged values as follows. Using the method `isApplicable(updateUpdate, originElement)` of the facade, we may check whether the stereotype `UpdateUpdate` is applicable to the specified model element in the origin model (cf. line 30). If this is the case, we may use the facade to apply the stereotype using the method `apply(updateUpdate, originElement)` as depicted in line 31. This method returns the created instance of stereotype application called `application`. The facade may also be used to retrieve all features (i.e., tagged values) of a stereotype (cf. line 36) as well as to assign specific values for them (cf. line 43). In our case, the update value of one parallel version (e.g., of User A) is assigned to a tagged value of the Update/Update stereotype.

In [PWWZ11], we proposed conflict-tolerant model versioning for supporting collaboration in cross-organizational modeling. This use case scenario is presented in the following.

Use Case: Collaborative Business Document Modeling

Business documents, also serialized in XMI like software models, are typically defined through Standard Developing Organizations (SDOs) such as the United Nations Centre for Trade Facilitation and eBusiness (UN/CEFACT). In today's highly dynamic environment with ever-changing market demands, SDOs are confronted with the need to constantly evolve their standardized business documents based on the needs of business partners utilizing these documents. However, the business document development process between SDOs and business partners is currently lacking efficient collaborative support that is described in more detail in [PWWZ11]. Thus, we extended the conflict-tolerant merge approach by the means of a reference model supporting hierarchical collaborative cross-organizational business document modeling, and adapted the conflict resolution model depicted in Figure 5.13, p. 84 to find a consolidated version of a new business document model.

(i) Reference Model for Collaborative Cross-organizational Modeling. Our reference model is designed for collaborative scenarios meeting the following two characteristics, which are prescribed by the UN/CEFACT. First, the reference model addresses a *cross-organizational* aspect meaning that the different stakeholders involved in the collaborative process are spread across different organizations and institutions. Second, the stakeholders form a *hierarchy*, i.e., one stakeholder may overrule decisions of another stakeholder involved in the same development process. Based on these needs, we define a generic reference model supporting hierarchical collaborative cross-organizational modeling, as illustrated in Figure 6.12. The reference model

Listing 6.2: Code Excerpt for Annotating Update/Update Conflicts

```
1  // Load the model to be annotated
2  Resource modelResource = ...;
3
4  // Detect conflicts
5  ConflictReport conflictReport =...;
6
7  // Load the profile
8  Resource profileResource = ...;
9  Profile conflictProfile = (Profile) profileResource.getContents().get(0);
10
11 // Create the resource that contains the profile application
12 Resource profileApplicationResource = ...;
13
14 // Initialize the profile facade
15 IProfileFacade profileFacade = new ProfileFacadeImpl();
16 profileFacade.setProfileApplicationResource(profileApplicationResource);
17 profileFacade.loadProfile(conflictProfile);
18
19 // Assign stereotypes to variables
20 Stereotype updUpdStereo = profile.getStereotype("UpdateUpdate");
21 ...
22
23 EList<Conflict> conflicts = conflictReport.getConflicts();
24 for (Conflict conflict : conflicts) {
25   if (conflict instanceof UpdateUpdate) {
26     UpdateUpdate updateUpdate = (UpdateUpdate) conflict;
27     DiffElement change = updateUpdate.getLeftChange();
28     EObject originElement = DiffUtil.getLeftElement(change);
29
30     // Apply stereotype
31     Stereotype updateUpdate = conflictProfile.getStereotype("UpdateUpdate");
32     StereotypeApplication application = null;
33     if (profileFacade.isApplicable(updateUpdate, originElement)) {
34       application = profileFacade.apply(updateUpdate, originElement);
35     }
36
37     // Set tagged values
38     EStructuralFeature updUpdFeature = DiffUtil.getUpdatedFeature(change);
39     EAttribute leftValueFeature = updUpdStereo.getEAttributes().get(0);
40     // Value updated by User_A
41     String updValue = DiffUtil.getLeftElement(change)
42               .eGet(updUpdFeature).toString();
43     // Set tagged value for saving the update value
44     profileFacade.setStereotypeApplicationFeatureValue(application,
45               leftValueFeature, updValue);
46     ...
47
48   }
49 }
```

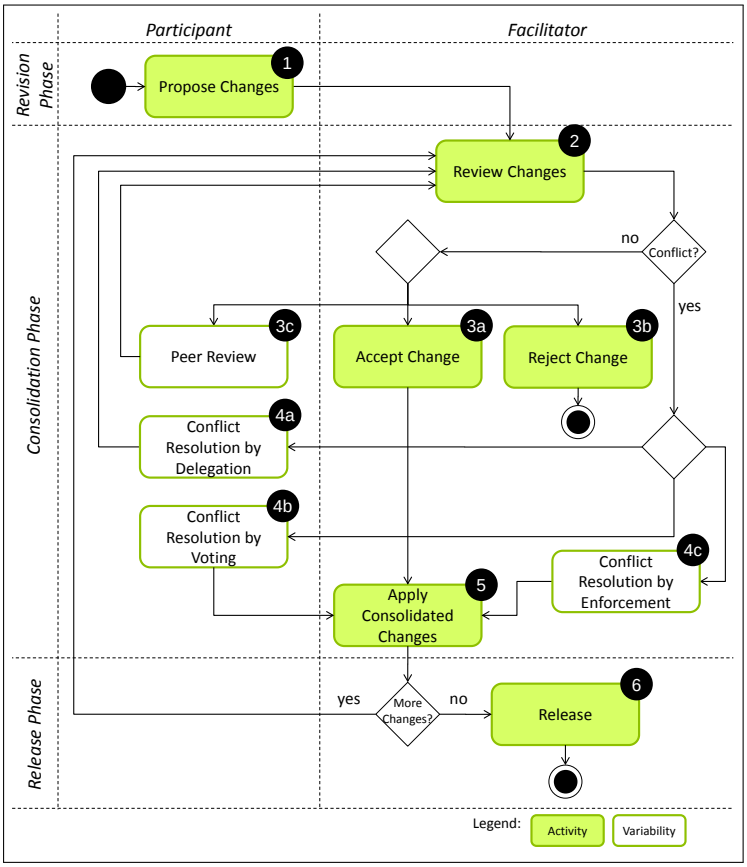

Figure 6.12: Reference Model for Collaborative Cross-organizational Modeling

describes a generic workflow as well as offers variability aspects for customizing the workflow for a particular collaboration scenario. Generally speaking, the workflow comprises three different phases, namely the *Revision* phase, the *Consolidation* phase, as well as the *Release* phase. Furthermore, two different stakeholders are involved in the different phases whereas each stakeholder takes on a particular role. The roles defined in our reference model are *Participant* as well as *Facilitator*, forming a hierarchical relationship. In other words, the *Participant* may propose changes to a particular model and the *Facilitator* reviews the proposed changes and decides whether the changes are applied to the model.

Revision Phase. Throughout the *Revision* phase, different *Participants* may propose changes for a model in parallel and independently from each other (cf. Mark 1). Business partners would take on the role of *Participant* and propose changes to business document model contained in the model repository.

Consolidation Phase. At a given point in time, the *Facilitator* brings the *Revision* phase to an end and initiates the *Consolidation* phase. In this phase, the *Facilitator* reviews the proposed changes, indicated through *Review Changes* (cf. Mark 2). Applied to our example, UN/CEFACT takes on the role of the *Facilitator*.

In case the proposed change is not conflicting with any other changes, the *Facilitator* decides whether to *Accept* (cf. Mark 3a), *Reject* (cf. Mark 3b), or *Peer Review* (cf. Mark 3c) a particular change. In case the change is *Accepted* it is then ready to be incorporated into the model. However, the *Facilitator* may as well decide to *Reject* a particular change. Furthermore, the *Facilitator* may want to discuss the proposed change with the *Participant*, indicated through the activity *Peer Review* which represents the first variability aspect within our reference model. This means that the detailed workflow within the activity can be customized to fit the requirements of a particular scenario. For example, in the *Peer Review* activity, the *Participant* may then either accept or reject the *Facilitator*'s alternative, as well as suggest another alternative to the *Facilitator*.

However, it may occur that two different *Participants* propose changes resulting in a conflict. In this case we propose three options for handling conflicting changes including *Conflict Resolution by Delegation* (cf. Mark 4a), *Conflict Resolution by Voting* (cf. Mark 4b), as well as *Conflict Resolution by Enforcement* (cf. Mark 4c). All three options for resolving conflicts represent further variability aspects of our reference model and could be extended in future. In pursuing the first option, the *Facilitator* resolves the conflicting changes and makes a decision, which may overrule change requests of the *Participants*. In utilizing the second option, i.e., *Conflict Resolution by Delegation*, the *Facilitator* does not influence the decision process, but leaves the process of resolving the conflict up to the *Participants*. For instance, in a customized reference model fitting a certain business scenario, *Participants* may utilize synchronous collaboration techniques for resolving conflicts. In the third option, the *Facilitator* provides several alternatives for resolving the conflict to the *Participants*. The *Participants* may then vote for a particular conflict resolution. After completing the second or third option, the Facilitator reviews the outcome of the conflict resolution strategy. In case the pursued conflict resolution strategy resulted in another conflict, a new review cycle is started (cf. Mark 2).

Regardless, whether a change has been accepted at the very beginning, has been peer reviewed, or has been resolved following a particular conflict resolution strategy, once an agreement between the *Facilitator* and the *Participant* is found, the consolidated change is incorporated into the model (cf. Mark 5).

Release Phase. Assuming that all changes are consolidated, the *Facilitator* introduces the *Release* phase. In this phase, a new, consolidated, version of the model is released (cf. Mark 6).

(ii) Conflict Resolution. Following UN/CEFACT's approach, business partners may request changes to the business document models. As elaborated on earlier, it may occur that different business partners submit change requests resulting in a conflict. For instance, consider the example scenario illustrated in Figure 6.13, which is an excerpt of the running example introduced in Section 5.1. It is assumed, that the business document model provided by UN/CEFACT contains two classes, namely again, `Employee` as well as `Car` having different attributes. Due to changing market requirements and evolving business needs, business partners are confronted with the need to update their business documents often requiring to update the underlying metamodel. Following UN/CEFACT's approach, business partners have to submit their change requests to UN/CEFACT. For instance, as illustrated in Figure 6.13, *Business Partner A* proposes to change the attribute `bday` to `birthday` resulting in an intermediary version *V1a*. At the same time,

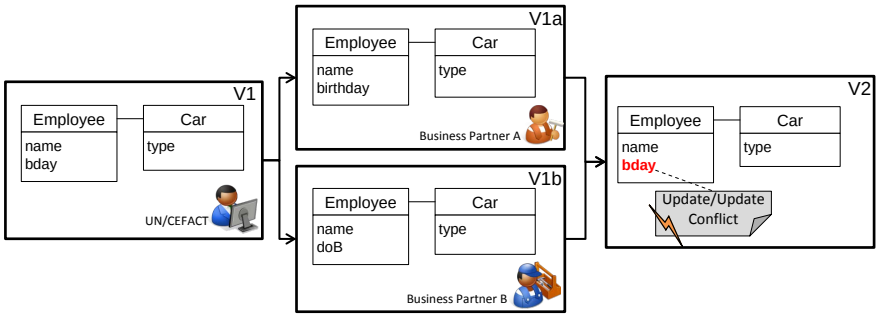

Figure 6.13: Conflict Example

Business Partner B proposes changing the same attribute from `bday` to `doB` resulting in version *V1b*. Consequently, UN/CEFACT reviews the change requests and plans to release a new version thereof. However, as illustrated in Figure 6.13, the changes proposed by both business partners result in an *Update/Update* conflict since the same attribute is renamed differently.

