FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Conflict Resolution in Model Versioning

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin der Sozial- und Wirtschaftswissenschaften

eingereicht von

Petra Brosch
Matrikelnummer 0125826

an der
Fakultät für Informatik der Technischen Universität Wien

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

Diese Dissertation haben begutachtet:

(O.Univ.-Prof. Dr. Gerti Kappel) (Prof. Alfonso Pierantonio)

Wien, 14.02.2012

(Petra Brosch)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Conflict Resolution in Model Versioning

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktorin der Sozial- und Wirtschaftswissenschaften

by

Petra Brosch
Registration Number 0125826

to the Faculty of Informatics
at the Vienna University of Technology

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

The dissertation has been reviewed by:

(O.Univ.-Prof. Dr. Gerti Kappel) (Prof. Alfonso Pierantonio)

Wien, 14.02.2012

(Petra Brosch)

Erklärung zur Verfassung der Arbeit

Petra Brosch
Josef Madersperger Gasse 3, 2353 Guntramsdorf

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 Verfasserin)

Acknowledgements

Even if writing down a PhD thesis is performed by an individual, the whole process of conducting a PhD is the achievement of an excellent team. There are numerous decisions, which are influenced by many people. I owe my gratitude to all those, who have supported me to improve myself further.

First of all, outstanding work is never possible without outstanding assistance. I am grateful to my family for their kind support in every aspect of my life. Adele, Helmut, and Karin always encouraged me to aim high and move on. My family is an essential part of my life and equally share my success and failure. I am indebted to Harald for his unconditional love. He supported me with great understanding and patience while I spent several nights and weekends working on this thesis instead of spending my time with him. His calmness and balance makes me stronger and provides a dependable source of fortitude and motivation.

My special thanks go to Gerti Kappel, who trusted and encouraged me to conduct my PhD. Gerti's support unifies the freedom to explore new research fields and the constant advice to keep on track, based on her endless background of related work starting from the first steps in computer science until now.

I would like to thank the AMOR team for all the interesting and fruitful discussions and for jointly establishing the foundations of this thesis. Thanks to Philip Langer, Konrad Wieland, Horst Kargl, Martina Seidl, and Manuel Wimmer for having a great time coined by hard work and fun. I would never have appreciated the last years so much without them. Especially Martina and Manuel constantly accompanied my work with advice and constructive feedback.

It was also a pleasure to work with Alfonso Pierantonio, to whom I owe my gratitude for providing me with valuable and systematic feedback despite of the geographical distance.

I am very thankful to Michi, Katja, Marion, Sabine, and Vera for their support in all aspects of work besides research and for accompanying my last years. Regardless, whether there were technical, organizational, financial, or teaching related issues, all of them showed patience and provided me with kind assistance.

This work has not been possible without financial support of third-party funds. I am indebted to the Vienna University of Technology and the Austrian Federal Ministry of Science and Research for jointly funding the fFORTE WIT Program ultimately rendering this thesis possible. I would like to thank the Austrian Federal Ministry of Transport, Innovation, and Technology and the Austrian Research Promotion Agency for funding the AMOR project under grant FIT-IT-819584. Finally, I am thankful to the Vienna Science and Technology Fund (WWTF) for funding the FAME project under grant ICT10-018, ensuring further development and enhancements of the work conducted in this thesis.

to my grandmother

Abstract

In most engineering disciplines, models are built as pragmatic, yet precise abstractions of huge systems. The model building process requires multiple people jointly elaborating on artifacts, which are analyzed, used to communicate among stakeholders, and act finally as construction plan for realizing the modeled system. In the field of software engineering, modeling languages such as the *Unified Modeling Language* (UML) provide multiple diagrams to describe various viewpoints of a system in a concrete graphical notation. While the code-centric software engineering discipline adopted those models as visual language for describing the system under study, the increasing complexity of modern software systems accompanied by ever shorter time to market constraints has asked for new techniques. The upcoming *Model-Driven Engineering* (MDE) approach aims at additionally exploiting models to automatically generate executable code. This paradigm shift lifts models to first-class citizens within the whole engineering process, effectively shaping the primary artifact of change undergoing the collaborative refinement from informal sketches to blueprints. This upgrowth intrinsically demands tool support for managing the models' history including merging of parallel evolved models. Optimistic versioning systems, which are already successfully applied for the management of source code, handle both issues. However, applying those systems to models fails due to the models' graph-based structure. Consequently, first dedicated model versioning systems emerged. Although current model versioning systems provide decent conflict detection facilities, they (1) ignore the graphical representation of the models, and (2) neglect conflict resolution by totally shifting the responsibility to the user. Yet, the central role of models unifying the human-centric, collaborative abstraction and design process with the computation-centric process of generating executable systems, demands proper mechanisms to foster validity and quality of the merged model.

In this thesis, we first analyze specifics of model versioning and elaborate on the notion of conflict to improve conflict resolution respecting the central role of models. To cope with the human-centric aspect, we present a conflict aware merge strategy to calculate a tentatively merged *conflict diagram* as accelerator for conflict resolution retaining the graphical representation of the model. The conflict diagram unifies non-conflicting changes and materializes merge conflicts in form of annotations, rendering a coherent picture of the model's evolution. To further support the conflict resolution process, we elaborate on a *conflict resolution recommender system* on top of the conflict diagram, which recommends automatically executable conflict resolution patterns. Finally, to satisfy validity conditions of the computation-centric aspect, we establish a *formal framework* based on graph transformation theory, to showcase the feasibility of our approach.

Kurzfassung

In nahezu jeder Ingenieurdisziplin werden große Systeme mittels pragmatischer und doch präziser Modelle abstrahiert. Der Modellbildungsprozess erfordert die Zusammenarbeit im Team und resultiert letztendlich in Artefakten zur Analyse und Kommunikation zwischen Interessengruppen, die schließlich als Bauplan zur Realisierung des Systems eingesetzt werden. Modellierungssprachen wie die *Unified Modeling Language* (UML) stellen speziell für den Bereich der Softwareentwicklung zahlreiche Diagramme zur grafischen Beschreibung der verschiedenen Gesichtspunkte eines Systems zur Verfügung. Die codezentrierte Softwareentwicklung benutzt solche Modelle im traditionellen Sinn. Die steigende Komplexität moderner Softwaresysteme und die immer kürzer werdenden Vorlaufzeiten erfordern jedoch neue Techniken. Das aufstrebende Paradigma der *Modellgetriebenen Softwareetwicklung* (MDE) verfolgt das Ziel Modelle zur automatischen Generierung von ausführbaren Code auszunutzen. Modelle werden dadurch zum zentralen Artefakt im gesamten Entwicklungsprozess und durchleben die kollaborative Verfeinerung von informellen Skizzen bis hin zum fertigen Bauplan. Dadurch wird die werkzeugunterstützte Verwaltung der Änderungen und das Zusammenführen paralleler Änderungen immer wichtiger. Optimistische Versionierungssysteme, welche bereits erfolgreich zur Verwaltung von Code eingesetzt werden, unterstützen beide Aufgaben. Allerdings können sie aufgrund der graphbasierten Struktur von Modellen nicht direkt auf Modelle angewendet werden. Folglich entstanden spezielle Modellversionierungssysteme, die bereits brauchbare Mechanismen zur Konflikterkennung bieten. Dennoch gibt es Schwachpunkte: (1) die grafische Repräsentation wird ignoriert und (2) die Auflösung von Konflikten wird in die Verantwortung der Benutzer abgeschoben. Da Modelle nun die direkte Drehscheibe zwischen dem menschenzentrierten Systementwicklungsprozess und dem berechnungszentrierten Generierungsprozess bilden, werden geeignete Mechanismen zur Förderung der Qualität und Gültigkeit des zusammengeführten Modells benötigt.

Um der zentralen Rolle der Modelle im Konfliktauflösungsprozess gerecht zu werden, untersuchen wir in dieser Arbeit zuerst die Spezifika der Modellversionierung und den Konfliktbegriff. Die menschenzentrierten Anforderungen meistern wir durch die konfliktbewusste Berechnung eines *Konfliktdiagramms*, das durch die Erhaltung der grafischen Repräsentation als Basis für die Konfliktauflösung dient. Das Konfliktdiagramm vereint unproblematische Änderungen und materialisiert Konflikte mittels Annotationen, um ein nachvollziehbares Bild der gesamten Evolution zu schaffen. Auf das Konfliktdiagramm aufbauend präsentieren wir einen *Empfehlungsdienst*, der ausführbare Muster zur Konfliktauflösung vorschlägt. Schlussendlich genügen wir den berechnungszentrierten Anforderungen, indem wir einen *formalen Rahmen* basierend auf Graphtransformationstheorie schaffen, der die Anwendbarkeit unseres Ansatzes zeigt.

Contents

CHAPTER 1

Introduction

With the increasing adoption of model-driven engineering in academia and in practice, the need for model versioning support grows ever louder. While current approaches focus on detecting conflicting changes in parallel development, the resolution part remains neglected.

This thesis copes with conflict resolution in model versioning. In particular, we investigate conflict resolution support using the concrete graphical syntax of models and guide modelers through conflict resolution by recommending proper conflict resolution patterns. We start this chapter with a brief overview on modeling and model-driven engineering to establish a common understanding on the key notions necessary for the remainder of this thesis. We argue why model versioning is needed and discuss the problems which are addressed in this thesis. Finally, we give an overview on our contributions and sketch the outline of the thesis.

1.1 Motivation

Modeling is the task of building abstract representations of certain observations and reflects the typical way how human cope with reality [153,192]. The roots of modeling dates back to the beginning of mankind with numbers as first traceable representatives. Anthropologists claim that modeling is the most important skill distinguishing mankind and less developed races [192]. Models enable talking and reasoning about things, that are not (yet) here, what explains attractiveness of adopting models in several engineering disciplines. According to Stachowiak [207], a model is a representation of an *original* phenomenon, item, system, etc. reflecting a relevant subset of its properties and is built with some pragmatism in mind to use the model instead the original for a special purpose. This definition allows only for *descriptive models*, describing an original, which already exists. In contrast, *prescriptive models* may serve as blueprint or construction plan to detect flaws in a system before the system is built, i.e., at an early stage of a project, when costs for improvement in form of corrections and extensions are low. Thus, the definition of Stachowiak is extended to allow an original yet to be built [142].

1.1.1 Model-Driven Engineering

According to Brooks [28], software is inherently complex. This complexity arises on the one hand from the essential complexity of the problem domain itself, and on the other hand from accidental complexity due to inadequate techniques to represent the problem domain. To cope with this complexity, software development should be approached as other engineering disciplines, such as civil engineering, coming along with dedicated management and organizational methodologies, theories, tools, and abstraction techniques [25]. Hence, similar to using construction plans in civil engineering, also in computer science, models are soon used to represent and design multiple viewpoints of a system under study before actually constructing it [201,202]. In traditional code-centric software engineering practice, models are created during the design-phase using existing graph-based modeling languages, such as the *Unified Modeling Language* (UML) [114]. Thanks to the models' graphical notation, software is no longer invisible, what enables the communication among stakeholders [142]. In fact, although those models have a clear data structure due to their graph-based abstract syntax, only their associated graphical concrete syntax in form of diagrams is used, which act as blueprint for implementing the system. As recognized soon, one major capability of modeling is the high level of automation regarding the generation of code. The idea of MDE was born; the repetitive task of translating diagrammatic blueprints to code should be automated and developers should be enabled to concentrate on creative and non-trivial tasks which computers cannot do, i.e., creating those blueprints. An early attempt towards MDE was made in the 1980s with computer-aided software engineering (CASE) tools, following the goal to directly generate executable systems based on graphical models [193]. However, CASE integrated general-purpose modeling methods, which poorly mapped to the target platforms. Hence, the generated code was complex and manual customization, extension, and debugging was hardly possible. Domain-specific modeling environments aim at closing the gap between CASE and the problem domain by jointly capturing the requirements of domain-specific modeling and its target platform. Model-integrated computing (MIC) targets the costly development of domain-specific modeling environments by effectively combining generic processes, such as analysis methods, and domain-specific models. It was soon realized that the development of domain-specific environments was itself a domain and metamodeling environments to create domain-specific environments were established [139,147,167,206]. In fact, the term *meta* denotes that an operation is applied on itself, e.g., a discussion about conducting a discussion is called meta-discussion [142]. In a similar vein, metamodeling is referred to modeling modeling languages.

In an endeavor to establish a commonly accepted set of key concepts and to preserve interoperability between the rapidly growing number of domain-specific development environments, the Object Management Group (OMG) released the specification for *Model Driven Architecture* (MDA) [109], standardizing the definition and usage of (meta-)metamodels as driving factor of software development. To this end, the OMG proposes a layered organization of the metamodeling stack similar to the architecture of formal programming languages [22], as depicted in Figure 1.1. The meta-metamodel level M3 manifests the role of the *Meta-Object Facility* (MOF) [111] as the unique and self-defined metamodel for building metamodels, i.e., the meta-metamodel ensuring interoperability of any metamodel defined therewith. MOF may be compared to the Extended Backus–Naur Form (EBNF) [92], the metagrammar for expressing

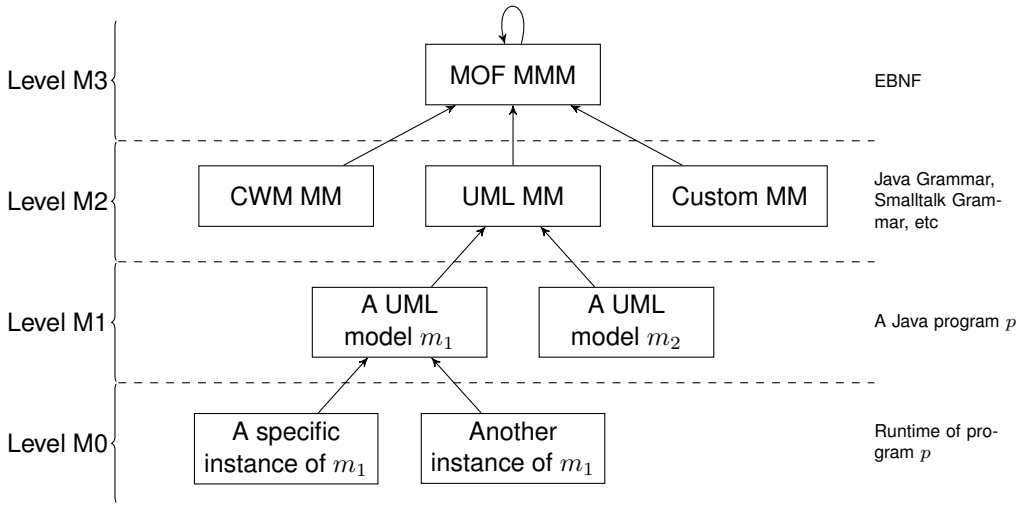


Figure 1.1: Metamodeling Layers; adapted from [22]

programming languages. The metamodel level M2 contains any metamodel defined with MOF, including the UML, the Common Warehouse Metamodel (CWM) [108] also specified by the OMG, and any custom domain-specific metamodel. A metamodel at this level conforms to a definition of a programming language with EBNF, such as the Smalltalk grammar or the Java grammar. A metamodel defines the abstract syntax of the modeling language and is usually supplemented with one or more concrete syntactics. A graphical concrete syntax is again defined as metamodel at level M2 and frames graphical elements such as shapes and edges by extending a standardized diagram interchange metamodel [171, 172] and associates those elements with corresponding elements of the abstract syntax metamodel. The model level M1 contains any model built with a metamodel of level M2, e.g., a UML model. An equivalent for a model is a specific program written in any programming language defined in EBNF. Finally, the concrete level M0 reflects any model based representation of a real situation. This representation is an instance of a model defined in level M1. We may again draw the parallel to the formal programming language architecture. Level M0 corresponds to all dynamic execution traces of a program of level M1.