As discussed earlier, our reference workflow model provides three different options for resolving conflicts. For all three options, we present a *Conflict Resolution Model* defining the relevant information about a concrete conflict resolution. A generic conflict resolution model is presented in 5, which we have adopted and extended in this section to satisfy the needs of this use-case. In particular, one option of the reference workflow is to involve the business partners themselves in resolving the conflict represented through *Conflict Resolution by Delegation*. This option has the advantage, that business partners may discuss changes amongst themselves for reaching an ideal agreement fitting the requirements of both business partners. Therefore, we firstly present the Conflict Resolution Model supporting cross-organizational modeling and, secondly, we demonstrate the *Conflict Resolution by Delegation* pattern based on the example presented above.

Adopted Conflict Resolution Model. For supporting the consolidation phase, we have de-

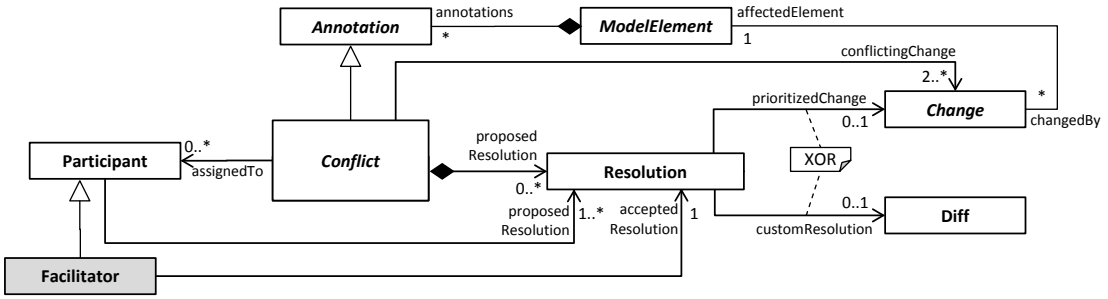

Figure 6.14: Adopted Conflict Resolution Model

veloped a dedicated model defining the relevant information about a concrete conflict resolution. The resulting *Conflict Resolution Model* is depicted in Figure 6.14. A `ModelElement` may be annotated by a `conflict`. A `conflict` links two conflicting `changes` and may be as-

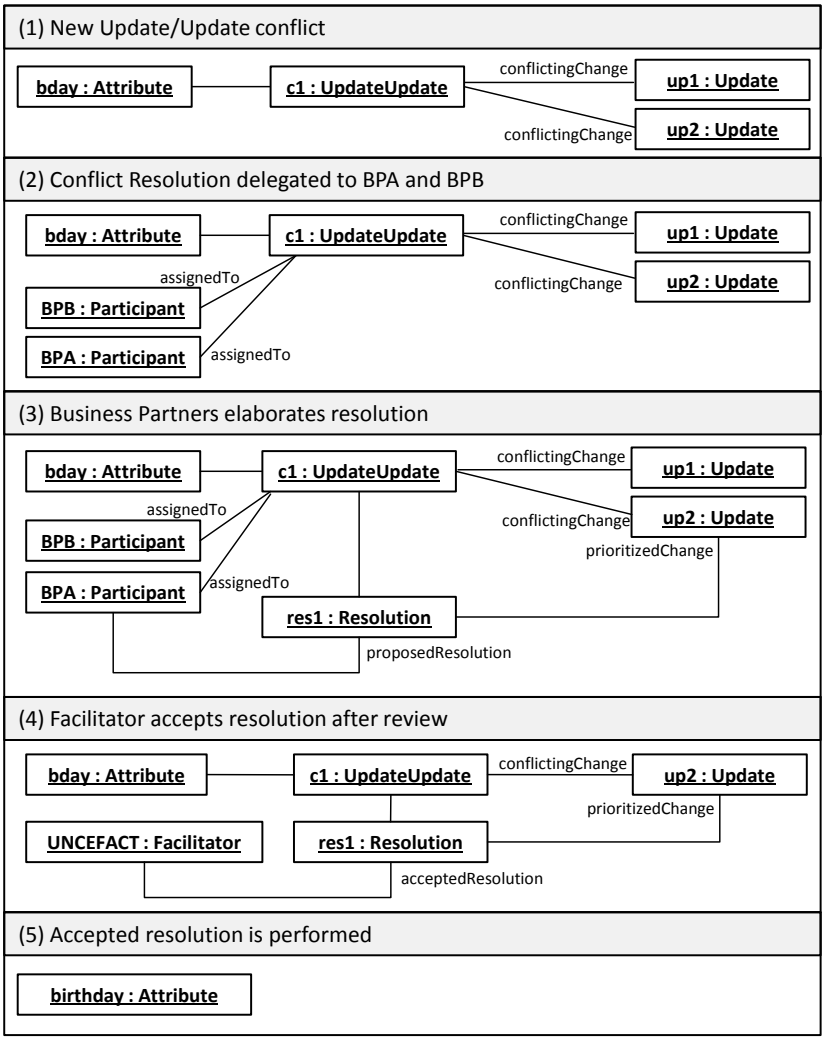

Figure 6.15: Conflict Resolution by Delegation

signed to different `participants`, i.e., business partners, which are responsible to resolve the conflict. In case multiple conflicts exist for the same `ModelElement`, the `ModelElement` is annotated with multiple conflicts. These participants may propose different `resolutions`, but exactly one of these resolutions has to be finally accepted by the `facilitator` in order to resolve the conflict and, furthermore, to apply the consolidated changes. No matter which consolidation strategy is chosen, two kinds of concrete resolution possibilities exist: (1) either select *one* out of the conflicting changes, or (2) discard both and perform a custom resolution, which may contain several changes. In the latter case, the modeled resolution is stored as its own `Diff` to comprehend afterwards what happened to the conflict in the resolution process.

Conflict Resolution by Example. In Figure 6.15, we present the *Conflict Resolution by Delegation* process on the basis of the example presented before. Please note, that due to read-

ability, only the most important relationships are illustrated. In this concrete example, a conflict occurred due to concurrent changes of the attribute `bday`. Again, Business Partner A (`BPA`) has renamed the attribute to `birthday`, whereas Business Partner B (`BPB`) has renamed the same attribute to `doB` leading to a so-called `Update/Update` conflict. The facilitator decides to delegate the resolution of this conflict to both business partners and, thus, `BPA` and `BPB` are assigned to the `Update/Update` conflict to collaboratively propose a resolution. They decide to prioritize the update of `BPA`, i.e. `up2`. The facilitator may now accept or reject the proposed resolution. In our example, the proposed resolution, i.e., the rename of the attribute `bday` to `birthday`, is accepted and, thus, the change is applied to the model.

6.5 Summary

In this chapter, we have presented the architecture of the Conflict-tolerant Merge and gave insights of the CTMerger process. One of the major components of the Conflict-tolerant Merge is the light-weight model annotation mechanism called EMF Profiles, of which we presented the design principles and rationale behind it in more detail. Furthermore, EMF Profiles also provides advanced reuse mechanisms, such as meta profiles, which are immediately applicable to all DSMLs without further user intervention. The Conflict-tolerant Merge uses such a meta profile to automatically annotate occurred conflicts when merging parallel versions of any Ecore-based model. These conflicts are additionally visualized with dedicated icons defined in the conflict profile.

Finally, we presented a use case scenario, which underlined the need for the Conflict-tolerant Merge in the context of cross-organizational modeling of business documents. In the next chapter, we present the evaluation of our approach in terms of a quasi-experimental study.

CHAPTER 7

Evaluation

In this chapter, we report on our initial evaluation of the Conflict-tolerant Merge presented in this thesis in comparison with EMF Compare[1], the state-of-the-art tool for three-way model comparison and model merging for EMF-based models. We decided to use EMF Compare, because it is part of the Eclipse Modeling Project[2] and freely available. We have done a quasi-experimental study [CSG63] with 18 participants and interviewed them afterwards with the help of a questionnaire. In the following, we describe the study in detail, discuss the challenges encountered by the participants, and present the results and our findings of this evaluation. The evaluation is based on observations and valuable feedback from our participants.

7.1 General Setting

As already discussed in previous chapters, the Conflict-tolerant Merge follows a new paradigm for developing models in teams. It offers an alternative to the traditional versioning paradigm, in which the developer who commits the changes is solely responsible for resolving the detected conflicts immediately. To simulate this iterative process, we used EMF Compare (version 1.1.2), which offers a tree-based representation of two versions of the model and correspondences between the detected conflicts. In contrast, the Conflict-tolerant Merge incorporates all changes of all developers, marks occurred conflicts with the help of EMF Profiles and at a later point of time the conflicts are resolved leading to a consolidated model. In addition, this annotation mechanism offers the possibility to visualize the conflicts in the concrete, e.g., graphical syntax of the models.

With the help of this evaluation, we want to gain experiences of using both systems based on the following assumptions:

1. The Conflict-tolerant Merge avoids that changes of participants get lost when merging different versions of a model.

[1] http://www.eclipse.org/emf/compare/
[2] http://eclipse.org/modeling/

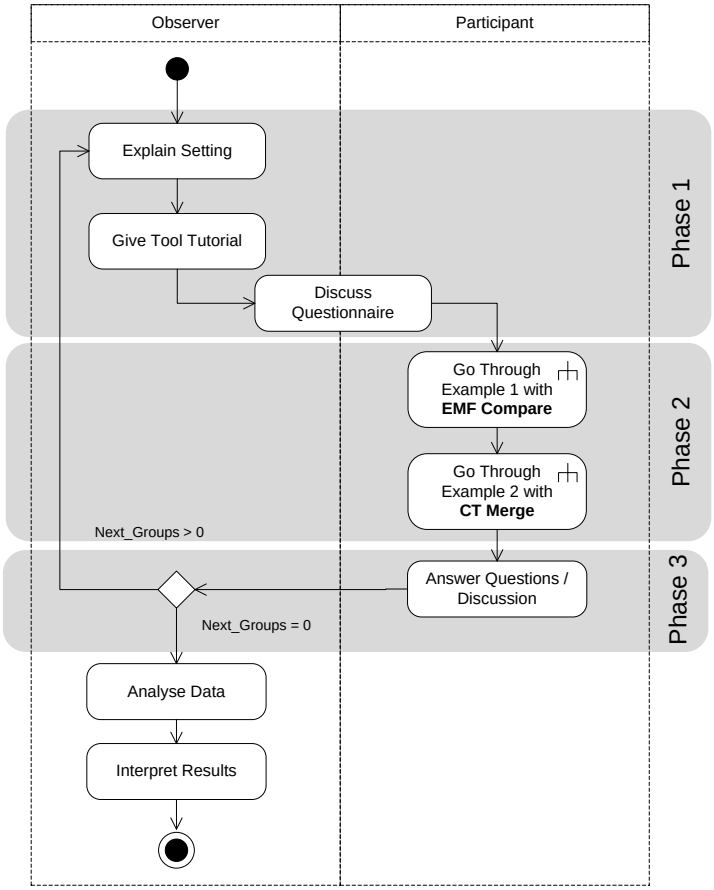

Figure 7.1: Study Procedure

2. The Conflict-tolerant Merge makes the reason why conflicts occur more obvious.

3. Collaborative conflict resolution based on Conflict-tolerant Merge leads to more accepted models.

The evaluation of the approach was conducted with groups of participants, which had to change a pre-defined model in parallel and independently of each other with the help of concrete change requests. One individual group, i.e., one run, passed through different phases, which are described in the following subsection in detail. The whole procedure is repeated for each group.