The major benefit of MDA is to decouple system specifications from the underlying platform [40]. In this way, the specification is much closer to the problem domain and not bound to a specific implementation technique. This benefit is maximized when domain-specific modeling languages are employed. Thus, the MDA specification [109] differentiates at level M2 languages for *Computation Independent Model* (CIM), *Platform Independent Model* (PIM), and *Platform Specific Model* (PSM) to quantify the abstraction quality of a model. While a CIM provides a fully computation independent viewpoint close to the domain in question, a PIM approximates the system description in a technology neutral manner. A PSM eventually unifies the PIM with the specifics of the underlying platform to be used as specification for code generation.

To bridge metamodeling and programming languages, and to justify MOF as interoperability standard, the OMG provides a standardized way for exchanging MOF based artifacts. OMG's standard for *XML Metadata Interchange* (XMI) [112] defines a mapping of any (meta)model ex-

pressed in MOF to the *Extensible Markup Language* (XML) [49]. The specification of MOF itself is divided into the kernel metamodel *Essential MOF* (EMOF) and the more expressive *Complete MOF* (CMOF) [111]. EMOF is closely related to object-oriented programming languages, such as Java, which allows a straightforward mapping, as implemented in [60, 177]. Especially the Eclipse Modeling Framework (EMF) with its reference implementations for EMOF [60] and UML [63] fosters several adjacent subprojects for arbitrary MDA tasks, such as querying and comparing models, building textual and graphical modeling editors, etc. leading to increasing adoption in academia and in practice. As the resulting vast amount of interrelated models may not be managed by hand, MDA is complemented with a mechanism, called *model transformation* for further computation of models, such as updating, synchronizing, translating models to other models, or translating models to code.

In order to express the holistic idea of modeling not only software architectures, but also capturing system behavior, we use the more general term MDE instead of MDA in the remainder of the thesis.

1.1.2 The Need for Model Versioning

With the practical realization and application of MDE techniques, modeling activities have matured from creating pretty pictures to producing artifacts translatable to executable code. Whereas those pretty pictures have been used for mere documentation purposes in order to communicate ideas, requirements, and designs, models are now lifted to first-class citizens taking an important role in the MDE development process. The multifaceted application of models for the entire software lifecycle leads to the refinement from informal sketches over blueprints to executable code. This upgrowth intrinsically demands tool support for managing evolution [95, 208], because models get as important as textual code in the traditional software engineering process. Whenever bugs are fixed or when functionalities are added or extended, the underlying models have to be updated. Consequently, the same kind of change management as successfully applied for textual code is required. Since models and code differ in many aspects, the techniques and tools available for code can be hardly reused.

Models provide a higher abstraction level than textual code necessary to deal with the complexity of modern software, what explains the attractiveness of applying MDE in the development process. In general, modern software is so huge that it cannot be built by one single modeler, but a team or teams of engineers are necessary to satisfy the time constraints given by the customers and the application domain [101]. Teamwork is a very complex interaction with the high risk making a project fail if it does not work properly [100, 102]. For traditional software engineering, several tools implementing the pessimistic or optimistic version control paradigm have been proposed for supporting the collaboration between multiple developers.

Pessimistic version control systems store the history of artifacts on a central server, which comes along with the benefit of sharing files within a team. Whenever an artifact is edited by one team member, the artifact is locked and read-only for the others. Conflicts are avoided by forbidding that more than one modeler edit an artifact at the same time. The modelers not in charge of the lock have to wait until it is released. Unnecessary idle times of the modelers might be the consequence.

In contrast, *optimistic version control systems* (VCS) handle both issues, namely they sup-

port handling the evolution of various artifacts by managing their different versions and they make effective collaboration in teams possible. VCS allow developers to work independently of other team members on their personal local copy and changes are merged at a later point in time. The benefit of working in parallel without idle times due to locked artifacts comes at the price of incorporating the isolated changes of the modified artifact. Whenever changes do not commute, conflicts might be reported and resolved manually, which is a tedious and error-prone task in general. Nevertheless, the merging of code works satisfying well in practice. For models, the situation is different.

1.2 Problem Statement

As models may be serialized into a textual representation, the first pragmatic attempts for realizing model evolution support as well as collaborative modeling support were built on top of text-based versioning systems like Subversion [93] and CVS [183], which were already successfully applied for code. It quickly turned out that XMI serializations [112] are neither an appropriate representation for machines to detect conflicts, nor an appropriate representation for humans to understand and resolve conflicts [4, 17]. As consequence, dedicated model versioning systems emerged, operating on a graph-based representation of models [5]. While graph-based approaches advantage precise conflict detection, the expected boost for manual conflict resolution is still absent. Conflict resolution is nevertheless cumbersome and error-prone, for the following reasons.

First, conflicting changes are mostly indicated in the abstract syntax of the model, while modelers are familiar with the concrete graphical syntax only. For meaningful integration of all intentions behind the changes, all changes and the resulting conflicts must be well understood, which is even harder without the familiar view in the concrete graphical syntax carrying the mental map [59] of the model, i.e., the personal view of the modeler on the model.

Second, conflict resolution often requires more than solely deciding which of the two conflicting changes shall be included in the merged version. To integrate a meaningful combination of both changes, the modeler has to provide a completely new version of the model by hand. However, conflict resolution support in state-of-the-art versioning systems moves hardly beyond the choices “keep mine”, “keep theirs”, “take all changes”, or “abandon all changes”.

What is still missing, is an approach to merge conflicting models and their respective diagrams to provide a unified view on their partly contradictory evolution. Thereby, merge conflicts should be indicated in the concrete syntax and the modeler should remain in her familiar modeling environment. Further, supportive conflict resolution mechanisms should be provided based on this unified view.

1.3 Contribution

The overall goal of this work is to improve conflict resolution support in model versioning. We aim at rendering a coherent and comprehensible representation of merge conflicts and at mitigating obstacles in their resolution by guiding modelers through conflict resolution.

The building blocks realizing this goal are integrated in our research prototype, the adaptable model versioning system AMOR [7]. AMOR is a national funded research project, jointly elaborated at the Vienna University of Technology[1], the Johannes Kepler University Linz[2], and SparxSystems[3]. Overall, concepts and techniques developed in the course of the AMOR project resulted in three PhD theses. Information on change and conflict detection may be found in [145]. Collaborative aspects of conflict resolution are presented in [223]. The contributions presented in this thesis may be summarized as follows.

Contribution 1: Conflict Categorization. The most fundamental building block in optimistic versioning is a thorough understanding on *changes* and possibly resulting *conflicts*. Especially in model versioning, dealing with graph structures instead of mere text files, it does not suffice to describe conflicts to certain overlapping parts in the serialized artifacts. We therefore first analyze which kinds of change may be performed on models and what actually constitutes a conflict. We found out that there are quite different yet limited conflict types, not reducible to one single definition. To better reason about conflicts, we established a *conflict categorization*, effectively grouping those conflict situations in layers and subcategories giving hints how to detect them.

Contribution 2: Conflict Diagram. State-of-the-art model versioning tools lack at representing merge conflicts in a reproducible manner. Main deficiencies are the mismatch of modeling using the concrete graphical syntax of a modeling language, and merge support provided at the abstract syntax level only. Further, conflicting models are visualized side-by-side, delegating the responsibility for merging to the user. To overcome this undesirable situation, we developed a conflict aware merge strategy applicable to arbitrary modeling languages to calculate a tentatively, merged version of the model in the abstract, as well as in the concrete syntax of the model. The resulting model called *conflict diagram* effectively unifies non-conflicting changes as starting point for conflict resolution. In order to keep information on changes and conflicts, we developed dedicated annotations realized as UML profile to allow a model-based representation of the model's evolution. In this way we lifted changes and conflicts determined in the conflict categorization to first-class citizens of the versioning process.

Contribution 3: Conflict Resolution Recommender System. Further analysis of the conflict categorization reveals, that not only types of certain conflict situations reoccur, but that also the pattern for their resolution may be re-applied. Consequently, we elaborated a *conflict resolution recommender system* matching certain *conflict patterns* and suggesting suiting *conflict resolution patterns.* Those patterns are described by exploiting the model-based representation of changes and conflicts inherent to the conflict diagram. Thus, the developed conflict diagram provides besides the pure visualization also the foundation for the conflict resolution recommender system. We established a formal framework based on graph transformation theory and showcase the feasibility of the approach.

[1] `http://www.tuwien.ac.at`
[2] `http://www.jku.at`
[3] `http://www.sparxsystems.eu/`

1.4 Thesis Outline

This thesis is structured to stepwise elaborate the aforementioned contributions. Please note that the content of some chapters partly overlaps with our previous work jointly elaborated by the project team.

Chapter 2: Optimistic Model Versioning. This chapter presents the foundations of versioning in order to establish a common terminology. We review the history of versioning in general and the active research field of model versioning in particular. We finally identify challenges which have to be overcome for putting model versioning into practice. Parts of this chapter have been published in [31, 39].

Chapter 3: A Tour on AMOR. We subsequently discuss details on our research prototype, the adaptable model versioning system AMOR. AMOR is jointly developed at the Vienna University of Technology, the Johannes Kepler University Linz, and our industry partner SparxSystems. Starting with a thorough analysis of considered changes and conflict types (cf. Contribution 1), we present AMOR's flexible workflow and how it copes with the aforementioned challenges. Parts of this chapter have been published in [4, 31, 32].

Chapter 4: Model Transformation. Before we discuss our contributions on a technical level, we review concepts and techniques for model transformation to establish a common basis. We place emphasis on the formal basis of model transformation, namely graph transformation, especially on the algebraic graph transformation approach. The reader familiar with the concepts of graph transformation may skip this chapter and use it solely as reference book.

Chapter 5: Conflict Aware Merge. This chapter handles the generation of conflict diagrams forming Contribution 2. We start this chapter with a survey on related works regarding merge strategies complemented by approaches regarding the preservation of the diagram layout along sequences of model versions. We then elaborate on a model-based representation of merge conflicts and a conflict aware merge strategy for merging models and diagrams, jointly rendering a unified view called conflict diagram. Parts of this chapter have been published in [33].

Chapter 6: Conflict Resolution Support. Starting with a survey of related work on recommender systems in general, we learn from recommender systems dedicated to assist developers in software engineering tasks, how we may tackle Challenge 3, i.e., assist modelers in the task of merging. We discuss how conflict resolution patterns are defined based on the conflict diagram and present a conflict resolution recommender system. Finally, we show the feasibility of our approach and present a formal framework based on graph transformation theory, effectively realizing the conflict resolution recommender system. Parts of this chapter have been published in [37].

Chapter 7: Conclusion. Finally, we draw conclusions, summarize our contributions, and point to future research directions.

CHAPTER 2

Optimistic Model Versioning

Over the last years, models turned out to be one cornerstone for the development of complex software systems. According to the MDE paradigm, models are not only used as informal sketches to communicate ideas; they are stepwise refined and extended to support the whole lifecycle of a system. Hence, models may get huge and heavily interrelated or overlapping with other artifacts expressing different viewpoints or abstraction levels of the system. Consequently, the continuous evolution of these of models is accompanied by intensive teamwork.

The evolution of software models induces a plethora of challenging research issues. Only when these problems are solved, the techniques of MDE are able to fully exploit their potential in practice. Otherwise the advantages of MDE are relativized by time-consuming and cumbersome management tasks which are already well supported for traditional development based on textual code. One of these challenges is optimistic model versioning.

In this chapter, we review the active research field of model versioning, establish a common terminology, introduce the various techniques and technologies applied in state-of-the-art versioning systems, and conclude with open issues and challenges which have to be overcome for putting model versioning into practice.

2.1 The Problem of Collaborative Work

During the software development lifecycle, the various software artifacts under construction are subject to successive changes. Consequently, tool support for managing the evolution of these artifacts is indispensable [90, 158]. To this end, the discipline of Software Configuration Management (SCM) provides tools and techniques for making evolution manageable [215]. Amongst others, these tools include Version Control Systems (VCS) whose origins may be dated back to the early 1970s. Since then, the discipline of versioning is an active research topic generating a variety of different concepts, formalisms, and technologies.

The aims of versioning approaches are threefold. First, versioning systems maintain a *historical archive* of the different versions an artifact adopts during its development. With this archive, it is possible to undo harmful modifications by restoring previous development states.

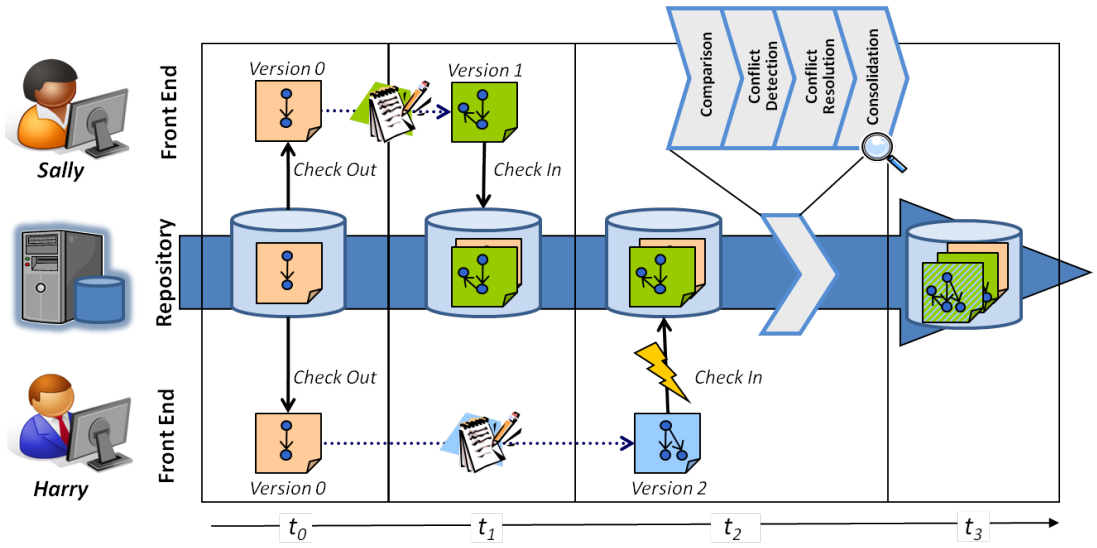


Figure 2.1: The Optimistic Versioning Process

Second, versioning systems support handling *different development branches*, which are needed for building different software variants. Third, versioning approaches manage the *parallel evolution* of software artifacts performed by a (distributed) team of developers. In this thesis, we focus on the latter aim.

In general, two different versioning strategies exist to cope with the concurrent evolution of one artifact. When pessimistic versioning is applied, an artifact is locked while it is changed by one developer. Since other developers cannot perform any changes while the artifact is locked, conflicts are completely avoided with this strategy. However, the drawbacks are possible idle times for developers waiting for the release of a locked artifact. To avoid such idle times, optimistic versioning allows the developers to change the same artifact in parallel and independently of each other. The typical workflow of optimistic versioning is depicted in Figure 2.1. Two users of the optimistic versioning system, Harry and Sally, check out the same artifact at time t0. Both modify the checked out Version 0 independently of each other. After Sally has finished, she checks in her modified version (Version 1) at t1. When Harry also tries to check in his modified version, he first has to merge his version (Version 2) with the latest version in the repository (Version 1). Merging, often a time-consuming and tedious task, is the price to pay, when concurrent modifications by several users are allowed. In general, the merge process may be divided into four steps: (1) identifying the differences between two concurrently modified versions, (2) detecting conflicts between these two modifications, (3) resolving these conflicts either automatically or manually, and finally (4) creating a new consolidated version which, in the best case, combines all intentions behind all concurrently performed modifications.

A wealth of works have been published which contribute in solving various versioning challenges (cf. [48] for a survey). However, the versioning research has mainly focused on the management of textual artifacts like source code so far. For such artifacts, a line-oriented processing

of files has largely been adopted by practitioners in the past. More fine-grained approaches, in which not lines but, e.g., words are considered as atomic units of comparison, have not gained much attention in practice. If the artifacts put under version control are graph-based artifacts like models, the situation is different. Here a more precise consideration of the model elements is necessary to obtain accurate reports on the performed modifications and potential conflicts between concurrently performed changes. When line-based VCS approaches are applied on the textual serialization of models, the result is unsatisfactory in general. Not least because of the models' graph-based structure, single changes on the model may result in multiple changed lines in the textual serialization like XMI [112]. Considering lines as unit of comparison, the information stemming from the graph-based structure is destroyed and associated syntactic and semantic information is lost. However, fine-grained versioning of software models is only limited supported by current modeling tools. Yet, with the increasing employment of model-driven engineering techniques [193] for the development of large software systems, the call for adequate infrastructural means supporting the effective management of collaboration when working on software models grows even louder.

When putting models under version control, the graph-based structure is not the only issue needing special treatment. In contrast to code, the abstract syntax of a model is separated from the layout information including the arrangement and the visualization of the dedicated model elements. Since models are normally developed and refined in a graphical syntax, fine-grained versioning considering both the model and the diagram is required, which is unfortunately not supported by current state-of-the-art VCSs. Furthermore, merging different diagram layout versions raises several challenges and open questions. No concise definition has been established yet which concurrently performed layout changes should be reported as conflicts and, furthermore, how these conflicts should be presented to the user. Consequently, the modifications and conflicts are usually presented within the tree representation of a model, abstaining from the visual information valuable for the human user. The visualization of differences and conflicts in the concrete syntax of the modeling language still remains an open issue.

Due to all these current drawbacks, modeling mostly remains a one-(wo)man show in order to avoid model merges. This, however, is in contradiction with the term "model-driven engineering", because an engineering discipline must include—besides other important features—the ability to build software systems that are so large that they have to be built by a team or teams of engineers [99]. Fortunately, the urgent need for a suitable infrastructure supporting optimistic model versioning has been widely recognized and first solutions start to emerge. Barrett et al. [15] have performed an evaluation of the versioning capabilities of commercial modeling tools and have provided an experience report. In [4] we have also compared different state-of-the-art tools and we have explored which kind of conflicts may be detected by recent tools. Since model versioning is urgently needed in practice, much effort is spent in this research area, resulting in a rapid evolution and maturation of model versioning approaches. New tools and approaches emerge in a very short cycle, contributing to an increase in the understanding of model versioning concepts. Based on this understanding, we aim to survey the past and present of model versioning in order to state important future challenges.

To lay out the foundations of versioning, we collect and unify important concepts, terminologies, and design possibilities regarding artifact and change representation from the past

achievements. To get a picture of the present situation, we then survey state-of-the-art model versioning systems. With this background, we are able to derive future challenges for model versioning systems. To underpin our conjectures on potential challenges, we have not only identified these challenges from literature and from our own experiences when building a model versioning system, but we have also conducted 15 expert interviews and an online survey which provided valuable insights on the users' requirements and expectations.

2.2 Foundations of Versioning

In the long history of active research on software versioning, diverse formalisms and technologies emerged. To categorize this variety of different approaches, Conradi & Westfechtel [48] 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 & 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 & Westfechtel distinguish between extensional and intentional versioning. *Extensional versioning* deals with the reconstruction of previously created versions and, therefore, concerns version identification, immutability, and efficient storage. All versions are explicit and have been checked in once before. *Intentional versioning* deals with flexible automatic construction of consistent versions from a version space. In other words, intentional versioning allows for annotating properties to specific versions and querying the version space for these properties in order to derive a new product consisting of a specific combination of different versions.

In this thesis, we only consider extensional versioning in terms of having explicit versions, because this kind of versioning is paramountly applied in practice nowadays. Furthermore, we focus on the *merge phase* in the optimistic versioning process (cf. Figure 2.1). 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

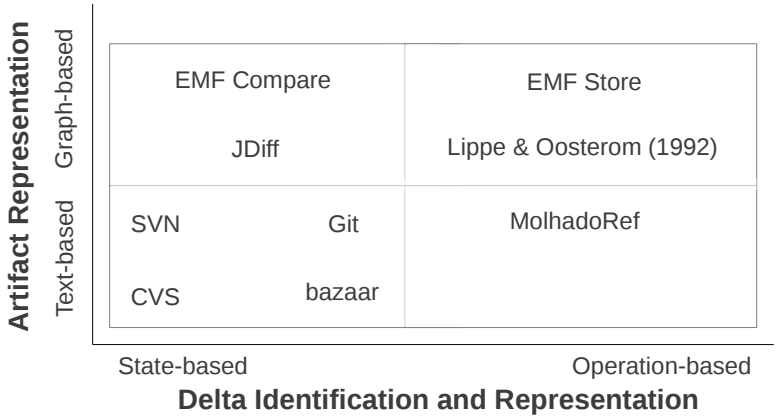


Figure 2.2: Categorization of Versioning Systems

are *identified*, *represented*, and *merged* in order to create a consolidated version. 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 2.1) and the two successors (cf. Version 1 and 2 in Figure 2.1).

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 [157].

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 a more appropriate 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 [157] 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. [157]) 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 since 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 is orthogonal to the first one and considers *how deltas are identified and merged* in order to create a consolidated version. This dimension is agnostic of the unit of versioning. Therefore, a versioned element might be a line in a flat text file, a node in a graph, or whatsoever constitutes the representation used for merging.

State-based merging compares the states, i.e., versions, of a software artifact to identify the differences (deltas) between 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 2.1), called two-way merging, or to three states (including their common ancestor Version 0 in Figure 2.1), called three-way merging. Two-way merging cannot identify deletions since the common origin 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. [136] for an overview of matching techniques). Model matching, or more generally the graph isomorphism problem is NP-hard (cf. [135]) 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. Hence, n_0 denotes an element in the common origin version and n_1 or n_2 indicate an element in the two revised versions, respectively.

Algorithm 2.1 iterates through each element n_0 in the initial version V_0 of a software artifact. The following two lines retrieve the elements matching with n_0 from the two modified versions V_1 and V_2. However, there might be no match for n_0 in V_1 or V_2 since it might have been removed. If n_0 has a match in both versions V_1 and V_2, the algorithm checks if it has been modified in the versions V_1 and V_2. If the matching element is different from the origin element n_0, i.e., it has been modified, in one and only one of the two versions V_1 and V_2, the modified element is used for creating the merged version. If the matching element is different in both

Input: Common origin model V_0, two revised models V_1 and V_2
Output: The merged model version V_m

```
1  foreach n0 ∈ V0 do
2      n1 ← match(n0 in V1);
3      n2 ← match(n0 in V2);
4      if hasMatch(n0 in V1) ∧ hasMatch(n0 in V2) then
5          if diff(n0, n1) ∧ ¬ diff(n0, n2) then
6              Use n1 in Vm
7          end
8          if ¬ diff(n0, n1) ∧ diff(n0, n2) then
9              Use n2 in Vm
10         end
11         if diff(n0, n1) ∧ diff(n0, n2) then
12             Raise Update/Update conflict
13         end
14         if ¬ diff(n0, n1) ∧ ¬ diff(n0, n2) then
15             Use n0 in Vm
16         end
17     end
18     if hasMatch(n0 in V1) ∧ ¬ hasMatch(n0 in V2) then
19         if diff(n0, n1) then
20             Raise Delete/Update conflict
21             else Remove n0 in Vm
22         end
23     end
24     if ¬ hasMatch(n0 in V1) ∧ hasMatch(n0 in V2) then
25         if diff(n0, n2) then
26             Raise Delete/Update conflict
27             else Remove n0 in Vm
28         end
29     end
30     if ¬ hasMatch(n0 in V1) ∧ ¬ hasMatch(n0 in V2) then
31         Remove n0 in Vm
32     end
33 end
34 foreach n1 ∈ V1 do
35     if ¬ hasMatch(n1 in V0) then Add n1 to Vm
36 end
37 foreach n2 ∈ V2 do
38     if ¬ hasMatch(n2 in V0) then Add n2 to Vm
39 end
```

Algorithm 2.1: State-based Merge Algorithm

versions, an Update/Update conflict is raised by the algorithm. If the matching element has not been modified at all, the origin unit n_0 is used for the merged version. Next, the algorithm checks if there is no match for n_0 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_0 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_1 and V_2, which have no match in the origin version V_0 and which, consequently, have been added in V_1 or V_2.

Operation-based merging does not operate on the states of an artifact. Instead, the operation sequences which have been concurrently applied to the origin 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 to record composite operations such as refactorings (e.g., [140]). The knowledge on applied refactorings may significantly increase the quality of the merge as stated by Dig et al. [57]. The downside of operation recording is the strong dependency on the applied editor, since it has to record each performed operation and it has to provide this operation sequence in a format which the merge approach is able to process. The directly recorded operation sequence might include obsolete operations such as updates to an element which will be removed later on. Therefore, many operation-based approaches apply a cleansing algorithm to the recorded operation sequence for more efficient merging. The operations within the operation sequence might be interdependent because some of the operations cannot be applied until other operations have been applied. As soon as the operation sequences are available, operation-based approaches check parallel operation sequences (Version 0 to Version 1 and Version 0 to Version 2) for commutativity to reveal conflicts (cf. [149]). 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 [140].

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

2.2.2 Selected Representatives

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 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 2.3.

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 [183] and SVN [93] 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 [121]. 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 [43] and bazaar [182], 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 [57], 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* [8], and one for software models, namely *EMF Compare* [40, 62]. 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 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 & Oosterom [149]. 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* [140] 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 2.3.

2.2.3 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 [67] 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

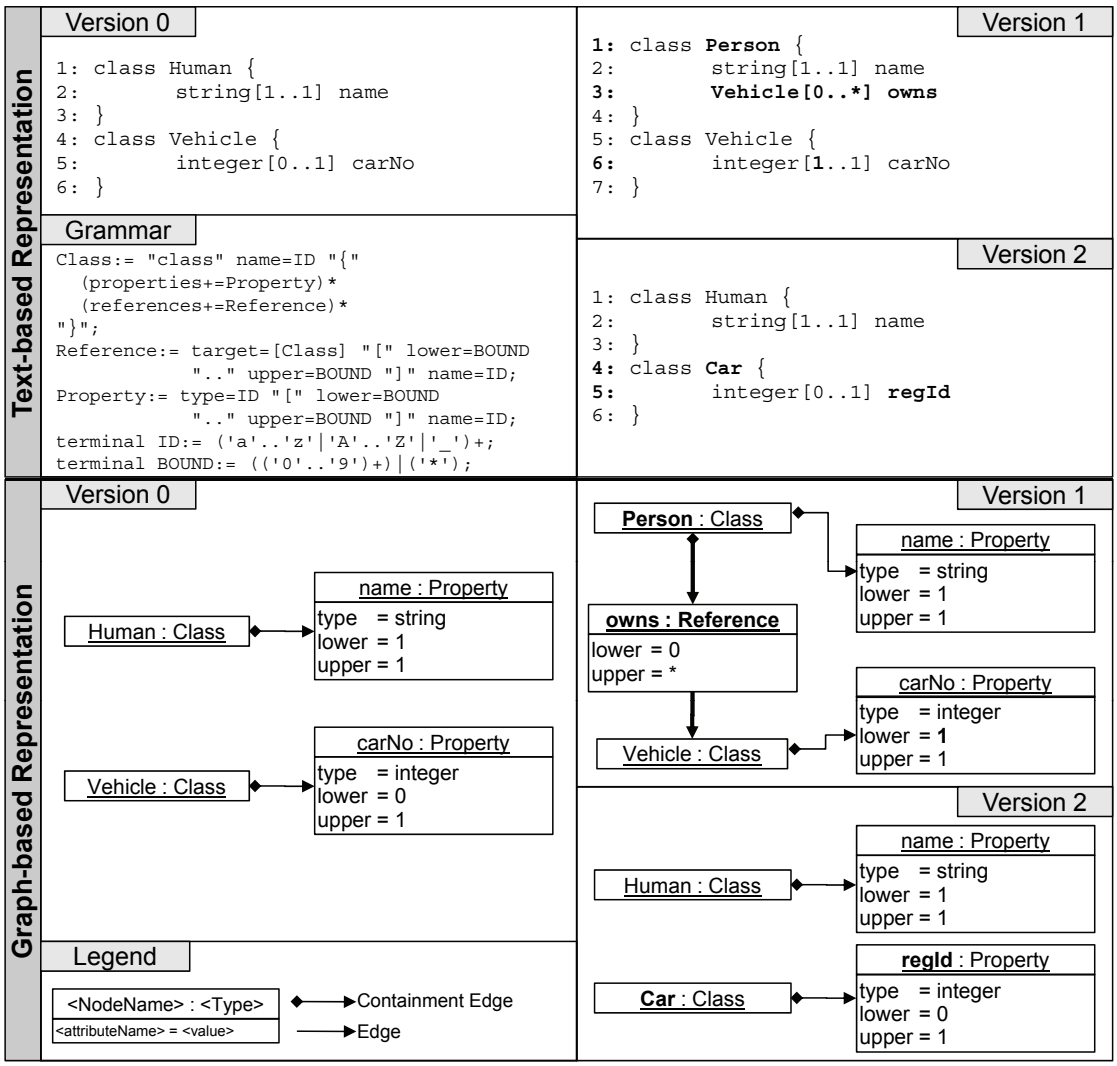


Figure 2.3: Versioning Example

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

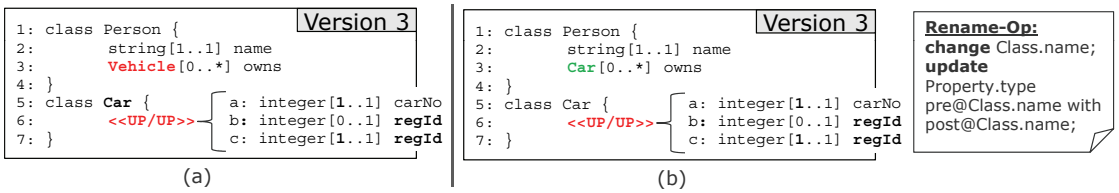


Figure 2.4: Text-based Versioning Example: (a) state, (b) operation

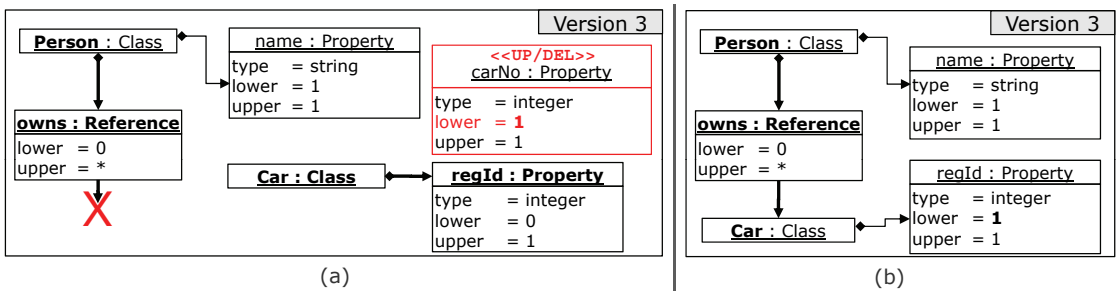


Figure 2.5: Graph-based Versioning Example: (a) state, (b) operation

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

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

dence 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 [144, 213, 222]. 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, several challenges still have to be addressed in future. First, 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. 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. Second, software models are usually created and maintained using a (textual or graphical) concrete syntax rather than using the abstract syntax. When model versioning approaches solely work on the graph-based abstract syntax representation, a huge gap accrues between the representation used for merging and the representation which developers use and consequently are familiar with. This hinders developers to efficiently grasp identified differences and raised conflicts. Moreover, the graphical representation of a model carries a lot of additional information as discussed by Misue et al. [163]. These considerations, however, are mostly ignored by recent model versioning approaches.

2.3 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, provide the basics for dedicated model versioning systems, which are urgently required in times when model-driven engineering technologies mature and find their way from academia into industry. In this section, we review the status-quo in the active research landscape on model versioning and discuss the current achievements and advancements.

2.3.1 Overview

Over the years, several graph-based versioning systems for software models have been proposed, which are shortly introduced in the following. We consider approaches that provide differencing and merging facilities as well as complete model versioning systems including a repository, and we highlight their distinguishing features.

Approach of Alanen and Porres. One of the earliest works on the versioning of models such as UML [114] was the paper by Alanen & Porres [3], who presented various metamodel independent algorithms for difference calculation, model merging as well as conflict resolution. Metamodel independence is achieved by exploiting MOF's reflection interface [111] for looking up metaclasses and their respective metafeatures. 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 origin 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.