7.2 Study Procedure

In this study, we distinguished between two different roles: the observer and the participant. The author of this thesis assumed the role of the observer, who was responsible for preparing this evaluation and for the trouble-free execution of the study.

In total, 18 participants took part in the study. All of them had knowledge in modeling, did not already work with one of the tools, and did not participate in the AMOR project to ensure that they answer in an objective way. We decided to invite participants with an industrial as well as research background. Three different "kinds" of groups were assembled by us: (i) purely scientific group, (ii) purely industrial group, and (iii) mixed group. In total, 10 participants were working in industry and 8 in research institutions (Academia). The allocation of the participants to the individual groups is depicted in Table 7.1.

Group	Academia	Industry
1	0	3
2	3	0
3	3	0
4	1	2
5	0	3
6	1	2

Table 7.1: Allocation of Participants

Each group contained three participants (Participant A, Participant B, and Participant C) who had to perform predefined changes in parallel and independently. They were randomly allocated to these roles. They worked on a prepared model and had to merge these three versions including the resolution of occurred conflicts.

The whole procedure of the evaluation is depicted in Figure 7.1. One run with one group contained three phases and the duration of it was not restricted.

Phase 1: Preparation. The first phase started with an introduction, in which the general setting and the context of the evaluation were explained. The two different kinds of versioning paradigms and the two example models were presented. In the next step, the tools were presented to avoid that learning how to use a tool influences the evaluation results. After the short tool tutorials, the last step of phase 1 was to discuss each statement of the questionnaire, which was to be filled-in in phase 3 by the participants, to ensure that each question was not misinterpreted.

Phase 2: Modeling. For this phase, we prepared two different—but very similar—examples (Example A is depicted in Figure 7.3 and Example B in Figure 7.4). First, the participants have to go through one of the examples with EMF Compare and, secondly, through the other example with the Conflict-tolerant Merge. These two subprocesses of this phase are presented in more detail in the following.

The three participants got predefined change requests, which stated how they had to change the prepared models leading to three different versions of the model. The change requests for one participant contained almost four small modifications. The overall goal for the participants was the development of a merged version of each example, which was accepted by all participants.

The *first subprocess* describing the traditional versioning paradigm, which is simulated with the help of EMF Compare, is depicted in Figure 7.2 (a). Participants A, B, and C changed the model in parallel and independently of each other. First, Participant A had to check in her version

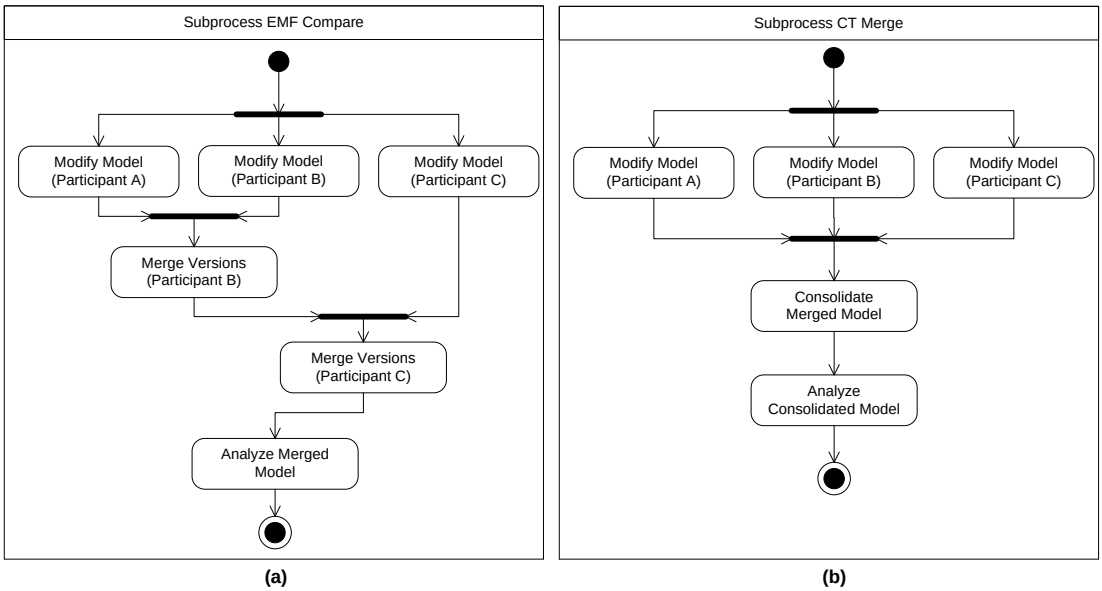

Figure 7.2: (a) Subprocess using EMF Compare and (b) Subprocess using CT Merge

resulting in a new head version in the central repository. When Participant B also tried the same, EMF Compare reported conflicts, which had to be resolved by her alone, when merging her version with the head version. When she finished merging the different versions of the model, a new head version was in the repository. Now, Participant C checked-in her changes and she had also to resolve conflicts when merging her version with the head version. At the end, all participants were invited to analyze the final result and to discuss the merged model.

After this process, the participants got another model and had to change it according to new change requests, which, again, are very similar to the change requests of the first example. Now, they used the Conflict-tolerant Merge tool to get a merged model. This *second subprocess* is depicted in Figure 7.2 (b). Now, the participants did not have to resolve the conflicts immediately after they checked-in. The Conflict-tolerant Merge incorporated all changes of all three participants and annotated the occurred conflicts. After all participants finished changing the model, they discussed the merged version and resolved the conflicts together. After that, they analyzed the consolidated version of the model. For this study, we decided that the participants had to resolve the conflicts in a face-to-face session. However, the Conflict-tolerant Merge builds a good basis also for other synchronous and asynchronous conflict resolution approaches as discussed in Section 5.3 of this thesis.

Phase 3: Post-processing. After the participants went through both examples, the post-processing phase was conducted. The participants had to answer the before discussed statements of the questionnaire and gave feedback during a final discussion. The questions and the discussion aimed to gather pros and cons of both systems and to get answers to the assumptions presented in Section 7.1.

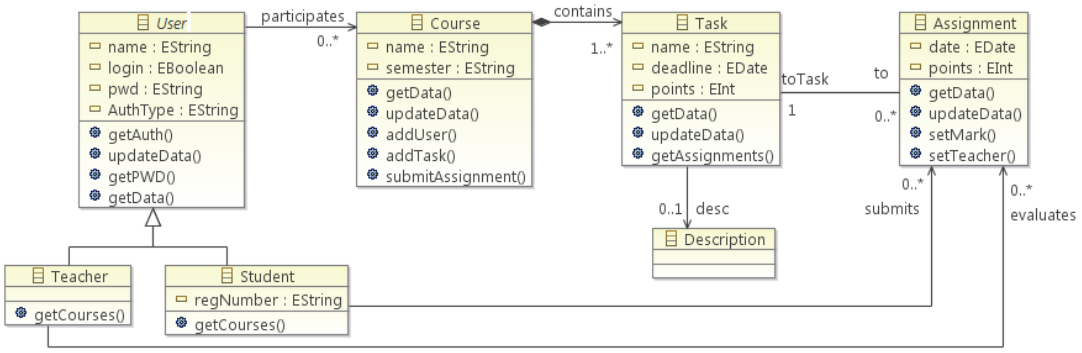

Figure 7.3: Example A: E-Learning System

7.3 Selection of Examples

In the following, we shortly present the examples used in the study. We decided to take two different examples for each tool to ensure objectiveness. Furthermore, two different domains were chosen, i.e., cinema platform and e-learning system, with which all participants were familiar with. We also tried to keep the models quite simple and defined change requests, which were—in our opinion—not too complex.

7.3.1 Example A

Example A represents the original model of an e-learning system as depicted in Figure 7.3. This system consists of a class `Teacher` and `Student`; both inherit from a common superclass `User`. A `User` may participate in a `Course`, which contains different tasks. Students may submit `Assignments` to a task, which are evaluated by a teacher.

The change requests for the participants and the resulting conflicts are presented in Table 7.2. "D/U" indicates a Delete/Update conflict, "U/U" indicates an Update/Update conflict, and a Move/Move conflict is represented by "M/M". In addition, a $\times$ indicates that a conflict between the two change requests is already marked with the dedicated abbreviation in the table.

Occurred Conflicts:

- Participant A changed the datatype of `AuthType` from `String` to `Boolean`, whereas Participant B created an enumeration containing different types of authorization. These concurrent changes led to an *Update/Update conflict*.
- Participant A performed the refactoring "Extract Class" on the attribute `deadline` of the class `Task`. She deleted this attribute, created a new class `Duration`, included the attributes `start` and `end`, and set a reference between the classes `Task` and `Duration`. In contrast, Participant B renamed the attribute `deadline` to `end` and included a new attribute `start`. Since the attribute `deadline` was deleted and updated in parallel, a *Delete/Update conflict* occurred.

- The final task of Participant A was to include the attributes `abstract` and `details` to the class `Description`. In contrast, Participant C deletes the class `Description` and, instead, she added an attribute `description` to the class `Task`. These concurrent changes led to *Delete/Update conflict*.
- Participant B moved the attribute `login` to the class `Course` whereas Participant C moved the same attribute to the newly added class `Configuration`. Here, a *Move/-Move* was reported by the Conflict-tolerant Merge. EMF Compare could not report such a conflict. Instead, it duplicated the moved model element.
- Finally, Participant B and C concurrently renamed the attribute `points` in the classes `Task` and `Assignment` leading to *Update/Update conflicts*.

CR#	Change Request	A1	A2	A3	B1	B2	B3	B4	C1	C2	C3
Participant A											
A1	The system should only distinguish whether a user of the system has the necessary authorization or not.				*U/U*						
A2	Instead of a simple attribute deadline, the duration of a task with a start and an end date should be in an own class.						*D/U*				
A3	A description of a task should contain an abstract and additional details.								*D/U*		
Participant B											
B1	Different authorization types should be included like “admin”, “standard”, etc.	×									
B2	The login status of a user should be handled in the dedicated course.									*M/M*	
B3	A task should have a start and end date.		×								
B4	The difference between points of tasks and points of assignments should be more clear.										*U/U*
Participant C											
C1	The description of a task should not be an own class.			×							
C2	The login status of a user should be handled in an own configuration class.					×					
C3	The difference between points of tasks and points of assignments should be clearer.						×				

U/U ... Update/Update Conflict
D/U ... Delete/Update Conflict
M/M ... Move/Move Conflict
× *... Conflict already marked in the table*

Table 7.2: Change Requests for Example A

7.3.2 Example B

Example B represents a cinema platform as depicted in Figure 7.4. On such a platform, two different kinds of users exist, namely `Visitor` and `Admin`. Both inherit from the common superclass `User`. The administrator may create on the platform a new `Cinema` with its `Rooms` or a new `Film`, which can be rated by the visitors.

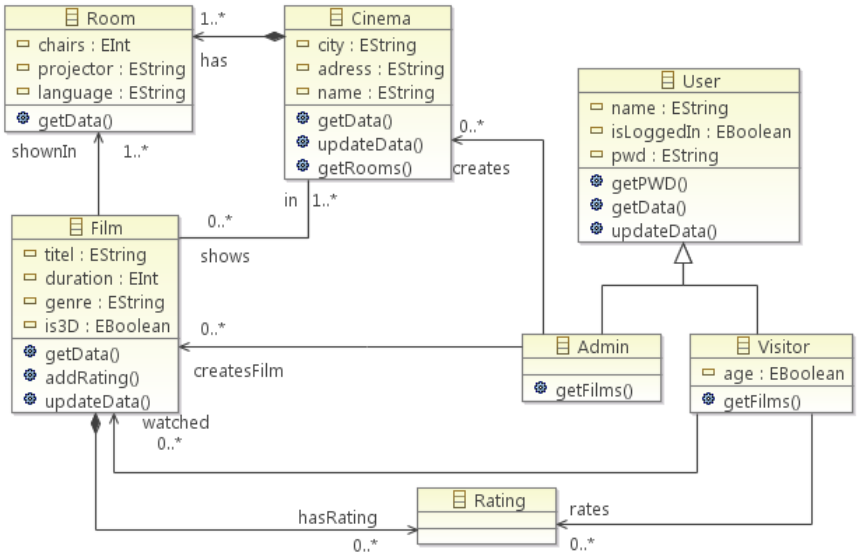

Figure 7.4: Example B: Cinema Platform

The change requests for the participants and resulting conflicts are presented in Table 7.3.

Occurred Conflicts:

- In this example, Participant A deleted the class `Rating` and added an attribute `rating` to class `Film`. In contrast, Participant B added to the deleted class an attribute `stars` and, thus, a *Delete/Update conflict* is reported.
- Participant A moved the attribute `language` of the class `Room` into the class `Film`, whereas Participant C moved the same attribute into another class, namely `Cinema`, leading to a *Move/Move conflict*.
- Furthermore, Participant A and B renamed the class `Room` to `Hall` or to `CinemaHall`, respectively. This concurrent change led to a *Update/Update conflict*.
- Participant B deleted the attribute `chairs` and added instead a new class `Chairs` with the attributes `row` and `number`. In contrast, Participant C renamed the attribute `chairs` to `seats` leading to an *Delete/Update conflict*.
- Finally, Participant B changed the datatype of the attribute `age` of the class `Visitor` from `String` to `Integer`. In parallel, Participant C set the datatype of the same attribute to a newly added enumeration containing literals such as `under12`, `under14`, etc. These changes led to an *Update/Update conflict*.

CR#	Change Request	A1	A2	A3	B1	B2	B3	B4	C1	C2	C3
Participant A											
A1	The film rating should be an attribute of a film instead of an own class.				*D/U*						
A2	The attribute language should describe a film (not a room).								*M/M*		
A3	Rename the class room to hall.						*U/U*				
Participant B											
B1	A rating should additionally have an attribute, which represents the actual rating in terms of stars.	×									
B2	The seats of a room should be reflected in an own class to include the attributes row and number.									*D/U*	
B3	Rename the class room to "CinemaHall".			×							
B4	The age of a visitor should be represented as Integer value.										*U/U*
Participant C											
C1	The attribute language should describe a cinema (not a room).		×								
C2	Rename the chairs of a room to seats.					×					
C3	Three different types of the age of a visitor exist, namely "under12", "under14" and "under16".							×			

U ... Update/Update Conflict
D ... Delete/Update Conflict
M ... Move/Move Conflict
× *... Conflict already marked in the table*

Table 7.3: Change Requests for Example B

7.4 Elaboration of Questionnaire

After the examples were conducted with EMF Compare and with the Conflict-tolerant Merge, the participants had to rate different statements. Five answers each question were possible, namely "strongly agree", "agree", "disagree", and "strongly disagree". In addition, the answer "don't know" was also possible. The selection of the statements was based on two criteria. They should give us the possibility to (i) test the above mentioned hypotheses and (ii) to check if objective and generally applicable criteria for the comparison of different data integration methods are fulfilled. These criteria are described in [BLN86] and are also suitable when merging different versions of a model in the context of model versioning, because the general problems are quite similar. Although the criteria can be taken as granted, all integration methods have to pit against them. In the following, we shortly describe them in the context of this evaluation:

- *Completeness:* If two versions of a model are merged, the resulting model should be complete, i.e., all changes of the participants should be incorporated in this model.
- *Correctness:* It is not enough that all changes of the participants are incorporated in the merged model. They also have to be incorporated correctly.
- *Minimality:* Although it has to be ensured that all changes of all participants are incorporated in the merged model, the model should also be minimal. This means that redundant elements should not be available in the merged model.

- *Understandability:* In addition, the changes of the participants, the occurred conflicts, and the merged model should be understandable. That means it should be comprehensible or obvious, which model element has been concurrently changed when regarding the resulting merged model.

In Table 7.4, the statements, which had to be rated by the participants are listed according to the assumptions:

S#	Statements of Questionnaire	C
Hypothesis 1: The Conflict-tolerant Merge avoids that changes of participants get lost when merging different versions of a model.		
1.	In contrast to EMF Compare, the CT Merge incorporates all of my changes.	Compl.
2.	In contrast to EMF Compare, the CT Merge incorporates the changes of the other developers.	
Hypothesis 2: The Conflict-tolerant Merge makes the reason why conflicts occur more obvious.		
3.	The merged model of the CT Merge is more comprehensible than the one of EMF Compare.	Understandability
4.	The changes and resulting conflicts are more comprehensible when using the CT Merge than merging with EMF Compare.	
5.	The changes and resulting conflicts are more comprehensible if they are visualized in a merged model in terms of annotations than being visualized between different versions of the original model in terms of correspondences.	
Hypothesis 3: Collaborative conflict resolution based on Conflict-tolerant Merge leads to more accepted models.		
6.	All of my changes are more adequately incorporated in the final model of the CT Merge than in the one of EMF Compare.	Correctness / Minimality
7.	I do more agree with the changes of the other developers incorporated in the final model of the CT Merge than in the one of EMF Compare.	
8.	Changes incorporated in the final model of the CT Merge are less redundant than in the one of EMF Compare.	
9.	All changes incorporated in the final model of the CT Merge are more correct than in the one of EMF Compare.	

S# ... Statement Number
C ... Data Integration Criteria
Compl. ... Completeness

Table 7.4: Statements to be Rated by the Participants

7.5 Results

In the following, we present the results of the questionnaire and the most important observation as well as a critical discussion about this study.

7.5.1 Results of Questionnaire and Observations

After the whole study, we analyzed the filled-in questionnaire. The results and are presented in Table 7.5.

We transformed the possible answers into numerical values to empirically analyze them. "Strongly agree" is represented by "1", "agree" by "2", "disagree" by "3", and "strongly disagree" by "4". In the analysis, the "don't know" answers were subtracted out for the following reason. In all cases in which a participant gave the answer "don't know", there was a reason why she could not rate the statement. Thus, it never happened that a participant simply did not have an opinion regarding a statement.

Statement	Mean Value	Median	Min Value	Max Value	Total Number (*)	"Don't know"	Generally Agree (1+2)	Generally Disagree (3+4)	"Strongly agree" (1)	"Agree" (2)
Assumption 1: The Conflict-tolerant Merge avoids that changes of participants get lost when merging different versions of a model.										
1.	1.31	1.00	1	2	16	2	100.0%	0.0%	68.8%	31.3%
2.	1.44	1.00	1	2	18	0	100.0%	0.0%	55.6%	44.4%
Assumption 2: The Conflict-tolerant Merge makes the reason why conflicts occur more obvious.										
3.	1.17	1.00	1	2	18	0	100.0%	0.0%	83.3%	16.7%
4.	1.17	1.00	1	2	18	0	100.0%	0.0%	83.3%	16.7%
5.	1.38	1.00	1	2	16	2	100.0%	0.0%	62.5%	37.5%
Assumption 3: Collaborative conflict resolution based on Conflict-tolerant Merge leads to more accepted models.										
6.	1.79	2.00	1	3	14	4	92.9%	7.1%	30.8%	69.2%
7.	1.65	2.00	1	2	17	1	100.0%	0.0%	35.3%	64.7%
8.	1.50	1.50	1	2	6	12	100.0%	0.0%	50.0%	50.0%
9.	1.70	2.00	1	2	10	8	100.0%	0.0%	30.0%	70.0%

(*) ... without "Don't Know"
"strongly agree" ... 1
"agree" ... 2
"disagree" ... 3
"strongly disagree" ... 4

Table 7.5: Results of Questionnaire

In the first column, references to the statements (S1-S9) of the questionnaire are listed. For each statement, the mean value and median are presented, as well as the minimum and maximum value. Furthermore, the total number of answers, which are used for calculating the mean value and median, are presented. This number represents the answers without "don't knows", which

in turn are summed up and presented in the next column. Finally, we compared the amount of participants who generally agreed with a statement with those who generally disagreed. Finally, we present in the last two columns the difference between participants who strongly agree with those who simply agree with a statement. In addition to the results of the questionnaire, we report about our observations and the feedback from our participants.

All of the participants (68.8% strongly agreed and 31.3% agreed) generally agreed with statement 1 that, in contrast to EMF Compare, the Conflict-tolerant Merge incorporates all of their self-performed changes. Furthermore, all participants (55.6% strongly agreed and 44.4% agreed) stated that, in contrast to EMF Compare, the Conflict-tolerant Merge incorporates also the changes of the others (cf. statement 2). After the first example was conducted with EMF Compare, most of the participants claimed that their changes are not available in the merged model. In particular, we observed that all participants in the role of Participant A, who have been the first committing their changes, missed them completely. But also the participants in the role of Participant B missed nearly all of her changes. It was often the case that the participants in the roles of Participant B and C did not understand the changes and, especially, the conflicts when using EMF Compare. Thus, they ignored the changes of the others leading to a merged model including only their own changes.

The understandability of the merged model and the occurred conflicts are demonstrated with statements 3 and 4, respectively. All participants generally agreed (83.3% strongly agreed and 16.7% agreed) with statement 3 that the merged model of the Conflict-tolerant Merge is more comprehensible than the one of EMF Compare (cf. statement 3). Also the occurred conflicts were more comprehensible when using the Conflict-tolerant Merge (83.3% strongly agreed and 16.7% agreed) as demonstrated with statement 4. We observed that all of the participants struggled to comprehend why a conflict occurred when merging with EMF Compare. In addition, for all participants it was difficult to comprehend the intentions behind the parallel changes of the other participants which led to a conflict.

The reason for these results might be explained by statement 5: All participants (62.5% strongly agreed and 37.5% agreed) stated that visualizing conflicts in terms of annotations in a merged model instead of visualizing them with the help of correspondences between two versions of a model leads to more understandability. For all participants it was easier to comprehend the conflicts in a merged version of the model although all changes of three participants are merged instead of having two different versions side by side and then producing a merged version. Further, newly added and also deleted elements are visualized in EMF Compare with correspondences pointing somewhere between two elements in the tree view. This situation led to confusions by nearly all participants, because there did not exist a correspondent model element. Overall, the high understandability of the conflicts and the resulting merged model when using the Conflict-tolerant Merge can be explained by the possibility of visualizing the conflicts in the concrete syntax of the models, in which the participants also performed their changes. In contrast, merging with EMF Compare was impeded by the mismatch of modeling in the concrete syntax and resolving the conflicts in the abstract syntax of the model.

92.9% of all participants (30.8% strongly agreed and 69.2% agreed) generally agreed that their changes are more adequately incorporated in the final model of the Conflict-tolerant Merge than in the one of EMF Compare that was demonstrated with statement 6. However, some of the

participants (4 out of 18) stated “don’t know” at statement 6, because when using EMF Compare it often happened that no single change was incorporated in the merged model, because the participants declined the changes of the others when merging due to problems in understanding the changes and resulting conflicts. Thus, it was difficult to assess if a change is “more adequately” incorporated with the Conflict-tolerant Merge.

A similar problem became evident regarding statement 9: Although all participants agreed that all of their changes in the merged version are more correct when using the Conflict-tolerant Merge, 8 participants could not rate this statement, because a comparison with EMF Compare was not possible, when no change was incorporated.