Approach of Oda & Saeki. The version control support proposed by Oda & Saeki [173] builds upon the facilities offered by a meta-CASE tool which allows the construction of mod-

eling 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.

Approach of Ohst, Welle, and Kelter. Within their merge algorithm, Ohst et al. [175] 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.

Approach of Mehra, Grundy, and Hosking. The approach of Mehra et al. [156] 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.

Approach of Cicchetti, Di Ruscio, and Pierantonio. Cicchetti et al. [46] 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 con-

flict 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 [53]. ADAMS may be integrated via specific plugins 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. [54] present an ADAMS plugin 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 necessary. In ADAMS, the layout information is also considered as a model and is therefore also set under version control.

AMOR. The model versioning system AMOR presented in [32] implements a conflict detection component for Ecore based models 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) 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 [33, 35]. Furhter details on AMOR are discussed in Chapter 3.

CoObRA. The Concurrent Object Replication framework CoObRA developed by Schneider et al. [194] realizes optimistic versioning for the UML case tool Fujaba [107]. 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 [40], which is part of the Eclipse Modeling Framework Technology (EMFT) project [61], 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.

IBM Rational Software Architect (RSA). The RSA [123], 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 [122]. 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. [140], which has been initially developed as part of the Unicase project [219], 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. [140], 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. [176] 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 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. [165] 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. [186] 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.

2.3.2 Features

When considering the model versioning systems above, it becomes obvious that the individual systems set different focus on the challenges they tackle although all of them follow the same goal of providing sophisticated versioning facilities for software models. Table 2.1 provides an overview of the distinguished features offered by the various systems grouped into four categories which we discuss in the following. Applicable features are indicated with the checkmark symbol (✓); partly applicable features are marked with the tilde symbol (∼). Empty boxes state that the specific feature is not applicable to the approach.

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

Table 2.1: State-of-the-Art Evaluation

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 following chapter. 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 and ADAMS 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, when applied to the origin model, the revised model may be recreated.

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.
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. [175] and Mehra et al. [156]. However, these approaches require the implementation of special editor extensions. Thus, to the best of our knowledge, we started in [33] the first attempts of tackling these challenges using UML Profiles which are integrated in the AMOR model versioning system.
Automatic Conflict Resolution: Most approaches require manual conflict resolution. In AMOR and the approach of Cicchetti et al. [46], 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 [173] and Mehra et al. [156] 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 MOF or in Ecore. Other approaches consider one language only, for example, CoObRA is implemented for models formulated in Fujaba, only.
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 chapter.
Resolution Strategies: Only few versioning systems provide automatic conflict resolution facilities. In some versioning systems, rudimentary rules stating how to react on a certain conflict are hard coded. AMOR and the approach of Cicchetti et al. allow specifying conflict-specific resolution strategies.

2.4 Summary and Future Challenges

Whereas for some versioning approaches deployable software or even the source code is available, others are still subject to ongoing development work. However, all of them contribute important ideas and concepts for building reliable model versioning systems. 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 [60], which offers a sophisticated environment for the development of such model manipulation tools. As we have seen, 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, there are also open issues not yet covered by any model versioning approach. As we have learned from literature and from interviewing modeling experts and practitioners [223], the main challenges which have to be solved in order to realize reasonable model versioning support may be summarized as follows.

Challenge 1: Notion of Conflict. Interestingly, the notion of *conflict*, a very central concept when building a versioning system, is hardly explicitly discussed. In most systems, differences and conflicts are not explicitly stated as first-class citizens, but they are hard-coded within the corresponding algorithms. Conflicts are mainly reduced to Update/Update and Delete/Update conflicts. Concerning the model correctness and consistency after the merge process, all cur-

rent approaches refer to external solutions which for example ensure the conformance to the metamodel.

Challenge 2: Adaptable Versioning System. Existing model versioning systems are often inflexible with respect to the trade-off between generic applicability and proper versioning support. The systems are either generic, i.e., applicable to any modeling language, being however, characterized by limited versioning support, or, to exhibit enhanced versioning support, bound to a specific modeling language. This inflexibility is even worse because of the rapidly growing number of domain-specific modeling languages (DSMLs), entailing either the straightforward application of an existing generic system or requiring big efforts in developing a dedicated VCS from scratch. Thus, a big challenge in model versioning is to develop a framework which on the one hand may be used out of the box in a generic sense, and on the other hand, may be adapted for a certain modeling language on basis of a set of meaningful and well-defined extension points.

Challenge 3: Intention Aware Versioning. When merging two concurrently modified versions, ideally the merged version should constitute a combination of all intentions each modeler had in mind when performing their changes. Merging intentions is often more than just naively combining all non-conflicting atomic changes of both sides. Changing a model is realizing a certain goal rather than simply modifying some parts of it. However, capturing the modeler's intention from a set of changes is a major challenge. First steps in this direction aim at treating composite operations such as model refactorings as first-class entities, because composite operations are more than the set of atomic changes applied in a certain sequence. Usually, they consist of preconditions and an intended final state (i.e., the operation contract). Therefore, detecting applications of well-defined composite operations and regarding their conditions during the merge is a first valuable step towards intention aware versioning which is, for instance, done by Dig et al. [57] for code versioning.

Challenge 4: Supportive Conflict Resolution. Current model versioning approaches focus on detecting conflicts only, totally neglecting the resolution thereof. Consequently, conflict resolution remains a tedious and error-prone task, which is manually performed by the modeler who did the later check-in. As the way how conflicts are resolved often set the course for future development, conflict resolution should gain more attention. First, different strategies for managing conflict resolution should be considered, where not one modeler alone is responsible for resolving a conflict but both modelers whose changes are in conflict resolve the conflict collaboratively. Second, with current tool support, merging the different versions manually poses a very time-intensive, repetitive challenge in order to obtain a consistent artifact which meaningfully integrates the work of all involved developers. In the field of software merging, approaches for automating the merge process already exist. They include the calculation of all possible combinations of parallel performed operations leading to a valid version and merge policies to privilege specific operations or operations of specific users (cf. [157] for a survey). These approaches only ask for user input in undecidable cases and reduce the manual resolution effort to a minimum. Cicchetti et al. [46] already proposed to automate model merging by defining certain

conflict patterns and corresponding reconciliation strategies. However, those approaches do not consider resolution strategies that go beyond a recombination of conflicting changes. In some situations conflicts may be resolved more reasonable by providing a completely new version requiring means of semi-automatic conflict resolution.

Challenge 5: Conflict Resolution in Concrete Syntax. As modelers are usually familiar with the concrete graphical syntax of a model but not with computer internal representations like XMI or abstract syntax graphs, the demand for versioning support directly in the concrete syntax grows ever louder. Some dedicated approaches have been proposed for visualizing differences of models (cf. [156, 175]). They construct a dedicated view using the concrete syntax, which combines and highlights changes of both models using coloring techniques. However, these approaches require for heavy-weight extensions of the modeling editors which seems to be a barrier for adoption. When visualizing conflicts in the concrete syntax it should, of course, improve the conflict resolution process even when the model is getting large to quickly find changes and to understand which changes have led to a certain conflict. For this, some interactive techniques in the modeling editors are needed to understand changes and conflicts within the concrete syntax. In addition, for concentrating on the resolution of one conflict, the computation of a specialized conflict view which only incorporates one conflict as well as the necessary context to understand the model fragment seems to be a way to go.

Challenge 6: Diagram Versioning. Not only the visualization of changes and conflicts has to be tackled by model versioning systems, but also the diagram layout information has to co-evolve with the model. Thus, the diagram layout information has to be put under version control. Due to the fact that diagram information is nowadays also represented in terms of models, model versioning features may be reused. However, this approach neglects the nature of 2D diagram layout by representing only x/y coordinates as Integer values. Thus, for larger models, this approach seems to be not accessible for modelers. For providing appropriate diagram versioning support, the following sub-challenges arise. First, means for visualizing changes of the layout information are needed within modeling editors by, e.g., interactively showing the changes by animation techniques. Ohst et al. [175] and Mehra et al. [156] have proposed first approaches going in this direction. Second, no general notion has been established yet which concurrently performed layout changes are in fact contradicting changes, e.g., considering the inconvenience of small unintended changes when a modeler moves an element one pixel without intending it. Third, appropriate resolution mechanisms for layout conflicts have to be established. Here, the most challenging question is how to preserve the mental map [163] of both modelers between the initial version, the two parallel changed versions, and the finally merged version.

Challenge 7: Tolerating Conflicts. In contrast to code, models are often used in an informal manner for sketching ideas and discussing design alternatives in the early phases of software development. One major benefit of models in this phase is to manage and improve communication among team members by establishing common domain knowledge. In this context, it is desirable to keep all or at least many of the model changes, even if they are conflicting. The reason for this is that conflicts may not only be seen as negative result of collaborative development,

but rather as necessary means for identifying aspects of systems which need further analysis or which need to reflect different viewpoints of different stakeholders. Thus, conflicts may help to develop a common understanding of the requirements for to be developed systems. A versioning system for tolerating conflicts should allow to explicitly represent and persist conflicts as well as to track conflicts to resolve them later on. A promising resource for learning how to tolerate conflicts is the field of multi-perspective development, where inconsistencies can be compared with conflicts in the area of versioning. 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 (cf. [13, 91, 170, 200]). First step in tolerating conflicts in the context of versioning systems for UML have been made in [33, 35].

Challenge 8: Avoiding Conflicts. Conflicts are completely avoided by pessimistic versioning. However, this approach may be undesireable because of arising idle times and the aforementioned positive effects of the team members' different viewpoints. An alternative to locking artifacts is synchronous development, where conflicts are avoided through awareness [56]. Synchronous development requires dedicated development environments informing modelers as soon as they are concurrently working on an overlapping part of the model. The modelers then have the possibility to consolidate their planned changes with the help of collaborative tool features such as chats. However, it depends on the purpose of a model and on its development stage, to define which parts of the model are considered as overlapping. For example, when models are used for implementation of object-oriented systems, object-orientation provides a perfect separation of concerns into packages, classes, operations and so on. However, in the requirement phase of a project, where the goal is to find the structure for the system to implement, further partitioning techniques and design approaches are needed. An alternative paradigm would be not to design one model, but to express the different concerns in separate models and then try to integrate these different models to form the whole system. In the field of aspect-oriented requirements engineering, there exist promising approaches separating different parts of a systems by concerns, which could be adopted for model partitioning in the context of model versioning. Finding appropriate techniques for effectively separating concerns suiting the different phases of software development poses a challenge.

Challenge 9: Consistency Aware Versioning. When merging two independently evolved versions of one model, it might occur—even if both versions are consistent on their own—that the merged version violates consistency rules of, e.g., the underlying modeling language. In order to produce consistent models only, changes leading to inconsistencies should be specially treated in the merge process. Unfortunately, those inconsistent changes might not be overlapping and therefore harder to detect. Beside language constraints, also user-defined constraints might be violated during the merge. User-defined constraints may be stated in requirements documents or in additional models, capturing partly overlapping static and dynamic viewpoints of the system. To avoid unexpected or contradicting properties of the merged model, a combination of syntactic and semantic comparison of two model versions is highly valuable. However, currently no commonly agreed formal semantics exists for widespread employed modeling languages like UML. Even worse, the semantics of semantic is heavily discussed within the modeling commu-

nity [116]. The definition of a formal semantics for a comprehensive set of the UML, including intra-model dependencies, is a challenge on its own. Further, violations are hardly detectable until the merged version is produced. Thus, another challenge is to trace back to the relevant changes causing the violation. For doing this, first, the model elements have to be identified where the evaluation of a consistency rule fails. Second, changes operating on those model elements have to be selected from the complete change set, and third, if several changes have been performed on these elements, the relevant subset of changes has to be distinguished which actually is producing the violation. First promising approaches for defining relationships between models to detect inconsistencies are already available [150].

Overall, we conclude with the observation that the research area of model versioning still offers a multitude of tough challenges despite the many achievements which have been made until today. These challenges must be overcome in order to obtain solutions which ease the work of the modelers in practice. The final aim is to establish methods which are so well integrated in the development process that the modelers themselves do not have to care about versioning tasks and that they are not distracted from their actual work by time consuming management activities. Therefore, different facets of the modeling process itself have to be reviewed to gain a better understanding of the inherent dynamic. Versioning is about supporting team work, i.e., about the management of people who work together in order to achieve a common goal. Consequently, versioning solutions require not only the handling of technical issues like adequate differencing and conflict detection algorithms or adequate visualization approaches, but also the consideration of social and organizational aspects. Especially in the context of modeling, the current versioning approaches have to be questioned and eventually revised. Here the requirements posed on the versioning systems may depend on the intended usage of the models. Models are not always created with the intention to obtain an exact representation of the world, but to obtain a shared understanding on a very specific aspect. Besides the graphical visualization and the separation of concerns which is carried to extremes by the means of different views like in the various diagrams of UML, this fuzziness is a feature which sets models apart from mere source code of traditional, textual programming languages demanding new methods for the management of collaborations.

CHAPTER 3

A Tour on AMOR

In optimistic versioning, multiple developers are allowed to modify an artifact at the same time. On the one hand this approach increases productivity as the development process is never stalled due to locks on an artifact. On the other hand conflicts may arise when it comes to merging the different modifications into one consolidated version. As we learned in the previous chapter, the resolution of such conflicts is not only cumbersome and time-consuming, but also error-prone. Especially if the artifacts under version control are models, little support is provided by standard VCSs. Hence, model versioning has become an active research field with the endeavor to improve this situation.

In this chapter, we present the enhanced versioning process of the adaptable model versioning system AMOR. We first show how AMOR tackles the challenges, we identified when surveying the emerging research field of model versioning (cf. Section 2.4). AMOR aims to combine the advantages of both generic and language-specific VCSs by providing a generic framework exhibiting well-defined extension points for including language-specific features. We discuss how AMOR is configured in order to obtain a precise conflict report which enables smart conflict resolution. AMOR offers different strategies for supporting conflict resolution. (1) When using semi-automatic conflict resolution, the user of AMOR chooses one of the recommended, automatically executable resolution patterns. (2) The manual resolution may be performed in a single setting or in collaboration. (3) Besides immediate conflict resolution, AMOR allows to tolerate conflicts for a while, which is especially useful in the early phases of software development projects, where a common understanding is not yet established.

However, prerequisits for building a model versioning system are thorough insights about changes and conflicts, which we discuss in the following.

3.1 The Notion of Conflict

As discussed in the previous chapter, 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

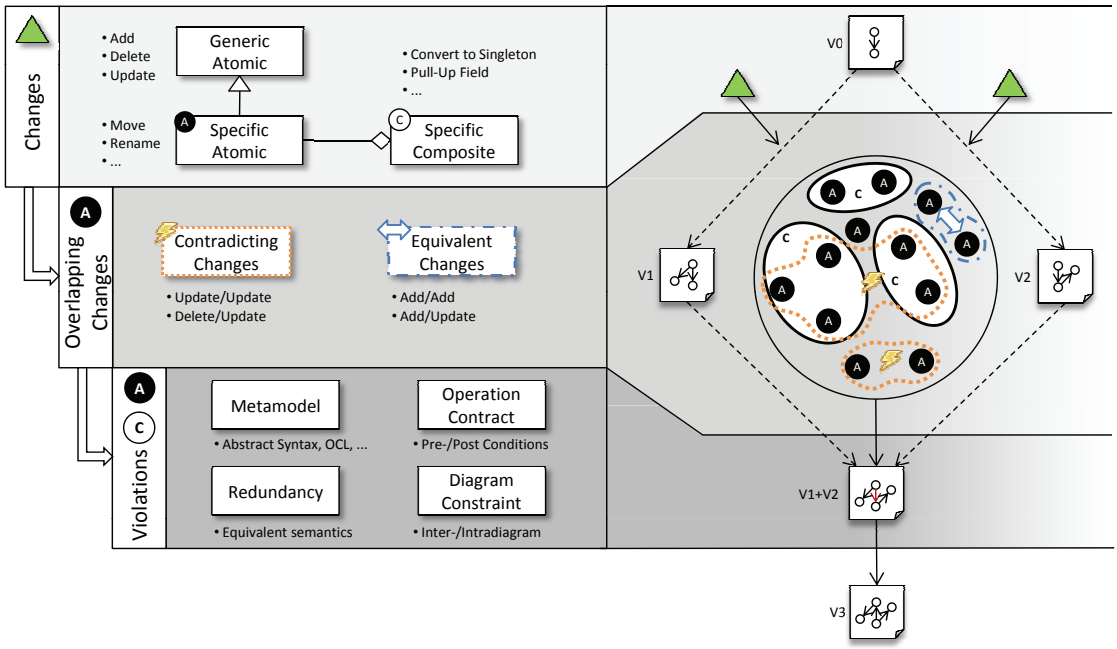


Figure 3.1: Change and Conflict Categorization

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. The basis for any merge conflict are though at least two changes. Therefore, in this section we first analyze which changes may be applied to a model and then present a comprehensive categorization of conflicts based on our previous work [31, 33, 34].