Statement 7 demonstrated, if the participants did more agree with the changes of the others incorporated in the final model. All participants agreed (35.3% strongly agreed and 64.7% agreed) that the acceptance of the concurrent changes of the others, which led to conflicts, increased with the conflict-tolerant approach, because they better understood the reasons behind the changes.

Finally, to prove that the final models were minimal in the sense that no redundancies existed in the final model, statement 8 was established. However, it did not deliver a result because most of the participants (12 out of 18) could not rate this statement. On the one hand, no redundancies occurred after merging with EMF Compare due to discarding some of the changes by the participants, and, on the other hand, some participants did not recognize the redundancies. Redundancies would occur when merging with EMF Compare, because it duplicates a model element when two developers move it concurrently. However, 6 out of our 18 participants agreed with statement 8 that the changes incorporated in the final model of the Conflict-tolerant Merge are less redundant than in the one of EMF Compare.

7.5.2 Implications for Assumptions

Implication for Assumption 1: This assumption was that the Conflict-tolerant Merge avoids that changes of participants get lost when merging different versions of a model. Statement 1 and 2 of the questionnaire have been established to check if the merged models are complete. This means that no changes of the participants get lost when merging parallel versions of one model. In the final discussions and according to the results of the questionnaire, all participants agreed that the Conflict-tolerant Merge incorporates their own changes and the changes of the others.

Implication for Assumption 2: Statement 3, 4, and 5 were established to find out the understandability of the produced artifacts. According to the observations, interviews, and questionnaire the changes of the participants, the resulting conflicts, and the merged model are more comprehensible when using the Conflict-tolerant Merge. Also the visualization of conflicts in terms of annotations in the merged model are more preferred by the participants of this evaluation. We may state that the Conflict-tolerant Merge makes the reason why conflicts occur more obvious.

Implication for Assumption 3: Statement 6 and 7 led to clear results. However, statement 8 could not be answered by several participants as discussed in the subsection before. Thus, we cannot report that the merged model is minimal when using the Conflict-tolerant Merge tool. We also observed that many participants did discovered redundancies in the final model. To test such a criterion, other tests would be more adequate than user experiments like conducted for the evaluation of this thesis. Concerning statement 9, 8 out of 18 participants could not say whether they agree or disagree with this statement. However, we have also collected qualitative data in terms of observations during the study and interviews after the study. Overall, all participants confirmed that collaborative conflict resolution based on the Conflict-tolerant Merge leads to a final model which is more accepted than merging with EMF Compare following the traditional versioning paradigm.

7.5.3 Critical Discussion

In this quasi-experimental study, we gained experiences about the usage of the Conflict-tolerant Merge when merging different versions of model in a small team. We conducted a comparison with EMF Compare, but due to the lack of empirical evidence, we do not want to rate both tools. This evidence will be checked when conducting real-world case studies or controlled experiments in future. In the following, we outline findings of the presented initial evaluation, which will help us to conduct such case studies or experiments.

Within each individual run of the study presented before, three people participated who took different roles with different tasks. For example, it happened that all changes of Participant C were also correctly incorporated with EMF Compare, because she was responsible for the last merge. In contrast, the changes of Participant A and B were at least partly discarded during the merge. This influenced the rating of the statements.

Furthermore, the study was conducted with predefined processes following the conflict-tolerant merge approach and the traditional versioning paradigm. We consider the Conflict-tolerant Merge as basis for collaborative conflict resolution. However, it would also lead to interesting findings when the participants had to resolve the conflicts alone using the Conflict-tolerant Merge. On the other side, EMF Compare is however not designed for resolving the conflicts collaboratively but it would be another interesting variability of the process to let the participants resolve the conflicts together.

We used two different examples for the evaluation to facilitate that one group may evaluate both tools. We designed both examples and the change requests as similar as possible although we used two very different domains to ensure an objective rating of both tools. In addition, for each group we switched example A and B to avoid that the used example influenced the rating of a tool. That means three groups used example A in combination with EMF Compare and, then, example B with the Conflict-tolerant Merge and the other three groups started with example B using EMF Compare.

Furthermore, we prepared relatively small models and not too complex change request for the participants. Another interesting finding would be if the future experiments would lead to the same results when larger models and more complex change requests are used.

As already mentioned, we did not restrict the duration of one individual run, because we wanted to focus on the usability of the systems and the applicability of the conflict-tolerant

approach. In future experiments, it would lead to interesting findings, if the time for merging with EMF Compare is restricted and the quality of the model is then evaluated.

CHAPTER 8

Conclusion

In the following, we briefly recall the major contributions presented in this thesis, discuss the lessons learned from conducting the research, and conclude with an outlook on future work.

8.1 Contributions

Three major contributions have been elaborated in detail:

Survey on Versioning in Practice. To overcome the lack of empirical studies in the area of software and model versioning, we have conducted an online survey with 90 participants and in-depth interviews with ten experts. Such studies are needed to gather requirements from practice with empirical evidence. We were able to gain participants from industry as well as from different research communities. Thus, the heterogeneity of participants enabled us to get insights into a broad spectrum of different application scenarios in the domain of versioning. Moreover, with the help of the empirical study, we learned how the challenge of coping with collaborative development is currently addressed in practice. The results of the questionnaire showed us how the state-of-the-art versioning habits and processes look like when setting different artifacts under version control. The expert interviews gave us deep insights in experiences stemming from applying versioning tools in different modeling domains. We learned that the current best practice is to find a reasonable *division of work* among project members. However, dividing the work still does not avoid entirely the possibility of conflicts. If conflicts occur, dedicated tool support for the *management of conflicts* is urgently needed, especially they occur in concurrently changed models.

Conflict-tolerant Model Versioning System. In this thesis, we have presented a new paradigm for optimistic model versioning. Instead of forcing the developers to resolve the conflicts immediately and alone, we have extended our model versioning system AMOR with a conflict-tolerant

model merge strategy. Thereby, several parallel versions of a model are merged into one common version using dedicated merge rules. The detected conflicts between all versions are marked in order to delay the conflict resolution to a later point in time when the involved developers are ready to address and resolve occurred conflicts. The merged model constitutes a good basis for discussing and resolving conflicts in a team. Moreover, in this thesis we also presented a conflict resolution model, which allows to assign developers to a specific conflict and to prioritize the concurrent changes in an asynchronous manner. Since we have also introduced a lifecycle for conflicts and maintain the information on how a conflict is has been resolved, developers are fully aware of what happened in the development process of a model. The evaluation of the conflict-tolerant approach clearly showed that the evolution of a model is more comprehensible and a consolidated version of the concurrently changed model can be found more easily among all developers. This approach is supported by a light-weight model annotation mechanism, which is summarized in the following description of the third contribution of this thesis.

Collaboration Support through Model Annotations. When merging parallel versions of a model in a conflict-tolerant way, conflicts are annotated. In addition, these conflict annotations are enriched with meta information to better support the understanding of the parallel changes of all participants. We also showed in this thesis how this model annotation mechanism supports the developers when resolving the conflicts in an asynchronous way. This light-weight annotation mechanism, called EMF Profiles, follows the principle of AMOR which avoids restrictions regarding the modeling language and modeling editor. Thereby, every EMF-based model may be annotated with stereotypes containing tagged values. The annotations are saved in a separate model to avoid polluting the annotated model. In this thesis, we also presented two novel techniques to enable the systematic reuse of profile definitions across different modeling languages. In addition, the conflict profile may also be flexibly extended to add new kind of conflicts and warnings.

8.2 Discussion

From conducting the research work presented in this thesis, we have learned that the rise of model-driven software development poses a new challenge, namely the concurrent and collaborative development of models. Therefore, dedicated model versioning systems are needed to address this challenge. However, state-of-the-art model versioning systems largely focus on the technical aspect of model versioning and often neglect the fact that the major purpose of versioning is to *support people* in developing software artifacts collaboratively. Thus, the contributions of this thesis aimed at addressing this deficiency of current approaches. More precisely, the goal of this thesis was to improve the support of groups of developers when concurrently developing models and, therefore, to bridge the gap between the research areas of model versioning and collaborative modeling—on the basis of the maxim that software is *by* and *for* people. Following this maxim, the first step was to obtain important input not only from a broad variety of literature but also from deriving requirements from people developing software in practice. Therefore, we conducted an online survey and expert interviews. Finally, we performed an evaluation of the

presented contributions by gathering experiences and valuable feedback from people, who took time to test our tool.

Survey. The survey provided us with first insights in the versioning process applied in practice and the versioning habits of our survey participants. However, also several additional questions arose from analyzing the survey data, because different stakeholders of a versioning system have diverging requirements and expectations on such a system. To investigate the applied versioning process, the best practices in versioning, and the requirements for versioning systems with respect to different stakeholders of a versioning system more deeply, we decided to conduct semi-structured interviews with different people coming from different domains. The broad variety of people working at different institutions and having diverging roles led to very different interviews regarding the duration but also regarding the gained insights. It was very challenging to analyze the results of these interviews in order to get a unified picture of the interviewee's expectations and opinions. Due to the broad variety of people and their different viewpoints, we were not able find a consolidated picture of *all* comments of the interviewees and we could not address all of the ideas, expectations, and requirements learned from the interviews. Nevertheless, we have elaborated several very interesting requirements, which most of the interviewees agreed upon. These common requirements constituted the input for our further research.

Conflict Tolerance. For instance, one of the requirements that all interviewees directly or indirectly mentioned, concerned the inevitability of conflicts, despite a reasonable division of work, and the major challenge that is posed when a conflict occurs. Many of the interviewees mentioned that a conflict often interferes the entire work, because a team member is forced to resolve the conflict immediately. For resolving this conflict, the team member often has to consult the other developers, who applied the conflicting changes, in order to understand their rationale; otherwise, important changes might get lost. Thus, resolving a conflict might prevent several developers from doing their actual work.

To address this issue derived from the interviews, we proposed the Conflict-tolerant Merge. We managed to develop a system with which merging of different parallel versions is possible without worrying immediately about the critical resolution of occurred conflicts. Thus, developers may proceed with their work and complete the work items they are currently working on according to the applied development process. Also, another major requirement that was raised commonly by several interviewees concerns the tangibility and understandability of conflicts. Several interviewees mentioned that it is hard to understand the underlying conflicts when only considering *changes* applied to models without having available the *model* resulting from these changes. Using the conflict-tolerant merge in combination with our annotation mechanism, we aimed at making conflicts more tangible and comprehensible. Whenever the developers are able to come together and collaborate on the resolution of occurred conflicts, they are able to create a basis for discussing and resolving the conflicts. Thereby, developers may reason about a conflicting *model* and not only about conflicting changes.

Conflict-tolerant merging also has its disadvantages in certain settings. When models and teams working on these models get larger and larger, conflict-tolerant merging might cause a "rank growth" of parallel versions and conflicts. The larger the model and the more people

working those models, the more important gets a discipline handling of the conflicts. Comparable to bugs, conflicts also need to be tracked and monitored precisely. Also, when living with conflicts in the most current version of a model, the resolution of conflicts has to be integrated into the applied development process. The right point in time to resolve all or a subset of conflicts strongly depends on the specific development process. Therefore, one important next step is to review existing development processes and how they are conducted in practice, to identify the best place in the process for resolving conflicts and tightly integrate this step into the process. Another challenge arising from large models and large teams concerns the scalability of the visualization of conflicts. If a model with thousands of model elements comprises hundreds of unresolved conflicts, developers may quickly become overcharged. Therefore, further research is required to investigate the impact of conflicts in large models and to develop adequate techniques for conflict filters and views.

Although this thesis is only concerned with *model* versioning, the concept behind the conflict tolerant merge seems to be beneficial for versioning in general (i.e., also for code and documents). Especially for documents, it is likely that the advantages of our merge strategy help authors to collaboratively work on their documents. In this domain, several research has been conducted for synchronous editing (e.g., [MM93]), but our merge strategy might improve the asynchronous collaboration in writing. When adopting conflict-tolerant merging for code, however, one major challenge is that the merged code might not be executable any more. Code that cannot be compiled and executed anymore might be very hard to analyze and has to be fixed before developers may investigate the affect of certain changes on the functioning of a program. This aspect might mitigate the advantages of conflict-tolerant merging for code.

Conflict-tolerant merging is to a certain extent also reflected in current text-based versioning systems. For instance, if a conflict occurs using SVN, a merged version is generated, which duplicates the text lines that have been concurrently modified. Moreover, these duplicated text lines are surrounded by dedicated conflict markers to indicate the conflict more prominently. Developers may now use this version, mark it as resolved and save them into the repository again. In this way, the resolution of the conflict is postponed to another point in time. However, in contrast to the approach proposed in this thesis, the conflicting changes are not really merged; the concurrently changed lines are simply duplicated and important meta-data about the conflict is missing. Moreover, the conflict annotation is intermingled with the code and destroys readability and understandability of both the code and the conflict.

EMF Profiles. We developed EMF Profiles with the purpose to annotate occurred conflicts between EMF-based models and visualize them appropriately. In addition, we also annotate critical situations, which contain per se no conflicts, with warnings to increase the awareness of contradicting modification in a team. It provides a new way for presenting changes and conflicts in the context of model versioning. Since every EMF-based models can be annotating with EMF Profiles, the capability of this project is way beyond the use case in model versioning. For example, we will use EMF Profiles for enhancing code generation capabilities using annotated models and for annotating models during model reviews. When using EMF, a situation in which annotating and extending an EMF model is necessary, might easily occur, but changing its metamodel is a tedious task to make this possible. There are many reasons why changing

the metamodel is not desired. For example, the additional information is only needed for one particular project and you do not want to pollute others; or you want to avoid the tedious task to recreate the graphical modeling environment and migrate already existing models to the new metamodel version. This light-weight model annotation mechanism is realized by adopting the idea of UML profiles, which have been a key enabler for the success of UML.

8.3 Future Work

In the following, we discuss current limitations of the approach presented in this thesis and interesting directions for future work.

Usability. First of all, increasing the usability of our research prototype presented in this thesis is one important task for future work. Not only the look-and-feel could be improved but also new features may be added. As presented in Chapter 6, the conflict annotation and the additional meta-information are saved as separate model. This gives us many different possibilities to improve the usability and to add new features for supporting the users during the merge. Conflicts may be integrated in a bug tracking tool or may be send to other users for resolving them, etc. Currently, much information is put in the properties view, which could be improved from an usability point of view. Furthermore, to tackle the problem of scalability of the visualized conflicts or warnings, different views and filters are necessary to avoid that the developers are becoming overcharged.

Diagram Versioning. The presented approach shows detected conflicts in the concrete or graphical syntax of a model. However, also this layout information of the model has to coevolve with the model. Thus, the diagram should be put under version control. Due to the fact that diagram information is nowadays also represented in terms of models, model versioning features might be reused, whereby the nature of 2D diagram layout must not be neglected. The most challenging question is how to preserve the mental map (cf. [MELS95]) of all developers when merging several parallel versions. In addition, no general notion has been established yet which concurrently performed layout changes are in fact contradicting changes (e.g., also considering the inconvenience of small unintended changes when a modeler moves an element one pixel without intending it).

EMF Profiles. One of the major components of the approach presented in this thesis is the EMF Profiles project. One conflict profile is used to annotate occurred conflicts between two or more concurrent changes. In the future, we plan to elaborate on more sophisticated restriction mechanisms to allow constraining the application of stereotypes (e.g. with OCL conditions) and composing several independent profiles which are not mutually complementary in one profile application as proposed by [NGTS10]. A consistent mix of several profiles requires a mechanism to specify conditions constraining applicability across more than one profile. For instance, one may need to specify that a stereotype of profile *A* may only be applied after a stereotype of profile *B*, holding a specific tagged value, has been applied.

Composite Changes. In this thesis, we have considered atomic and overlapping changes when merging different versions of a model. However, also other kinds of conflicts may arise when changing the model in parallel as presented in Section 3.2. For example, Philip Langer presented in his thesis [Lan11] the detection of conflicts due to composite changes such as model refactorings. When considering also composite changes the question arises how to support the user in resolving such conflicts by the annotation mechanism presented in this thesis.

Real-world Case Studies. As already discussed in Chapter 7, the experimental study to evaluate the presented approach was conducted in an artificial context. In addition, interesting findings might arise, when using the approach in a broader industrial context. Real-world case studies would better demonstrate the usefulness and scalability of the approach. However, this would go beyond the scope of this thesis and, thus, it is left for future work.

APPENDIX A

Questionnaire

On the following pages, we “print” the questionnaire as it was depicted on our project homepage `http://www.modelversioning.org`.

Questionnaire - Versioning in Practice

In the context of the project AMOR we are developing a system for model versioning. One of the first steps is collecting the requirements and demands of potential users for such a system. Therefore it is important to understand how versioning is performed in practice. So we kindly ask you to fill in the following questionnaire.

Thank you very much for your support!

Background

Which role do you mainly play in projects?

- manager
- developer
- architect
- tester
- Other:

Which other roles do you occasionally have in projects?

- manager
- developer
- architect
- tester
- Other:

How is your team geographically distributed?

- in the same building
- in the same town
- in the same time zone
- all over the world

How many people typically participate in your projects?

- up to 5 persons
- up to 20 persons
- up to 100 persons
- more

Page 2 After page 1 Continue to next page

Versioning Habits

What versioning strategy do you apply?

- pessimistic (lock/modify/unlock)

- none at all

Why do you use versioning systems?

- concurrent development
- branching
- traceability of changes
- Other:

Which artifacts do you version?

- code
- models
- documentation
- requirements
- change request/bugs
- Other:

Do you use different versioning strategies for different artifacts?

- yes
- no

Which level of granularity do you apply when versioning your code?

- file (coarse grained)
- elements within a file (fine grained)
- none at all
- don't know

Which level of granularity do you apply when versioning your models?

- file (coarse grained)
- elements within a file (fine grained)
- none at all
- don't know

I would like to version in a more fine grained manner.

- yes
- no

Are there different levels of authorization (e.g., read/write permissions) for changing artifacts within a project?

- yes
- no
- in planning stage

If you answered the last question with "yes" or with "in planning stage" (i.e., you have/plan to have levels of authorization), could you please describe them?

Do you distinguish between different roles in the versioning process? (Who is permitted to change something, who may commit changes,...)

- yes
- no
- in planning stage

Page 3 After page 2 Continue to next page

Versioning Process

Do you have a standard process (either on enterprise level or on project level) how versioning has to be performed?

- yes
- no
- in planning stage

If you answered the last question with "yes" or with "in planning stage" (i.e., you have/plan to have a standard process), could you please describe it?

Which phase of the versioning process causes the most effort?

- update
- conflict detection
- conflict resolution
- locking of artifacts
- unlocking locked artifacts
- Other:

What are the reasons for this effort?

How often should a check-in/comit be performed?

- not regulated
- when the employee leaves the workplace
- when a certain functionality is finished
- after testing

Based on which criteria will a changed version be accepted?

- there are no criteria
- unit tests have to be passed
- confirmation by a person in charge
- Other:

Who is allowed to perform big changes like refactorings?

- everybody
- a person in charge

Which criteria when resolving conflicts are most important?

- Minimal effort
- Shortest time
- quality first
- Other:

Who is responsible to resolve a conflict?

- the developer who is committing
- the developers who are responsible for the conflict
- a person in charge
- the whole team
- Other:

What do you think about tolerating all conflicts, marking them, and resolving them later on in a consolidation phase?

Lessons Learned

How often do multiple developers change the same artifact at the same time?

- often
- sometimes
- never

Do you think that the locking of one resource by one developer causes delays in the project's progress?

- yes
- no

Which tool do you use for versioning?

In your opinion, what are the biggest challenges of versioning?

Which of the following do you consider the most important requrirement for a versioning system?

- conflict detection
- visual conflict resolution support
- customizeable automatic conflict resolution

Which features would you like to have in a versioning system?

Thank you!

Thank you very much for your time. If you are interested in the results of this survey, please leave your contact data. Your data will be handled confidentially and will be processed anonymously.

Name (optional)

Email (optional)

Affiliation

List of Figures

List of Tables

Bibliography

[ABK+09] K. Altmanninger, P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. Why Model Versioning Research is Needed!? An Experience Report. In *Proceedings of the Joint MoDSE-MCCM 2009 Workshop @ MoDELS'09*, 2009.

[ADR06] S. Auer, S. Dietzold, and T. Riechert. OntoWiki - A Tool for Social, Semantic Collaboration. In *Proceedings of the 5th International Semantic Web Conference (ISWC'06)*, volume 4273 of *LNCS*, pages 736–749. Springer, 2006.

[AK01] C. Atkinson and T. Kühne. The Essence of Multilevel Metamodeling. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML'01)*, volume 2185 of *LNCS*, pages 19–33. Springer, 2001.

[AK07] C. Atkinson and T. Kühne. A Tour of Language Customization Concepts. *Advances in Computers*, 70:105–161, 2007.

[AKK+08] K. Altmanninger, G. Kappel, A. Kusel, W. Retschitzegger, M. Seidl, W. Schwinger, and M. Wimmer. AMOR—Towards Adaptable Model Versioning. In *Proceedings of the 1st International Workshop on Model Co-Evolution and Consistency Management @ MoDELS'08*, 2008.

[AOH07] T. Apiwattanapong, A. Orso, and M. Harrold. JDiff: A Differencing Technique and Tool for Object-oriented Programs. *Automated Software Engineering*, 14(1):3–36, 2007.

[AP03] M. Alanen and I. Porres. Difference and union of models. In *Proceedings of the International Conference on the Unified Modeling Language (UML'03)*, pages 2–17. Springer Verlag, 2003.

[ASW09] K. Altmanninger, M. Seidl, and M. Wimmer. A Survey on Model Versioning Approaches. *International Journal of Web Information Systems*, 5(3):271–304, 2009.

[Bae95] R. Baecker. *Readings in human-computer interaction: toward the year 2000*. The Morgan Kaufmann Series in Interactive Technologies Series. Morgan Kaufmann Publishers, 1995.

[Bal89] R. Balzer. Tolerating inconsistency. In *Proceedings of the 5th International Software Process Workshop (ISPW '89)*, pages 41–42. IEEE Computer Society, 1989.

[BCJM10] H. Bruneliere, J. Cabot, F. Jouault, and F. Madiot. MoDisco: a generic and extensible framework for model driven reverse engineering. In *Automated Software Engineering (ASE'10)*, pages 173–174. ACM, 2010.

[BE09] L. Bendix and P. Emanuelsson. Requirements for Practical Model Merge - An Industrial Perspective. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, volume 5795 of *LNCS*, pages 167–180. Springer, 2009.

[Béz05] J. Bézivin. On the unification power of models. *Software and Systems Modeling*, 4(2):171–188, 2005.

[BKL+11a] P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. *Emerging Technologies for the Evolution and Maintenance of Software Models*, chapter The Past, Present, and Future of Model Versioning. IGI Global, 2011.