3.1.1 Changes in Model Versioning

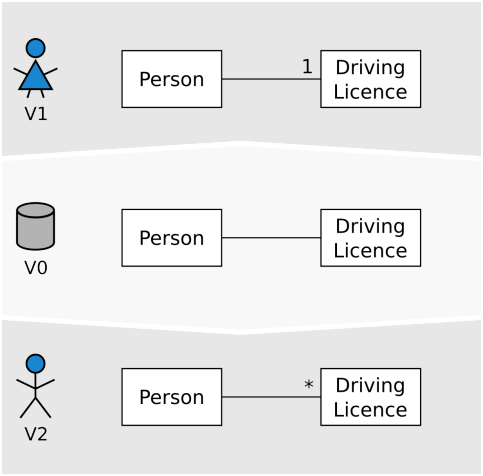
Changes describe all modifications performed on the common base version V0 resulting in the modified versions V1 and V2 (indicated by the symbol Δ in Figure 3.1). Due to the variety of existing modeling languages, changes may shape differently. One strength of the model-driven engineering paradigm is the common self-defined meta-meta-layer MOF [111]. Every metamodel defined according OMG's proposed MDA standard share the same metatypes and may be reflectively analyzed. Thus, a generic versioning infrastructure makes definitely sense, as it is highly reusable. However, the quality of the change detection directly correlates with the quality of the merged version. Thus, we classify changes according to two orthogonal dimensions. The first dimension represents their dependency on an underlying modeling language. A change is *generic* if it may be applied to any model irrespectively of the modeling language. In contrast, a *specific* change depends on a certain metamodel. Consequently, a specific change is a specialization of a generic change. The second dimension considers the divisibility of an operation by distinguishing between *atomic* and *composite* changes.

From the resulting four combinations of these classifications, we omit *generic composite* because a composite operation makes only sense in the context of a specific modeling language. *Generic atomic changes* comprise the primitive atomic operations `add`, `delete`, and `update`. They may be performed on model elements (`add`, `delete`) and model properties (`update`) independently of the modeling language, i.e., the underlying metamodel. Generic atomic operations build the basis for more complex and language specific operations. *Specific atomic changes* are indivisible language-dependent operations like `rename` and `move`. The operation `rename` modifies a specific property which assigns—according to the underlying metamodel—a name to a model element. The operation `move` changes the containment of a model element which also requires knowledge on the underlying metamodel. Note that in certain environments `move` is realized as composite operation if allowed by the metamodel. *Specific composite changes* comprise a set of atomic changes which should be applied in combination. Typical representatives for composite changes are refactorings, as they impart more than the sum of all assembled atomic changes. To identify specific composite changes like refactorings is challenging, but extremely important for the quality of the overall merge result [36, 57]. It enables a more compact representation of the difference report by folding atomic operations which belong to a composite operation. Thus, detecting applied composite operations allows a faster and better understanding of the modeler's original intention. Furthermore, it facilitates smarter conflict detection and resolution leading to a reduction of conflict alerts and identification of otherwise unrevealed merge issues.

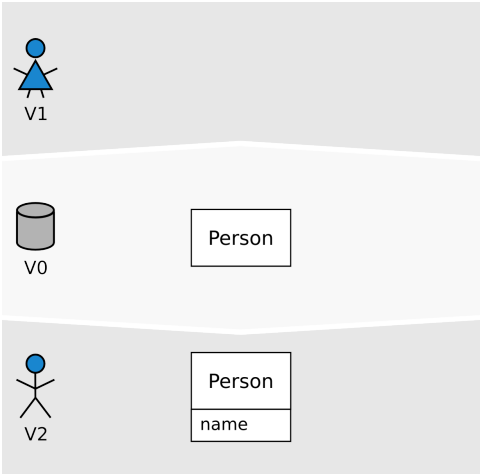
3.1.2 Conflicts in Model Versioning

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 [149]. 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 by Mens [157]. 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, but reports conflicts causing parse errors. Semantic merging goes one step further and reflects the semantic annotation of the parse tree 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. MolhadoRef [57] copes with such structural conflicts in software merging by distinguishing atomic changes and refactorings. Refactorings are replayed after merging atomic changes. Hence, atomic changes are either incorporated in the refactoring or a conflict becomes evident.

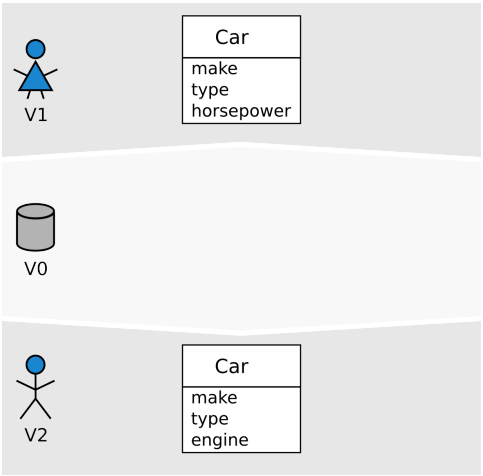
From the analysis of many conflict scenarios, we identified that such a stepwise refinement for detecting merge conflicts for models is also valuable, as only directly overlapping changes are detectable by analyzing the respective change sets. The merged model, however, may manifest inconsistencies, violating some constraint. Thus we categorize conflicts in model versioning into two main groups, namely *overlapping changes* and *violations*.



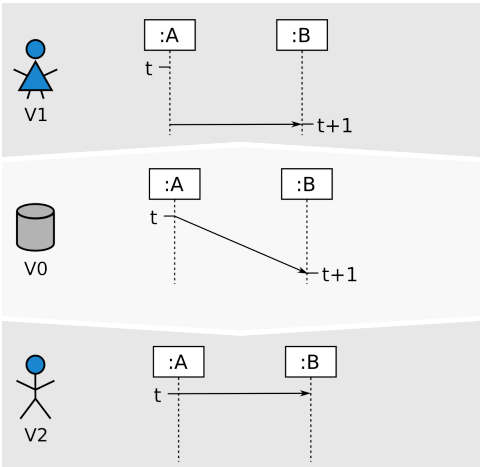
(a) Contradicting Update/Update



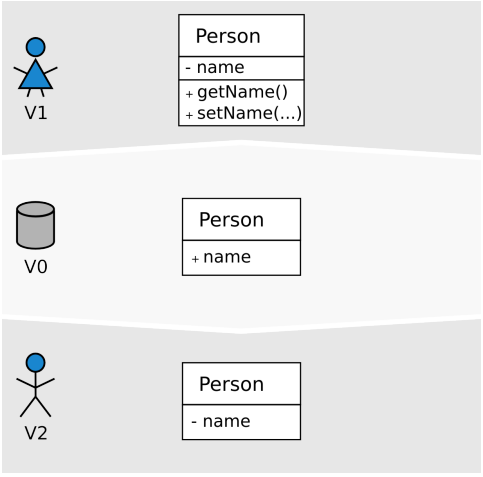
(b) Contradicting Delete/Update



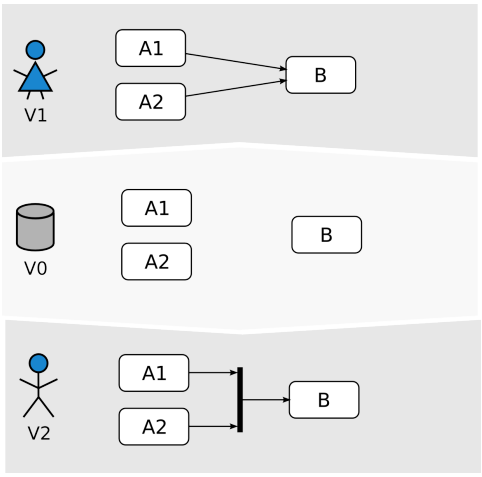
(c) Equivalent Add/Add



(d) Metamodel Violation



(e) Operation Contract Violation



(f) Redundancy

Figure 3.2: Conflict Examples

Conflicts due to 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 result 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 3.2a). 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 3.2b.

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 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 3.2c, an Add/Add conflict should be raised.

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 they are not revealed before a merged version is built. Further, additional knowledge is necessary. This knowledge regards the underlying modeling language and the modeled domain.

Metamodel Violations. In some cases, concurrent changes which do not overlap lead to an inconsistent model with respect to formal language constraints like the metamodel itself of additional OCL constraints. For an example of a metamodel violation, consider a UML sequence diagram with a message taking one time unit to be sent (cf. Figure 3.2d). Both modelers change the message to be sent without taking time. However, they do not agree on the point in time, when the message will be sent and change the `sendEvent` and `receiveEvent`, respec-

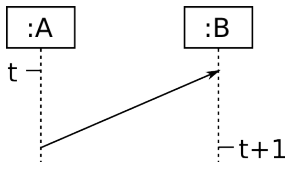


Figure 3.3: Naive Merge of Example in Figure 3.2d

tively, which do not syntactically overlap. A naive merge would produce a message going back in time (cf. Figure 3.3), which is technically not realizable.

Operation Contract Violations. 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 3.2e shows an example for an operation contract violation. One modeler applies an enclose variable refactoring to all public properties, i.e., `name`, of class `Person`. The refactoring sets the properties to private and generates public getter and setter methods. A parallel change setting the property `name` to private invalidates the operation contract and a conflict should be reported.

Redundancy. Different modeling concepts may also express equivalent semantics. For example, the actions `A1` and `A2` of the UML activity diagram depicted in Figure 3.2f 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. Thus, the two versions have equal semantics, i.e., the execution of `A1` and `A2` is required to execute `B`. However, naively merging these two variants would result in a redundant model (cf. Figure 3.4).

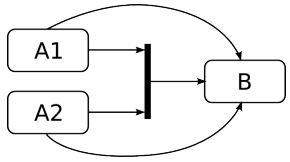


Figure 3.4: Naive Merge of Example in Figure 3.2f

Diagram Constraint. User-defined constraints within the model or stated in related models like use case descriptions, requirement specifications, or other views, restricts the modeled domain and therefore may be used to detect further violations. Consider for example different aspects modeled in various UML diagrams rendering together a holistic picture of a system. In some cases, a change in one diagram may contradict other diagrams.

3.2 Adaptable Model Versioning

The adaptable model versioning system AMOR [7,32] is a research prototype jointly developed at the Vienna University of Technology[1], the Johannes Kepler University Linz[2], and SparxSystems[3], the vendor of Enterprise Architect. AMOR aims at providing an adaptable framework to orchestrate various components realizing an optimistic model versioning process. In this way, AMOR copes with the challenge of adaptable versioning. We discuss the details on AMOR in the following.

[1] `http://www.tuwien.ac.at`
[2] `http://www.jku.at`
[3] `http://www.sparxsystems.eu/`

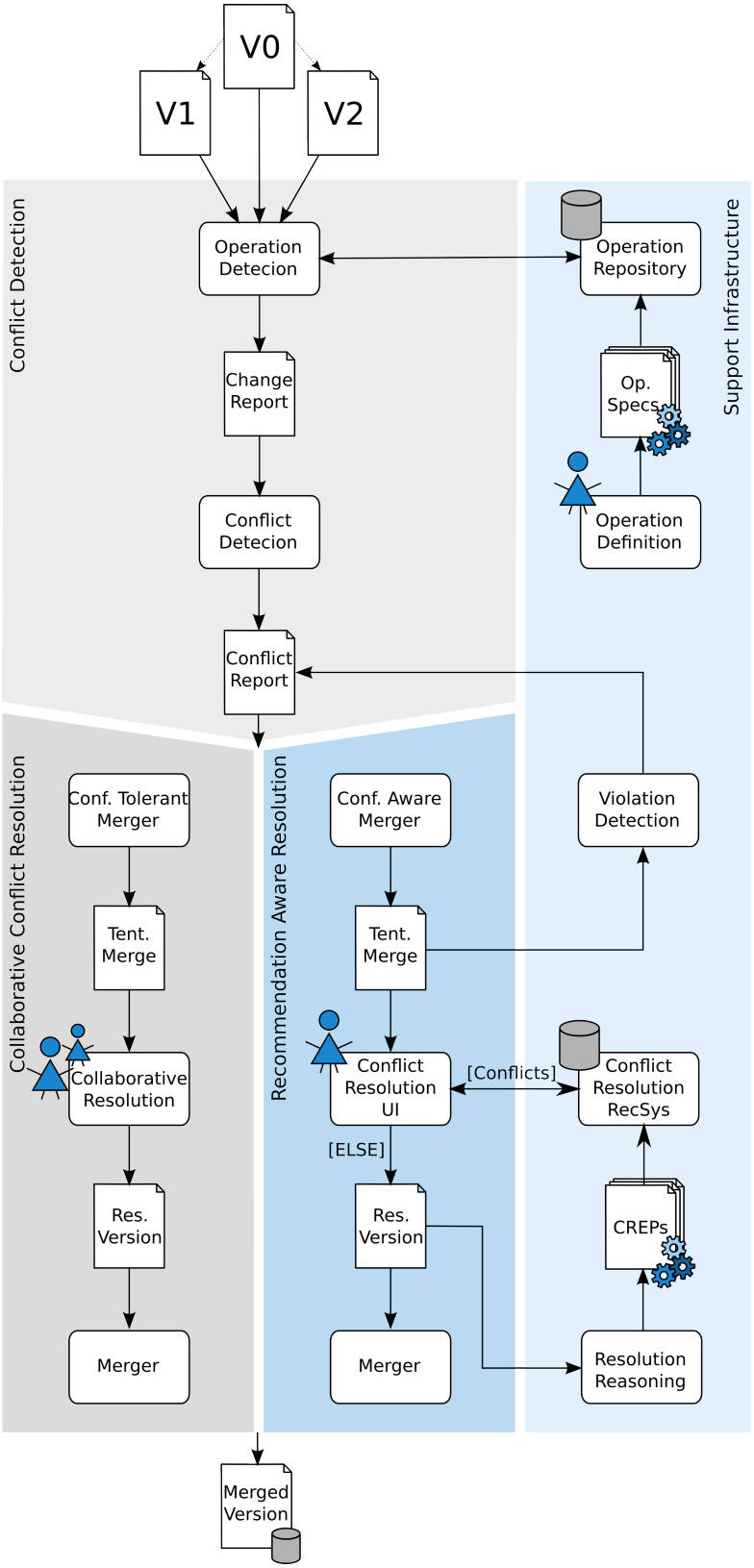


Figure 3.5: AMOR's Optimistic Model Versioning Process

3.2.1 The AMOR Workflow

Differently to the process in state-of-the-art versioning approaches, consisting of the phases *comparison*, *conflict detection*, *conflict resolution*, and *consolidation* (cf. Figure 2.1), the process implemented in AMOR is ready for (1) intention aware versioning, considering composite operations such as refactorings as first-class changes, and (2) consistency aware versioning, allowing to produce a tentatively merged version enabling the detection of violations. AMOR's merge phases are therefore *operation detection*, *conflict detection*, *tentative merge*, *violation detection*, and *resolution* (cf. Figure 3.5).

Operation Detection. AMOR provides out-of-the-box generic versioning capabilities based on the reflective interface of MOF implemented in the Eclipse Modeling Framework [60]. In this way, generic changes are detected in a state-based manner by comparing the modified versions V1 and V2 with the common ancestor version V0. However, the quality of the detection and resolution of conflicts may be considerably improved by incorporating language-specific knowledge. Thus, AMOR provides a dedicated operation detection phase, where language-specific knowledge is used to enhance generic change detection. Most importantly, information about composite operations is retrospectively inferred. The output of the operation detection is a change report combining atomic and composite changes performed to each modified version.

Conflict Detection. The change report serves as input for the conflict detection phase. In this phase, overlapping changes, i.e., contradicting changes and equivalent changes, are detected. As discussed in Section 3.1, overlapping changes concern atomic changes. However, overlapping changes may also regard an atomic change which is part of a composite change. Detected conflicts are summarized in a conflict report, which is handed over to a merge component.

Tentative Merge. The tentative merge phase calculates a merged model based on the conflict report. The tentative merge may be configured to operate on different merge strategies defining how to deal with conflicting changes. Strategies for excluding conflicting modifications, or privileging certain changes in the merged version are conceivable. In AMOR, a conflict tolerant merge strategy and a conflict aware merge strategy is implemented. The tentative merge forms the basis for the detection of violations. Further, it enables conflict resolution based on one integrated model.

Violation Detection. Violations are detected based on the tentatively merged model. AMOR provides an extension point to plug in external validation components to ensure consistency of the merged model. Currently, the validation of metamodel constraints and the validation of pre- and postcondition regarding composite operations are integrated. First steps towards incorporating a modeling language's semantics are realized in [29]. Information on detected violations are propagated back to the conflict report to collect all merge problems in a single artifact.

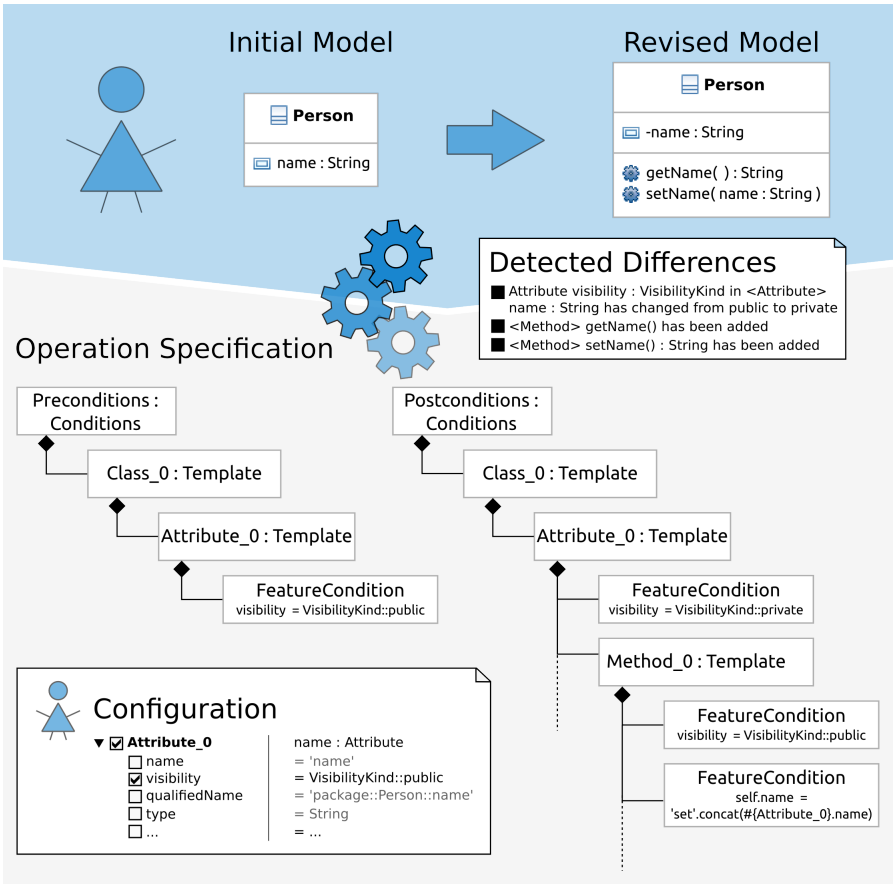


Figure 3.6: EMO: Enclose Variable Refactoring

Resolution. Conflict resolution is performed on the tentative merge considering information of the conflict report. Again, different strategies may be applied. AMOR offers two strategies: (1) conflicts may be tolerated for a while and are finally resolved collaboratively, or (2) conflicts are resolved by a single modeler, who is supported by a conflict resolution recommender system.

3.2.2 Components of AMOR

Several components and plugins are developed to realize AMOR's versioning workflow. In the following, we briefly discuss three main contributions of AMOR, each resulting in a PhD thesis.

EMF Modeling Operations. The key to AMOR's adaptability is the incorporation of language-specific composite changes, such as refactorings. Those composite changes improve conflict detection and the overall merge result. To enable modelers to specify such composite operations themselves without any programming effort, AMOR uses a by-demonstration approach implemented in EMF Modeling Operations (EMO). The specification process is depicted in Figure 3.6. The modeler provides an initial model and demonstrates the changes shaping the

composite operation. The two versions are compared and atomic differences are detected. In the example of Figure 3.6, the Enclose Variable refactoring is specified. The initial model comprises a class `Person` with a public attribute `name`, which is refined such that the visibility of the attribute `name` is set to private and corresponding public accessor methods, i.e., `getName` and `setName`, are added. EMO automatically derives a template for each detected difference disregarding the concrete names of the demonstrated model elements. Those templates form the pre- and postconditions of the so called *Operation Specification* model and may be configured manually. In the configuration step, dependencies between templates may be refined, e.g., the name of the accessor methods should concat the prefixes "get" and "set" with the name of the given attribute. Further, templates may be specified as iteration, e.g., to enclose all public variables of the given class. The configured operation specification is then persisted in an operation repository and (1) may be used to apply the demonstrated composite operation to arbitrary models ensuring the precondition, and (2) is used in the operation detection phase of the versioning workflow to a posteriori detect the application thereof. For more information about EMO and the resulting advancements regarding AMOR's versioning process, we kindly refer to [145].

Collaborative Conflict Resolution. AMOR allows to employ different strategies to handle conflicts. One strategy is to not consider conflicts harmful, but as potential of further discussion. Conflict resolution is not performed immediately by a single model after check-in as usual, but is postponed and accomplished in a synchronous collaborative setting, like a face-to-face workshop or chat. Instead of freezing versioning support until the collaborative merge is performed, a conflict tolerant merge strategy is applied. Here, information on conflicts is integrated into the merged model allowing to effectively reproduce the evolution of the model emerging in several check-in cycles. This information is analyzed to resolve the conflicts. As the resolution is performed collaboratively and synchronously, (1) conflicts resulting from inaccurate conflict resolution are avoided, and (2) the merged version is accepted by all parties. Collaborative conflict resolution is elaborated in [223].

Conflict Resolution Recommender System. A parallel strategy to collaborative conflict resolution is the use of the conflict resolution recommender system. Instead of resolving conflicts in a team, accepted conflict resolution patterns for recurring conflict situations are suggested and selected by the user. The conflict resolution recommender system directly operates on the model's concrete syntax to retain the user's mental map. Conflict resolution patterns go beyond prioritizing one of the conflicting changes and may provide a completely new solution. Thus, conflict resolution patterns are applied interactively, as possible new values, e.g., names for introduces model elements, are provided by the user. If no suggestion is available, a new conflict resolution pattern may be stored in the repository and used in similar future situations. More details on merging the model's concrete syntax and the conflict resolution recommender system are given in Chapter 5 and Chapter 6, respectively.

Overall, AMOR provides a generic versioning infrastructure supporting ECORE based models and allows for flexible configuration of its workflow resulting in precise conflict detection and supportive conflict resolution—either collaboratively or supported by recommendations.

CHAPTER 4

Model Transformation

In modern software engineering practice employing the MDE paradigm, models are the mean of choice to represent knowledge. The core idea of MDE is to utilize models to gain abstraction of the technical realization of a system, such that developers may completely concentrate on a specific matter of the problem domain. The resulting models are heavily interrelated and finally serve as construction plan for implementation. To free developers from the burden of repetitive and error-prone tasks such as translating models into source code and propagating changes throughout dependent models, a mechanism to transform and synchronize models is demanded. The field of *model transformation* accepts to play this central role and is thus noticed as the heart and soul of MDE [203]. Reflecting the plethora of application areas of models, model transformation tasks cover all sorts of activities. To support these activities, a multitude of either general or specifically tailored model transformation languages and approaches emerged.

In this chapter, we survey current model transformation approaches and refresh the fundamental concepts of graph transformation—the pioneer of concise model transformation.

4.1 Concepts and Techniques

As models and their underlying, historically more prominent graph structure, are referred to visual languages, model or graph transformation systems, also called graph rewriting systems, are classified as visual programming languages [14]. The history of visual programming languages is as old as the history of computers with graphics displays. Early representatives provided basic icon rewriting functionalities by matching identical occurrences of a search pattern and replacing it with a copy of the replacement pattern. Both patterns are stated in terms of a model. This straightforward procedure advanced the technique of programming by-demonstration, as for example done in [205]. To date, thanks to the flexible type-system gained by metamodeling introduced in Chapter 1, graph rewriting systems provide much more abstraction and expressiveness, allowing to specify rewriting rules not necessarily identical with the search pattern. Additionally, for the use in MDE, where arbitrary modeling languages not priorly known are employed, graph rewriting systems are not tailored to specific modeling languages, but are open

to define rules for arbitrary modeling languages. This flexibility comes along with the drawback that the visual language for rewriting rules is restricted to the abstract syntax graph, which may become complex and hard to read. Hence, many state-of-the-art model transformation languages either use a textual concrete syntax or aim at reinventing the by-demonstration approach by deducing a transformation rule from an example. However, the employed concrete syntax of the transformation approach is just one discriminator. Addressing the various scenarios for model transformation from slightly changing models in-place over converting models to different modeling languages to consistently synchronizing related models, different model transformation languages emerged, varying in their way of expressing and executing transformation rules [52, 160].

In the following, we briefly present distinguishing characteristics of those categories and survey their outstanding state-of-the-art representatives. However, the tools may not only be adopted to one unique category. As shown in [69], exogenous model transformations may be also achieved with tools primarily built for endogenous model transformations. Conversely, tools for exogenous model transformations may perform endogenous model transformations by defining transformations with equal input metamodel and output metamodel. Even bidirectional model transformations may be achieved by defining one transformation for each direction. Although this interchange is possible, defining transformations is much easier and safer with the right tool.

4.1.1 Endogenous Model Transformation

Endogenous model transformations describe transformations where source and target model conform to the same metamodel. In case that the source and target models are one and the same artifact, this kind of transformation is also called *in-place transformation*. If a new target model based on the source model's properties is created, the transformation is called *out-place*, even if both models conform to the same metamodel [160]. In the following, we introduce some outstanding approaches of this category.

EMF API. The *Eclipse Modeling Framework* (EMF) [61] provides a holistic platform for supporting and utilizing models. Models may be specified by means of EMF's meta-metamodel Ecore, a reference implementation of OMG's EMOF standard [111]. Additionally, EMF provides code-generation facilities realizing runtime support enabling the programmatic, API-based editing of models. Adjacent frameworks for visualization, validation, and other processing of models may interact with the generated code based on the integrated adapter mechanism.

EMF is no pure transformation approach, but allows for the direct manipulation of models. Because of its flexible and comprehensive API, EMF is often adopted as execution framework of other transformation languages.

AGG. The general purpose tool environment *AGG* supports the definition and execution of *typed attributed graph grammars* [87]. Such graph grammars consist of an optional type graph, a start graph called host graph, and a set of transformation rules, which may be attributed by Java objects. Rules are defined declaratively in terms of left-hand side (LHS) and right-hand side

(RHS) patterns, supplemented by a mapping between both sides. Restrictions may be stated as negative application conditions (NACs). The application order of matching rules is determined non-deterministically. However, AGG allows to control the order of rule application by means of layers, i.e., all rules of a lower layer have to be applied as long as possible, before rules of a higher layer may be executed. AGG is built on top of the formal foundations of the algebraic approach to graph transformation (cf. Section 4.2). Thus, it comes with comprehensive analysis techniques, such as consistency checking, critical pair analysis, dependency analysis, and the validation of termination criteria [212].

More recently, the formal foundations of AGG are integrated into the Eclipse incubation project *Henshin* [64], which was formerly known as EMF Tiger [18], to consistently transform EMF models [9, 23].

EMF Modeling Operations. *EMF Modeling Operations* (EMO) is an Eclipse based tool to define in-place model transformations for EMF models by-demonstration [36]. Such a model transformation is given by two concrete example models, which may be created using arbitrary EMF modeling editors including graphical editors. The *initial model* describes the situation before applying the transformation, i.e., the precondition, and the *revised model* states the transformation's result, i.e., the postcondition. Those examples are then compared and the actual transformation is derived automatically. Although the transformation may be manually configured, the transformation language and execution semantics are hidden from the user. More details may be found in Subsection 3.2.2 and [145].

4.1.2 Exogenous Model Transformation

An exogenous model transformation denotes a model transformation between models conforming to different metamodels, i.e., it takes one or more models of any modeling language as input and generates one or more new models of another modeling language. The first need for such kind of model transformation was observed in the area of domain-specific environment engineering [1, 148]. As stated in [1], existing graph transformation approaches as implemented in, e.g., AGG [87] and PROGRES [167, 199] are useful for formally specifying the behavior of model interpreters, but either lack built-in abilities of model to model transformation, or are built on top of proprietary metalanguages.

GReAT. The crucial factor for the development of the *Graph Rewriting and Transformation* (GReAT) language is the need for an efficient and automated approach for creating model interpreters within a comprehensive domain-driven software development framework [2]. The pattern specification language for GReAT's model to model transformation approach is derived by (1) unifying the input and output metamodels, and (2) adding references between the two metamodels [2, 12]. Inspired by string rewriting, GReAT's pattern specification language allows for the definition of a kind of regular expression in terms of vertex cardinalities. Setting a cardinality to zero effectively yields a negative application condition. Instead of stating search patterns and replacement patterns, GReAT uses a composite pattern, where each vertex and edge is annotated with one explicit action of *bind*, *new*, or *delete*. OCL expressions may act as guard

to determine whether an action should be executed for the bound subgraph. New or updated attribute values may be computed or set to given user inputs. To control rule execution, the following control flow concepts are introduced: Rules may be sequenced by passing bindings of a predecessor rule to a successor rule by so called input and output ports. If a binding is passed to more than one successor rules, non-determinism is introduced. Rules may be composed into hierarchically blocks, where a rule's output port may recursively be connected to an input port of a container rule. Finally, conditional execution is supported by test/case blocks. GReAT's transformation rules are then interpreted, debugged, or translated to C++ code.

ATL. The *ATLAS Transformation Language*, ATL for short, is a rule based, hybrid transformation language with textual concrete syntax [131]. A model transformation called *module* takes a read-only source model as input and generates a write-only target model. A module consists of *helpers* and *transformation rules*. While helpers can be used for accessing, traversing, and calculating source model elements only, transformation rules may produce target model elements based on occurrences in the source model. ATL automatically populates a transient trace model between source and target elements, which may be queried within the transformation. As a hybrid language, ATL offers a declarative part and an imperative part. Declarative rules, denoted by *matched rules* define a source and a target pattern with optional guards. Besides standard rules, which are automatically applied once per match in the given source model, matched rules may be explicitly triggered. Such rules, named *lazy rules*, may be called several times and produce the specified target pattern each time. *Unique lazy rules* in turn, produce the target elements only once and reuse them in later calls. To manually define the control flow, matched rules may be complemented by an optional action block, where a sequence of imperative statements may be specified and imperative *called rules* may be triggered.

Even though ATL is primary designed for exogenous model transformations, endogenous model transformations are supported using the *refining mode*. However, as source models are read-only, a new target model has to be generated. In refining mode, only changing parts of the model need to be explicitly transformed, unmatched parts are copied automatically.