[BKL+11b] P. Brosch, H. Kargl, P. Langer, M. Seidl, K. Wieland, M. Wimmer, and G. Kappel. Conflicts as First-Class Entities: A UML Profile for Model Versioning. In *Models in Software Engineering - Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers*, pages 184–193, Springer, 2011. LNCS Volume 6627.

[BKS+10] P. Brosch, G. Kappel, M. Seidl, K. Wieland, M. Wimmer, H. Kargl, and P. Langer. Adaptable model versioning in action. In *Modellierung 2010*, volume 161 of *LNI*, pages 221–236. GI, 2010.

[BLN86] C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Survey*, 18:323–364, 1986.

[BLS+09] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. An Example is Worth a Thousand Words: Composite Operation Modeling By-Example. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, pages 271–285. Springer, 2009.

[BLS+10a] P. Brosch, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. Colex: a web-based collaborative conflict lexicon . In *Proceedings of the 1st International Conference on Model Comparison in Practice (IWMCP)*, pages 42–49. ACM, 2010.

[BLS+10b] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, and G. Kappel. Concurrent Modeling in Early Phases of the Software Development Life Cycle. In *Proceedings of the 16th International Conference on Collaboration and Technology (CRIWG'10)*, pages 129–144, 2010.

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

[Bro11] P. Brosch. *Conflict Resolution in Model Versioning*. PhD thesis, Vienna University of Technology, 2011.

[BSM+03] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework (The Eclipse Series)*. Addison-Wesley, 2003.

[BSW+09] P. Brosch, M. Seidl, K. Wieland, M. Wimmer, and P. Langer. We can work it out: Collaborative Conflict Resolution in Model Versioning. In *Proceedings of the 11th European Conference on Computer Supported Cooperative Work (ECSCW'09)*, pages 207–214. Springer, 2009.

[CDRP08] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing Model Conflicts in Distributed Development. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'08)*, LNCS, pages 311–325. Springer, 2008.

[CESW04] T. Clark, A. Evans, P. Sammut, and J. Willans. *Applied Metamodelling, A Foundation for Language Driven Development*. Ceteva, Sheffield, 2004.

[Che76] P. P.-S. Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

[CSG63] D. Campbell, J. Stanley, and N. Gage. *Experimental and quasi-experimental designs for research*. Houghton Mifflin Boston, 1963.

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

[DA00] A. Dey and G. Abowd. Towards a better understanding of context and context-awareness. In *Proceedings of the Workshop on the What, Who, Where, When and How of Context-Awareness*. ACM Press, 2000.

[DH07] P. Dewan and R. Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *Proceedings of the 2007 Tenth European Conference on Computer-Supported Cooperative Work*, pages 159–178. Springer, 2007.

[Dij97] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1st edition, 1997.

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

[DLFOT06] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. ADAMS: Advanced Artefact Management System. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 349–350. IEEE, 2006.

[DLFST09] A. De Lucia, F. Fasano, G. Scanniello, and G. Tortora. Concurrent Fine-Grained Versioning of UML Models. In *European Conference on Software Maintenance and Reengineering*, pages 89–98. IEEE, 2009.

[DMJN08] D. Dig, K. Manzoor, R. Johnson, and T. Nguyen. Effective Software Merging in the Presence of Object-Oriented Refactorings. *IEEE Transactions on Software Engineering*, 34(3):321–335, 2008.

[Edw97] W. K. Edwards. Flexible Conflict Detection and Management in Collaborative Applications. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology (UIST'97)*, pages 139–148. ACM, 1997.

[EE68] D. Engelbart and W. English. A research center for augmenting human intellect. In *Proceedings of the Fall Joint Computer Conference, part I*, pages 395–410. ACM, 1968.

[ELH+05] J. Estublier, D. Leblang, A. Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. 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.

[FBJ+05] M. D. D. Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. AMW: a generic model weaver. In *Journée sur l'Ingénierie Dirigée par les Modeles (IDM'05)*, 2005.

[FFVM04] L. Fuentes-Fernández and A. Vallecillo-Moreno. An Introduction to UML Profiles. *UPGRADE, The European Journal for the Informatics Professional*, 5(2):5–13, 2004.

[FGH+94] A. Finkelstein, D. M. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency Handling in Multperspective Specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

[Flu09] M. Fluegge. Entwicklung einer kollaborativen Erweiterung fuer GMF-Editoren auf Basis modellgetriebener und webbasierter Technologien. Master's thesis, University of Applied Sciences Berlin, 2009.

[FMP06] G. Fitzpatrick, P. Marshall, and A. Phillips. CVS integration with notification and chat: lightweight software team collaboration. In *Proceedings of the 2006 20th anniversary conference on Computer Supported Cooperative Work*, CSCW '06, pages 49–58. ACM, 2006.

[Fow03] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[GJM02] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[GMB+11] J. Gallardo, A. Molina, C. Bravo, M. Redondo, and C. Collazos. An ontological conceptualization approach for awareness in domain-independent collaborative modeling systems: Application to a model-driven development method. *Expert Systems with Applications*, 38(2):1099 –1118, 2011.

[Gru93] T. R. Gruber. A translation approach to portable ontology specifications. *KNOWLEDGE ACQUISITION*, 5:199–220, 1993.

[HM76] J. Hunt and M. MCillroy. An Algorithm for Differential File Comparison. Technical report, AT&T Bell Laboratories Inc., 1976.

[HMPR04] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28:75–105, 2004.

[HN98] A. Hunter and B. Nuseibeh. Managing Inconsistent Specifications: Reasoning, Analysis, and Action. *ACM Transactions on Software Engineering and Methodology*, 7(4):335–367, 1998.

[HR04] D. Harel and B. Rumpe. Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004.

[Joh88] R. Johansen. *GroupWare: Computer Support for Business Teams*. The Free Press, 1988.

[KÖ6] T. Kühne. Matters of (meta-) modeling. *Software and Systems Modeling*, 5:369–385, 2006.

[KHWH10] M. Kögel, M. Herrmannsdörfer, O. Wesendonk, and J. Helming. Operation-based Conflict Detection on Models. In *Proceedings of the International Workshop on Model Comparison in Practice @ TOOLS Europe 2010*. ACM, 2010.

[KKK+06] G. Kappel, E. Kapsammer, H. Kargl, G. Kramler, T. Reiter, W. Retschitzegger, W. Schwinger, and M. Wimmer. Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In *9th International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, volume 4199 of *LNCS*, pages 528–542. Springer, 2006.

[KN06] M. Kim and D. Notkin. Program Element Matching for Multi-version Program Analyses. In *Proceedings of the International Workshop on Mining Software Repositories (MSR'06)*, 2006.

[KR96] S. Khuller and B. Raghavachari. Graph and network algorithms. *ACM Computing Surveys*, 28(1):43–45, 1996.

[KRDM+10] D. Kolovos, L. Rose, N. Drivalos Matragkas, R. Paige, F. Polack, and K. Fernandes. Constructing and Navigating Non-invasive Model Decorations. In *Theory and Practice of Model Transformations (ICMT'10)*, volume 6142 of *LNCS*, pages 138–152. Springer, 2010.

[KT08] S. Kelly and J.-P. Tolvanen. *Domain-Specific Modeling: Enabling Full Code Generation*. Wiley-IEEE Computer Society Press, 2008.

[Lan11] P. Langer. *Adaptable Model Versioning Based on Model Transformations by Demonstration*. PhD thesis, Vienna University of Technology, 2011.

[LvO92] E. Lippe and N. van Oosterom. Operation-Based Merging. In *ACM SIGSOFT Symposium on Software Development Environment*, pages 78–87. ACM, 1992.

[LWWC11] P. Langer, K. Wieland, M. Wimmer, and J. Cabot. From UML Profiles to EMF Profiles and Beyond. In *Proceedings of the 49th International Conference on Objects, Models, Components, Patterns, TOOLS Europe 2011*. LNCS 6705, Springer, 2011.

[MCPW08] L. Murta, C. Corrêa, J. Prudêncio, and C. Werner. 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*, pages 25–30. ACM, 2008.

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

[MELS95] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

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

[MGH05] A. Mehra, J. Grundy, and J. Hosking. A Generic Approach to Supporting Diagram Differencing and Merging for Collaborative Design. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 204–213. ACM, 2005.

[MGHW10] I. Mistrík, J. Grundy, A. Hoek, and J. Whitehead. *Collaborative Software Engineering*. Springer, 2010.

[MM93] S. Minor and B. Magnusson. A Model for Semi-(a)Synchronous Collaborative Editing. In *In Proceedings of the Third European Conference on Computer Supported Cooperative Work*, pages 219–231. Kluwer Academic Publishers, 1993.

[MS89] D. Musser and A. Stepanov. Generic Programming. In *Symbolic and Algebraic Computation*, volume 358 of *LNCS*, pages 13–25. Springer, 1989.

[MS95] S. T. March and G. F. Smith. Design and natural science research on information technology. *Decision Support Systems*, 15(4):251 – 266, 1995.

[NE00] B. Nuseibeh and S. Easterbrook. Requirements engineering: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 35–46. ACM, 2000.

[NER01] B. Nuseibeh, S. M. Easterbrook, and A. Russo. Making inconsistency respectable in software development. *Journal of Systems and Software*, 58(2):171–180, 2001.

[NGTS10] F. Noyrit, S. Gerard, F. Terrier, and B. Selic. Consistent Modeling Using Multiple UML Profiles. In *Model Driven Engineering Languages and Systems (MoDELS'10)*, pages 392–406. Springer, 2010.

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

[Obj03] Object Management Group (OMG). Unified Modeling Language 2.0 (UML). `http://www.omg.org/cgi-bin/doc?ptc/2003-09-15`, 09 2003.

[Obj04] Object Management Group (OMG). Meta-Object Facility 2.0 (MOF). `http://www.omg.org/cgi-bin/doc?ptc/03-10-04`, 10 2004.

[Obj07] Object Management Group (OMG). Unified Modeling Language Infrastructure Specification, Version 2.1.2, `http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF`, 2007.

[Obj10] Object Management Group (OMG). Unified Modeling Language Specification, Version 2.3, `http://www.omg.org/spec/UML/2.3/`, 2010.

[OMW05] H. Oliveira, L. Murta, and C. Werner. Odyssey-VCS: A Flexible Version Control System for UML Model Elements. In *Proceedings of the International Workshop on Software Configuration Management*, pages 1–16. ACM, 2005.

[OS05] T. Oda and M. Saeki. Generative Technique of Version Control Systems for Software Diagrams. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'05)*, pages 515–524. IEEE, 2005.

[OWK03] D. Ohst, M. Welle, and U. Kelter. Differences between versions of uml diagrams. *ACM SIGSOFT Software Engineering Notes*, 28(5):227–236, 2003.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, December 1972.

[PWWZ11] C. Pichler, M. Wimmer, K. Wieland, and M. Zapletal. Towards Collaborative Cross-Organizational Modeling. In *Proceedings of the 2nd International Workshop on Cross Enterprise Collaboration (CEC 2011) @ BPM 2011*, volume 0099 of *LNBIP*. Springer, 2011.

[RAB+07] T. Reiter, K. Altmanninger, A. Bergmayr, W. Schwinger, and G. Kotsis. Models in Conflict – Detection of Semantic Conflicts in Model-based Development. In *Proceedings of International Workshop on Model-Driven Enterprise Information Systems @ ICEIS'07*, pages 29–40, 2007.

[RH09] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14:131–164, 2009.