4.1.3 Bidirectional Model Transformation

Bidirectional model transformation provides a consistent and synchronous framework for exogenous model transformation, i.e., model to model transformations may be executed in both directions based on the same definition. The first mentioned approach for bidirectional model transformations are triple graph grammars (TGGs) introduced by Schürr [197] as extension to Pratt's pair grammars [181]. The necessity for bidirectional model transformation came up in the field of CASE tools providing data integration functionality [148]. Incremental updating of related models is likely to become inconsistent when two unidirectional transformations are manually specified. Hence, an adequate specification formalism based on a single definition is required. The idea of a TGG is to represent relationships between two metamodels as explicit correspondence graph between two metamodels and to automatically translate a model transformation defined for one metamodel to the other metamodel. Thus, the resulting graph grammar consists of the triple: metamodel A, metamodel B, and their correspondence graph A-B. Based

on this definition, TGGs facilitate more than one specific transformation task; it enables the execution of transformations in both directions and incremental change propagation from one model to another model thereafter. Further, the same definition may be used to check whether two given models are equivalent and to establish a trace model in between.

Several CASE tools designated for round-trip engineering of related models or models and code are built on top of TGGs, e.g., PROGRES [199], Fujaba [42], and MOFLON [6].

TGG Interpreter. As an implementation close to the formal TGG approach proposed by Schürr [197], *TGG Interpreter* [174] combines the sound graph theoretic foundations of bidirectional model transformation with Eclipse's modeling environment. TGG Interpreter supports the graphical definition and execution of TGG Rules. Those rules are specified within a *Triple Graph Grammar* and orchestrate correspondences between two or more domains. Besides rule inheritance, guards, and global constraints, TGG Interpreter offers advanced concepts, such as *reusable nodes* [138]. Reusable nodes enforce the reuse of already existing elements instead of creating new ones. This technique not only allows the elegant specification of complex forward transformation scenarios, it also enables *intelligent* model synchronization. This way, TGG Interpreter prevents information loss in incremental model synchronization, as the number of destroyed and re-created model elements is minimized.

QVT. *Query/View/Transformation* (QVT) [110] is OMG's standard for model transformation. QVT is a hybrid approach with both, declarative and imperative nature. It is split into three languages, namely QVT Core providing core concepts for declarative model transformation, QVT Relations, as user-friendly relations language on top of QVT Core, and QVT Operational as imperative complement to QVT Core and QVT Relations or as pure operational transformation language. Though QVT Relations and QVT Core embody same semantics, the relations language offers a higher level of abstraction. One major advantage of QVT Relations over QVT Core is, that the trace model is managed automatically. Further, in addition to the textual concrete syntax, which is defined for all parts of QVT, QVT Relations offers a graphical syntax. In QVT Relations, a transformation between two or more domains is specified as a set of relations. Those relations may be constrained by *when* and *where* predicates to specify under which conditions the relationship needs to hold, and which conditions must be satisfied by all elements participating in the relation, respectively. QVT Operational may be invoked from QVT Core and QVT Relations and is further divided into two parts. *Operational Mappings* is the standard language operating on the same trace models as the relations language. *Black-Box Operations* are defined outside the transformation and plugged in via MOF Operation bindings. In case of black-box operations, traces must be kept explicitly. While the declarative part of QVT allows bidirectional model transformation, for the operational mappings and black-box operations, additional inverse implementations must be provided. There are two modes for running QVT transformations: *check-only* mode is used to check whether the relations hold; changes are only propagated to the given direction in *enforce* mode.

The OMG provides only the specification of QVT. Implementations partly fulfilling the standard may be found in [65, 66, 124, 214]. Further, Greenyer and Kindler [104] proposed an align-

ment of QVT Core to TGG, such that TGG Interpreter may be used as execution engine for QVT.

4.1.4 Higher-Order Transformation

Following the relational nature of TGGs, Bézivin et al. [21] proposed to express model transformations in terms of transformation models. Having the transformation specification explicit as MOF compliant transformation model enables a special case in the field of model transformation: *Higher-Order Transformation* (HOT), i.e., model transformations which transform model transformations. HOTs are characterized such that their input and output are model transformations. Application areas for HOTs are transformation synthesis, analysis, (de)composition, and modification [216].

4.1.5 Common Characteristics

We may conclude that common characteristics of model transformation languages are the possibility to state rewriting patterns as pre- and postconditions to ensure the application of transformation rules in certain situations only [210]. Similarly, *graph transformation*, first introduced in the late 1960s as an extension to term rewriting of Chomsky grammars [44] for handling non-linear data structures, describe graph rewriting rules in a declarative manner. Graph rewriting rules are composed by a *left-hand side* pattern acting as precondition and a *right-hand side* pattern acting as postcondition [27, 190]. While early representatives of graph transformation systems from the field of visual programming where application centered or specified to certain domains [55, 209], formal foundations for general purpose graph transformations were approached soon [83, 166, 179]. As models are often treated as graphs and graph transformation theory comes along with a set of formal, mathematically founded techniques and concepts to manipulate graphs, model transformations are soon widely expressed in terms of graph transformations [72, 84, 117]. Alternatively, concepts of several model transformation languages are mapped to graph transformation theory, to provide a formal rewriting semantics [97, 104, 218]. The advantage of graph transformation is besides its conciseness, that the rewriting logic is not hidden in any transformation engine's source code, but explicitly available in terms of given formalisms. We therefore employ graph transformation as the transformation language of choice in the remainder of this thesis. The following section provides some prerequisites for a common understanding of the formal foundations.

4.2 Graph Transformation

Like a model, a graph consists of nodes and edges connecting these nodes. This structure allows for the wide application of graphs to describe various problem domains in an intuitive manner. To bridge (meta)modeling and graph theory, graphs may be regarded as attributed graphs with inheritance, composition, and multiplicities [76–78]. For example, the mapping of the Ecore metamodel to graph theory is profoundly discussed in [23, 96]. Hence, from a technical viewpoint, models *are* graphs and both terms may be used interchangeably. To honor the long history of graph transformation, we use the term graph in the following. While graphs represent static

snapshots of a situation, graph transformations describe their evolution and thus, bring dynamic behavior to graphs [73]. Graphs are often visualized as diagrams by two-dimensional shapes and arcs to represent nodes and edges, respectively. As graph transformation rules are described in terms of graphs and therefore are visualized as diagrams, graph transformations are considered as visual programming languages, more precisely as diagrammatic rule-based visual programming languages [14]. Within the last 40 years, different graph transformation approaches have been proposed. Main representatives are the algorithmic, set-theoretic approach [58, 86], the algebraic, category-theory based approach [51, 81, 83] implemented in AGG [87], and the programmed graph replacement approach [198] implemented in PROGRES [199]. These approaches provide similar functionality, but differ in detail, i.e., their theoretical foundations.

In the remainder of this section, we focus on algebraic approaches only. Hence, we first establish a basic understanding of graph transformation with an informal, example-driven explanation of the theory and continue with an overview on the formal foundations of algebraic graph transformation.

4.2.1 Algebraic Graph Transformation Theory by Example

As a starting point, we discuss the basic concepts of graph transformation with the help of a small example describing a *Pull-Up Field* refactoring depicted in Figure 4.1. According to [117], graph transformation theory is based on three basic ideas.

1. *Conceptual generalization.* The first basic idea is to extract general concepts (types) from the problem domain. In analogy to metamodels and models, *type graphs* define the concept level and *instance graphs* represent individual snapshots conforming to a type graph. In the example of Figure 4.1, the UML Class Diagrams on the left of the figure are considered as snapshots, represented by instance graphs depicted on the right. The type graph is given by the UML Superstructure. Consequently, the instance graphs are equal to the abstract syntax of UML Diagrams.

2. *Behavioral generalization.* The second basic idea is to extract rules describing the general behavior observed in reality. Rules describe the scope of the transformation and the changes from one snapshot to the next. For example, the rules displayed in Figure 4.2 describe the changes from Snapshot G_0 to G_1 of Figure 4.1, i.e., the shift of the common attribute of the subclasses to the superclass. Details of the rules are discussed below.

3. *Graph based representation.* Finally, the third basic idea is to represent each artifact, i.e., concepts, snapshots, and rules, by means of graphs. Hence, a rule's graphs conform to the same type graph as the instance graphs to transform.

A rule is applied in three steps: (1) *Find* an occurrence of the LHS's graph in the host graph. (2) *Delete* all nodes and edges of the LHS's graph from the host graph which are not present in the RHS's graph. (3) *Add* all nodes and edges of the RHS's graph which are not existent in the LHS's graph.

In case of our example, two occurrences of rule 1's LHS's graph (cf. Figure 4.2) are found in G_0 (cf. Figure 4.1) if the input value of the variable `propName` is `artist`. One subgraph

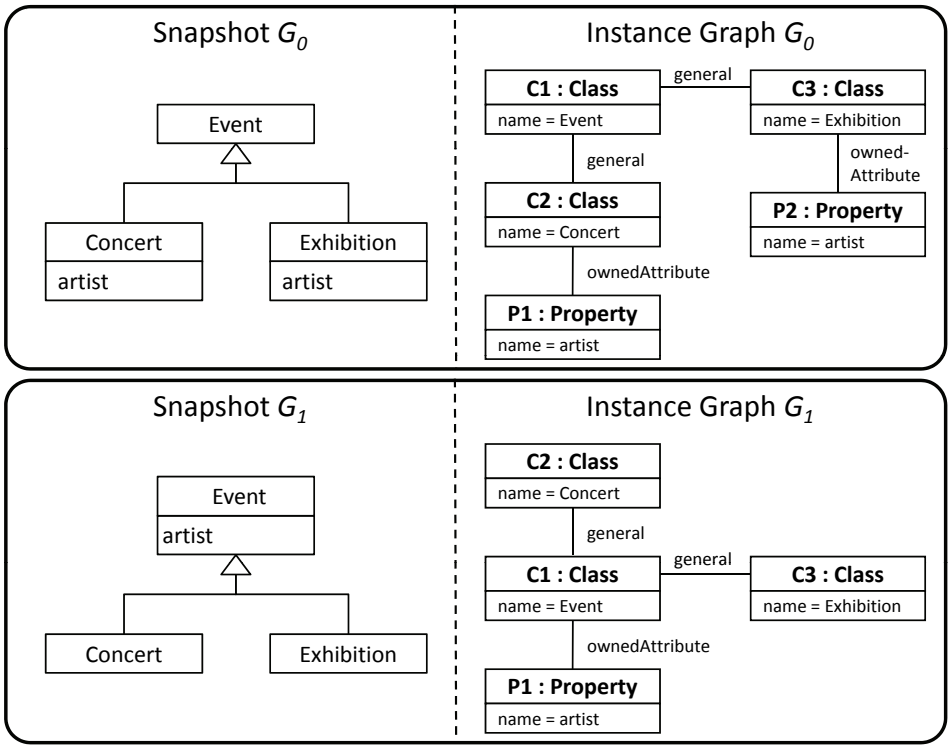


Figure 4.1: Pull-Up Field Example

is found by mapping class `0` of rule 1 to class `C1` of G_0, class `1` to class `C2`, property `2` to property `P1` as well as the references in between. This match is visualized in Figure 4.3. Dotted lines indicate the matched nodes. Matched edges are not indicated, for clarity reasons. The second match is based on the mapping of class `0` to class `C1`, class `1` to class `C3`, property `2` to property `P2` and respective references. The decision to which occurrence of the LHS's graph the rule is applied is non-deterministic. A second kind of non-determinism is due to the possibility of finding subgraphs for LHSs of multiple rules. In the example, the LHS of rule 2 is also a subgraph of G_0. To control the order of rule execution, additional nodes and edges may be introduced as control flow structures in the RHS's graph of early executed rules, which are used to match the LHS's graph of later rules and are eventually deleted. As the manual management of those control structures is error-prone, *programmed graph replacement systems* provide basic control flow expressions and allow to program with rules [198]. Algebraic approaches allow to define sequences by rule layers as done in the example; rules of layer 0 are applied first as long as possible, followed by rules of layer 1, and so forth [87]. After matching a subgraph, nodes and edges of the LHS which are not part of the RHS are determined. Now, the labels `0`, `1`, etc. of the LHS's and RHS's graphs come into play, as they indicate equal nodes and edges of both sides of a rule which are preserved. The unlabeled reference `ownedAttribute` of rule 1 (indicated red in Figure 4.3) has no counterpart in the RHS and is therefore removed from the instance graph. Finally, nodes and edges of the RHS which are not covered by the LHS are

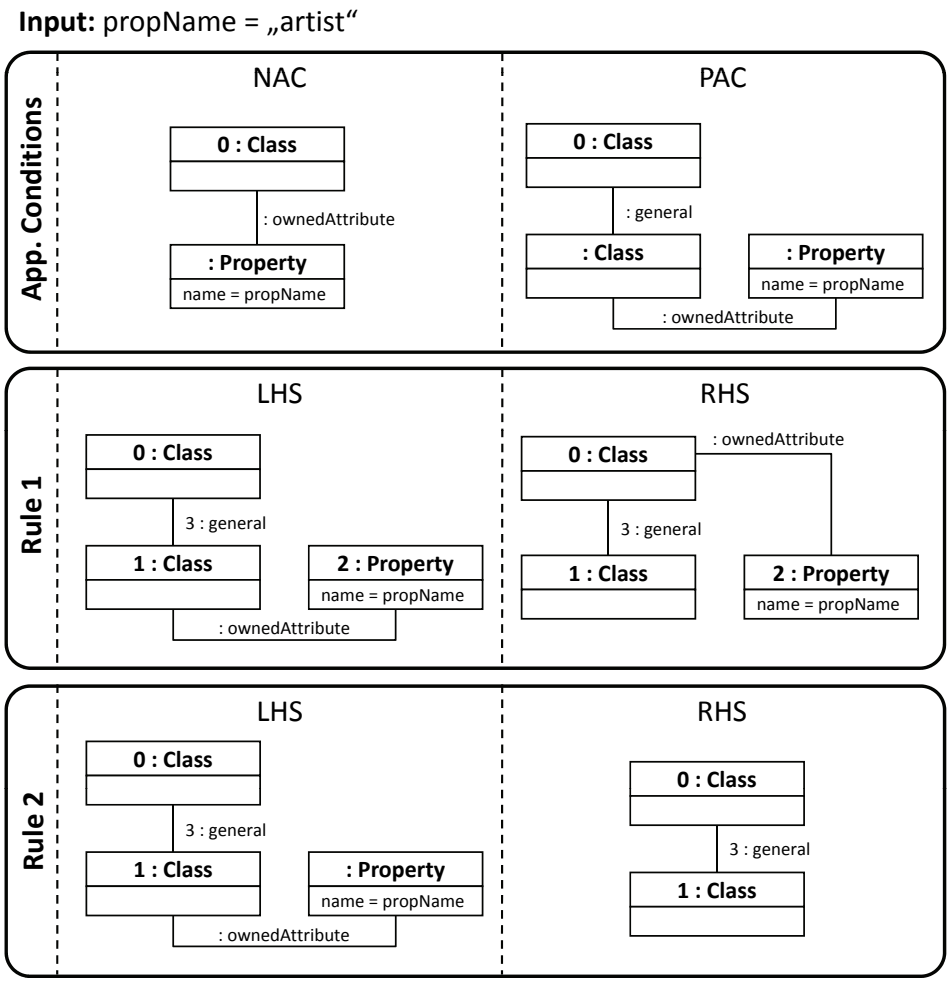


Figure 4.2: Pull-Up Field Graph Transformation

added to the instance graph. In our case, the reference `ownedAttribute` (marked green) is added to link class `0` and property `2`. Roughly spoken, rule 1 links a property named equal to the value of the variable `propname` from a subclass to a superclass. Rules are executed as long as their LHS is matched. Thus, rule 1 may be applied to G_0 a second time. As the superclass shall obtain the pulled-up property only once, additional application constraints are necessary. A *negative application condiction* (NAC) defines a pattern, which is not allowed in the instance graph. Contrary, a *positive application condition* (PAC) defines additional preconditions for rule application. The NAC in the example forbids the execution of rule 1, if class `0` matched by the LHS has already a link to a property with the given name, what is the case for G'_0. Further, as it is desired to apply the pull-up field refactoring only if the property is common to *all* subclasses, the PAC ensures that all subclasses of class `0` contain the property. With the application conditions at hand, rule 1 cannot be executed a second time and rule 2 is tested. A subgraph of LHS is found, and the rule is applied, which removes the property from the subclass. As no further