[RKdV08a] M. Renger, G. Kolfschoten, and G.-J. de Vreede. Using Interactive Whiteboard Technology to Support Collaborative Modeling. In *Groupware: Design, Implementation, and Use*, volume 5411 of *LNCS*, pages 356–363. Springer Berlin / Heidelberg, 2008.

[RKdV08b] M. Renger, G. L. Kolfschoten, and G.-J. de Vreede. Challenges in Collaborative Modeling: A Literature Review. In *Proceedings of the 4th International Workshop CIAO! and 4th International Workshop EOMAS*, volume 10 of *Lecture Notes in Business Information Processing*, pages 61–77. Springer, 2008.

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2008.

[Sch06] D. Schmidt. Guest Editor's Introduction: Model-driven Engineering. *Computer*, 39(2):25–31, 2006.

[Sel07] B. Selic. A Systematic Approach to Domain-Specific Language Design Using UML. In *Int. Symposium on Object-Oriented Real-Time Distributed Computing*, pages 2–9. IEEE Computer Society, 2007.

[She03] H. Shen. *Internet-Based Collaborative Programming Techniques and Environments*. PhD thesis, Griffith University, 2003.

[SK88] R. W. Schwanke and G. E. Kaiser. Living With Inconsistency in Large Systems. In *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 98–118, 1988.

[SNTM08] A. Sebastian, N. F. Noy, T. Tudorache, and M. A. Musen. A Generic Ontology for Collaborative Ontology-Development Workflows. In *Proceedings of the 16th International Conference on Knowledge Engineering and Knowledge Management (EKAW'08)*, volume 5268 of *LNCS*, pages 318–328. Springer, 2008.

[SZ01] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In *Handbook of Software Engineering and Knowledge Engineering*, pages 329–380. World Scientific, 2001.

[SZN04] C. Schneider, A. Zündorf, and J. Niere. CoObRA – A Small Step for Development Tools to Collaborative Environments. In *Proceedings of the Workshop on Directions in Software Engineering Environments*, 2004.

[TELW10] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer. Conflict Detection for Model Versioning Based on Graph Modifications. In *Proceedings of the International Conference on Graph Transformations*, volume 6372 of *LNCS*, pages 171–186. Springer, 2010.

[TG04] J. Tam and S. Greenberg. A Framework for Asynchronous Change Awareness in Collaboratively-Constructed Documents. In *Groupware: Design, Implementation, and Use*, volume 3198 of *LNCS*, pages 67–83. Springer Berlin / Heidelberg, 2004.

[Tic88] W. Tichy. Tools for software configuration management. In *Proceedings of the International Workshop on Software Version and Configuration Control*, pages 1–20. Teubner Verlag, 1988.

[TNTM08] T. Tudorache, N. F. Noy, S. W. Tu, and M. A. Musen. Supporting Collaborative Ontology Development in Protégé. In *Proceedings of the 7th International Semantic Web Conference (ISWC'08)*, volume 5318 of *LNCS*, pages 17–32. Springer, 2008.

[TSS09] C. Thum, M. Schwind, and M. Schader. SLIM–A Lightweight Environment for Synchronous Collaborative Modeling. In *Proceedings of the Intenational Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, pages 137–151. Springer, 2009.

[Wen01] T. Wengraf. *Qualitative research interviewing: biographic narrative and semi-structured methods*. SAGE Publications, 2001.

[Wes10] B. Westfechtel. A Formal Approach to Three-way Merging of EMF Models. In *Proceedings of the International Workshop on Model Comparison in Practice @ TOOLS'10*, pages 31–41. ACM, 2010.

[WFK+11] K. Wieland, G. Fitzpatrick, G. Kappel, M. Seidl, and M. Wimmer. Towards an Understanding of Requirements for Model Versioning Support. *Accepted at International Journal of People-Oriented Programming*, 1(2), 2011.

[WLS+11] K. Wieland, P. Langer, M. Seidl, M. Wimmer, and G. Kappel. Turning Conflicts into Collaboration: Concurrent Modeling in the Early Phases of Software Development. *Submitted at Computer Supported Cooperative Work (CSCW)*, 2011.

[WRK+06] M. Wimmer, T. Reiter, H. Kargl, G. Kramler, E. Kapsammer, W. Retschitzegger, W. Schwinger, and G. Kappel. Lifting metamodels to ontologies - a step to the semantic integration of modeling languages. In *Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 528–542. Springer, 2006.

[Yin02] R. K. Yin. *Case study research: design and methods*. Applied Social Research Methods Series. SAGE Publications, 2002.

[ZGH+07] N. Zhu, J. Grundy, J. Hosking, N. Liu, S. Cao, and A. Mehra. Pounamu: A meta-tool for exploratory domain-specific visual language tool development. *Journal of Systems and Software*, 80:1390–1407, 2007.

Curriculum Vitae

Dipl.-Ing. Konrad Wieland, BSc

Zwinzstr. 1b/13
1160 Wien
Austria

Email: wieland@big.tuwien.ac.at
Web: http://www.big.tuwien.ac.at/staff/kwieland
Date of Birth: 24-Feb-1985
Nationality: Austria

Education

2009 - 2011	PhD Studies in Business Informatics Vienna University of Technology, Austria Supervision: o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel Univ.-Prof. Geraldine Fitzpatrick, PhD
2008 - 2009	Master Studies Business Informatics Vienna University of Technology, Austria Emphasis on: “Rechtliche Aspekte des IT-Managements”
2003 - 2008	Bachelor Studies Business Informatics Vienna University of Technology, Austria Emphasis on: “Decision Support in E-Government”
2003	Graduation from Secondary School, Vienna

Work Experience *(Excerpt)*

April 09 - Nov 11	Research Assistant and PhD Student Business Informatics Group at Vienna University of Technology, Austria Research Interests: MDSE, CSCW Teaching: Model Engineering, Web Engineering
Sept 08 - Mar 09	Project Participant at "ZKK" Vienna University of Technology, Austria Establishment of "Vienna PhD School of Informatics"
Jul 09 - Mar 09	Project and Research Assistant at E-Voting.CC Gesellschaft für elektronische Wahlen und Partizipation gGmbH
Jul 08 - Mar 09	Teaching Assistant at Information & Software Engineering Group at Vienna University of Technology, Austria Teaching: Business Process Modeling and Business Engineering, Project Management, Security
Mar 06 - Jun 08	Tutor at Information & Software Engineering Group at Vienna University of Technology, Austria Teaching: Business Process Modeling and Business Engineering
2005 - 2010	Sport Management at Sport Productions GmbH Organization of Tennis Tournaments and Referee
2003 - 2009	Austrian National Union of Students at Vienna University of Technology, Austria

Publications

2011

1. C. Pichler, K. Wieland, M. Wimmer, M. Zapletal: *"Towards Collaborative Cross-Organizational Modeling"*; accepted at 2nd International Workshop on Cross Enterprise Collaboration (CEC 2011) @ BPM 2011, volume 0099 of LNBIP, Springer, 2011.

2. K. Wieland, G. Fitzpatrick, G. Kappel, M. Seidl, M. Wimmer: *"Towards an Understanding of Requirements for Model Versioning Support"*; accepted at International Journal of People-oriented Programming, IGI Global, 2011.

3. P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. Emerging Technologies for the Evolution and Maintenance of Software Models, chapter *The Past, Present, and Future of Model Versioning*, IGI Global, 2011.

4. K. Wieland, P. Langer, M. Seidl, M. Wimmer, G. Kappel: *"Turning Conflicts into Collaborations: Concurrent Modeling in the Early Phases of Software Development"*; submitted at: Computer Supported Cooperative Work, Springer, 2011.

5. P. Langer, K. Wieland, M. Wimmer, J. Cabot: *"From UML Profiles to EMF Profiles and Beyond"*; in: Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11), 2011.

6. M. Brandsteidl, K. Wieland, C. Huemer: *"Novel Communication Channels in Software Modeling Education"*; in: "Models in Software Engineering - Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers", Lecture Notes in Computer Science Volume 6627, Springer, 2011, 40 - 54.

7. P. Brosch, H. Kargl, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel: *"Conflicts as First-Class Entities: A UML Profile for Model Versioning"*; in: "Models in Software Engineering - Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers", Lecture Notes in Computer Science Volume 6627, Springer, 2011, ISBN: 978-3-642-21209-3, 184 - 193.

2010

7. P. Brosch, H. Kargl, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel: *"Representation and Visualization of Merge Conflicts with UML Profiles"*; in: "Proceedings of the International Workshop on Models and Evolution (ME 2010) @ MoDELS 2010", Online Publication, 2010, 53 - 62.

8. P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer: *"Colex: A Web-based Collaborative Conflict Lexicon"*; in: "Proceedings of the 1st International Workshop on Model Comparison in Practice @ TOOLS 2010", ACM, 2010, 42 - 49.

9. P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel: *"Concurrent Modeling in Early Phases of the Software Development Life Cycle"*; in: "Proceedings of the 16th Collaboration Researchers' International Working Group Conference on Collaboration and Technology (CRIWG 2010)", Springer, 2010, 129 - 144.

10. P. Brosch, M. Seidl, K. Wieland: *"Guiding Modelers through Conflict Resolution: A Recommender for Model Versioning"*; Poster: SPLASH'10, Reno/Tahoe, Nevada, USA; 10-17-2010 - 10-21-2010; in: "SPLASH'10: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion", ACM, (2010).

11. P. Brosch, G. Kappel, M. Seidl, K. Wieland, M. Wimmer, H. Kargl, P. Langer: *"Adaptable Model Versioning in Action"*; "Modellierung 2010", GI, LNI 161 (2010); 221 - 236.

12. P. Langer, K. Wieland, P. Brosch: *"Specification, Execution, and Detection of Refactorings for Software Models"*; in: "Proceedings of the Work-in-Progress Session at the 8th

International Conference on the Principles and Practice of Programming in Java (PPPJ 2010)", CEUR-WS.org, 2010, Paper ID 5, 5 pages.

13. P. Brosch, K. Wieland, G. Kappel: *"Conflict Resolution in Model Versioning"*; Poster: 1st International Master Class on Model-Driven Engineering, Oslo, Norwegen; 09-30-2010 - 10-02-2010; in: "1st International Master Class on Model-Driven Engineering, Poster Session Companion", (2010), 17 - 18.

2009

14. K. Altmanninger, P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, M. Wimmer: *"Why Model Versioning Research is Needed!? An Experience Report"*; in: "Proceedings of the Joint MoDSE-MC-CM 2009 Workshop", (2009), Paper ID 8, 12 pages.

15. P. Brosch, M. Seidl, K. Wieland, M. Wimmer, P. Langer: *"The Operation Recorder: Specifying Model Refactorings By-Example"*; in: "Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009", ACM, 2009, 791 - 792.

16. P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, W. Schwinger: *"An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example"*; in: "Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)", Springer, LNCS 5795 (2009), 271 - 285.

17. P. Brosch, M. Seidl, K. Wieland, M. Wimmer, P. Langer: *"By-example adaptation of the generic model versioning system AMOR: how to include language-specific features for improving the check-in process"*; in: "Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009", ACM, 2009, 739 - 740.

18. P. Brosch, M. Seidl, K. Wieland, M. Wimmer, P. Langer: *"We can work it out: Collaborative Conflict Resolution in Model Versioning"*; in: "Proceedings of the 11th European Conference on Computer Supported Cooperative Work (ECSCW 2009)", Springer, (2009), 207 - 214.

19. K. Wieland: *"Advanced Conflict Resolution Support for Model Versioning Systems"*; in: "Proceedings of the Doctoral Symposium at MODELS 2009", School of Computing, Queen's University, 2009, Paper ID 4, 6 pages.