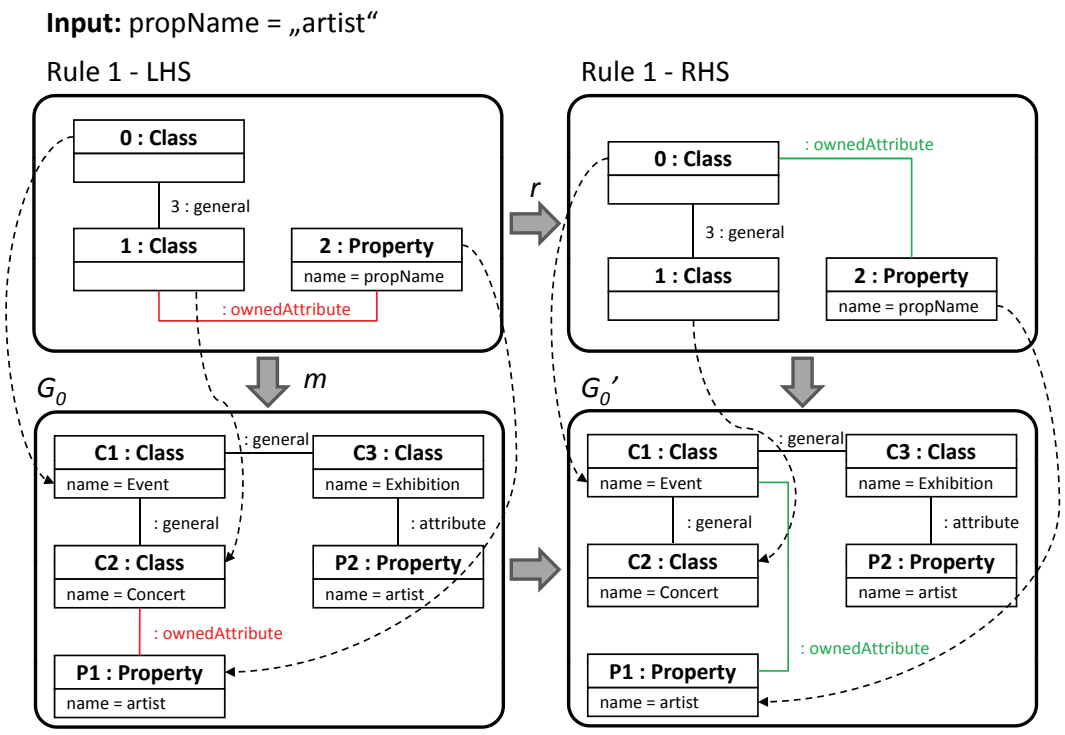


Figure 4.3: Application of the Pull-Up Field Transformation

match for the LHS is found, the transformation is finished.

Equipped with this exemplary overview on graph transformation, we may now go into details and discuss underlying formal concepts and definitions of the algebraic approach.

4.2.2 Foundations of Algebraic Graph Transformations

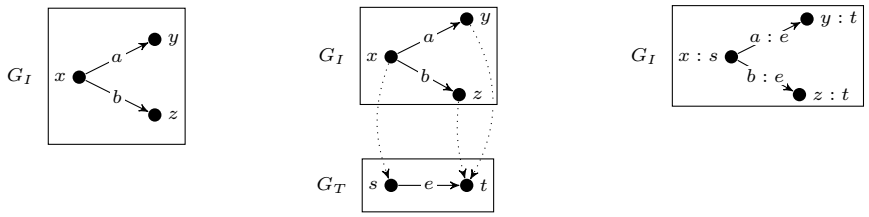
The algebraic approach to graph transformation is called *algebraic*, as it is characterized by the use of a special kind of algebra, i.e., *category theory* to describe graph structures and their relationships [51].

Introduction to Categories

A category is an algebraic structure comprising objects and morphisms with a composition operation on the morphisms and an identity morphism for each object [70, 155].

Definition 4.2.1: (category). *A category* $\boldsymbol{C} = (Ob_C, Mor_C, \circ, id)$ *is defined by a set* Ob_C*, called a* class *of objects satisfying the following conditions [70]:*

- Morphism. *For each pair of objects* $A, B \in Ob_C$*, there is a structure preserving mapping called* morphism $Mor_C(A, B)$ *written* $f : A \to B$*, if* f *is a morphism from* A *to* B*.* A *is called the* domain *of the morphism* f *and* B *the* codomain.



(a) Graph (b) Typed Graph

Figure 4.4: Examples for Graphs

- Composition. *For all triples* $A, B, C \in Ob_C$*, there exists a composition operation* $\circ_{(A,B,C)} : Mor_C(A, B) \times Mor_C(B, C) \to Mor_C(A, C)$.
- Identity. *For each object* $A \in Ob_C$*, there exists an identity morphism* $id_A \in Mor_C(A, A)$.

Further, for morphisms, composition, and identity, the following two axioms must be satisfied:

1. Associativity. *If* $A, B, C, D \in Ob_C$ *and morphisms* $f : A \to B$, $g : B \to C$, *and* $h : C \to D$, *then* $h \circ (g \circ f) = (h \circ g) \circ f$.
2. Identity. *If* $A, B \in Ob_C$ *and* $f : A \to B$, *then* $(id_A \circ f) = f$ *and* $(f \circ id_B) = f$.

In the following, we consider the *category of graphs and graph morphisms* and briefly summarize fundamental concepts of algebraic graph transformation approaches. For detailed formal foundations of algebraic graph transformation theory, please confer [27, 79, 190].

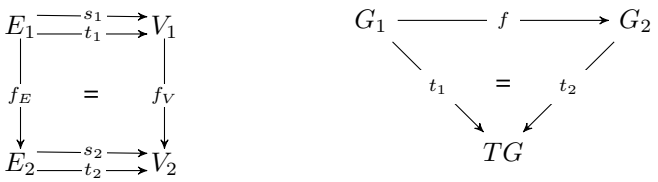
Category of Graphs and Graph Morphisms

Graphs are a central concept in graph transformation—the rule-based rewriting of graphs—and form the objects in the category of graphs and graph morphisms. Hence, we start our introduction to graph transformation with a definition of the basic notions *graph* and *graph morphism*.

Definition 4.2.2: (graph). *A* graph $G = (V, E, s, t)$ *is defined as a set of vertices called* nodes V *and a set of* edges E. *Edges are directed and related with a* source *node and a* target *node, which are assigned by the source function* $s : E \to V$ *and the target function* $t : E \to V$ *[75]:*

$$E \underset{t}{\overset{s}{\rightrightarrows}} V$$

Example 4.2.3: graph Consider a graph $G_I = (V_I, E_I, s_I, t_I)$ with node set $V_I = \{x, y, z\}$, edge set $E_I = \{a, b\}$, source function $s_I : E_I \to V_I : a, b \mapsto x$, and target function $t_I : E_I \to V_I : a \mapsto y; b \mapsto z$. G_I as visualized in Figure 4.4a.



(a) Graph Morphism (b) Typed Graph Morphism

Figure 4.5: Commutative Diagrams for Morphisms

Definition 4.2.4: (graph morphism, total graph morphism). *Given two graphs G_1, G_2 with $G_i = (V_i, E_i, s_i, t_i)$ for $i = \{1, 2\}$, a graph morphism is a mapping $f : G_1 \to G_2$, $f = (f_V, f_E)$ consisting of two functions $f_V : V_1 \to V_2$ and $f_E : E_1 \to E_2$. A graph morphism is* total*, if f_V and f_E are total structure preserving mappings, i.e., the source and target functions are preserved for each $v \in V_1$ and for each $e \in E_1$, such that $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$ is satisfied, as shown in the diagram of Figure 4.5a [51, 75].*

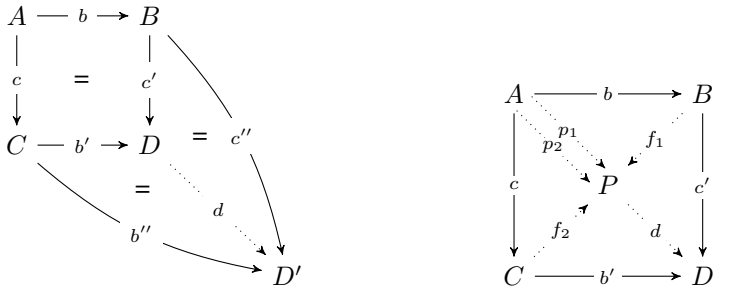
Definition 4.2.5: (partial graph morphism). *A partial graph morphism $f : G_1 \to G_2, f = (f_V, f_E)$ is a total graph morphism as given in Definition 4.2.4 from some subgraph $dom(f)$ of G_1 to G_2. $dom(f)$ is denoted as the* domain *of f [81].*

Remark 4.2.6: Given two graphs G_1 and G_2 with $G_i = (V_i, E_i, s_i, t_i)$ for $i = \{1, 2\}$ with node sets $V_1 = \{x\}$, $V_2 = \{y\}$ and a graph morphism $g : G_1 \to G_2 : x \mapsto y$, then x is denoted as *preimage* and y is called *image*.

Graphs may be typed over a *type graph*. A type graph is a distinguished graph, which defines a set of types, used to type nodes and edges of graphs.

Definition 4.2.7: (type graph, typed graph, and typed graph morphism). *A* type graph *is a distinguished graph $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$, with* vertex type alphabet *V_{TG} and* edge type alphabet *E_{TG}. Given a graph G together with a graph morphism $t : G \to TG$, the pair (G, t) is called a* typed graph *or instance graph of TG, denoted by G^T. A* typed graph morphism *$f : G_1^T \to G_2^T$, where $G_1^T = (G_1, t_1)$ and $G_2^T = (G_2, t_2)$ are typed graphs with type mapping $t_i : G_i \to TG$ for $i = \{1, 2\}$, is a graph morphism, such that $t_2 \circ f = t_1$, as depicted in Figure 4.5b [75].*

Example 4.2.8: typed graph Consider the type graph $G_T = (V_T, E_T, s_T, t_t)$ with $V_T = \{s, t\}$, $E_T = \{e\}$, $s_T : E_T \to V_T : e \mapsto s$, and $t_T : E_T \to V_T : e \mapsto t$. The type graph defines two types of nodes: s-typed source nodes with only outgoing edges and t-typed sink nodes with only incoming edges. The graph G_I from Example 4.2.3 together with the type mapping morphism $t = (t_V, t_E) : G_I \to G_T$ with $t_V : V_I \to V_T : x \mapsto s; y, z \mapsto t$ and $t_E : E_I \to E_T : a, b \mapsto e$, is then a typed graph, called instance graph of G_T. Two notations for visualizing typed graphs exist, as shown in the diagram of Figure 4.4b. In the explicit notation depicted on the left-hand side, both graphs G_I and G_T are visualized with explicit type mapping morphism



(a) Pushout (b) Canonical Construction

Figure 4.6: Commutative Diagrams for Pushout

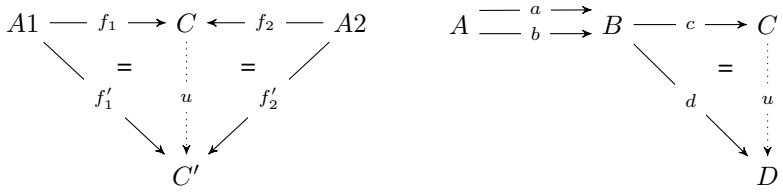
indicated by the dotted lines. In the compact notation of the right-hand side, the explicit visualization of the type graph is omitted and instead, type names are included in the instance graph's node and edge labels.

Remark 4.2.9: The definition of graphs and graph morphisms may be extended by arbitrary properties, e.g., label assignments to nodes and edges, or more sophisticated concepts like node and edge attributes, types, composition, and inheritance [77, 78]. The definition of *typed graphs* and *typed graph morphisms* is exemplarily shown in Definition 4.2.7. However, to get a general idea of algebraic graph transformation, the definition of untyped directed graphs is sufficient, as the genericity of the categorical approach renders all proofs and constructions valid for various types of graphs [162]. For the sake of simplicity, we therefore regard graphs as defined in Definition 4.2.2.

Graph Transformation

In order to evolve graphs, a mechanism is needed to change a given graph structure. This mechanism is described by graph transformation rules, called *productions*, specifying a rewriting step from a graph G to a derived graph H. A production $p : L \to R$ consists of two graphs, denoted as left-hand side (LHS) graph L and a right-hand side (RHS) graph R, with partial mapping between L and R, determining nodes and edges which are preserved by an application of p. Nodes and edges $\in L \backslash R$ are deleted by an application of p whereas nodes and edges $\in R \backslash L$ are added. Thus, the derived graph H is basically constructed by $G \backslash (L \backslash R) \cup (R \backslash L)$ [51].

A production is said to be applicable to a graph called *host graph*, if the LHS's graph forms a subgraph thereof. A *subgraph* of G in a host graph H, written as $G \subseteq H$, is given by the image G in H, called *match* of G in H. The match is identified by a graph morphism $g : G \to H$ and forms the basis of the algebraic construction of *gluing* two graphs, called *pushout*, i.e., merging two graphs with common subgraph. The application of a production p to a given host graph G with match $m : L \to G$ is called *direct derivation* or *direct graph transformation*, written as $G \overset{p,m}{\Longrightarrow} H$ [51, 74].



(a) Co-Product (b) Co-Equalizer

Figure 4.7: Commutative Diagrams for Co-Product and Co-Equalizer

Definition 4.2.10: (pushout). *Given two morphisms $b : A \to B$ and $c : A \to C$ in a category **C**, the triple $\langle D, b' : C \to D, c' : B \to D \rangle$ is a* pushout *over b and c, with pushout object D and morphisms $b' \circ c = c' \circ b$ such that the following universal property holds: For all objects D' and morphisms $b'' : C \to D'$ and $c'' : B \to D'$, with $b'' \circ c = c'' \circ b$, there exists a unique morphism $d : D \to D'$ with $d \circ b' = b''$ and $d \circ c' = c''$. The pushout object D is called the* gluing *of B and C via (A, b, c). Further, if $\langle D, b', c' \rangle$ is a pushout of b, c, then C is called the pushout complement object of $\langle b, c' \rangle$ defined by the triple $\langle C, b' : C \to D, c : A \to C \rangle$ [51,75]. The commutative diagram is given in Figure 4.6a.*

The universal property of pushouts ensures that if the pushout object exists, it is unique up to isomorphism. The pushout object D is canonically constructed via A, b, c by first building the disjoint union of B and C given by the *co-product* object, followed by identifying elements with equal preimage in A, i.e., constructing the *co-equalizer* yielding the pushout object D.

Definition 4.2.11: (co-product). *Let $A1$, $A2$ be a pair of objects in a category **C**. The* co-product *is a triple $\langle f_1, f_2, C \rangle$, where C is the co-product object, and f_1, f_2 are morphisms such that the following universal property holds: for all pairs of morphisms $\langle f_1' : A1 \to C', f_2' : A2 \to C' \rangle$, there exists a unique morphism $u : C \to C'$, where $u \circ f_1 = f_1'$ and $u \circ f_2 = f_2'$, such that the diagram in Figure 4.7a commutes [51, 195].*

In the category of sets and in the category of graphs, the co-product $\langle f_1, f_2, C \rangle$ of the objects $A1$ and $A2$ is the disjoint union $C = A1 \dot{\cup} A2$ together with morphisms f_1, f_2 denoted *injections* [71, 195].

Definition 4.2.12: (co-equalizer). *Let $a, b : A \to B$ be a pair of parallel morphisms in a category **C**. A* co-equalizer *of $\langle a, b \rangle$ is given by a pair $\langle C, c : B \to C \rangle$ such that the conditions depicted in Figure 4.7b hold: (1) $c \circ a = c \circ b$ and (2) for all objects D and morphisms $d : B \to D$, with $d \circ a = d \circ b$, there exists a unique morphism $u : C \to D$ such that $u \circ c = d$ [81, 195].*

The co-equalizer reduces two morphisms with identical preimage to their equal kernel and is thus considered as the smallest equivalence relation [81, 195].

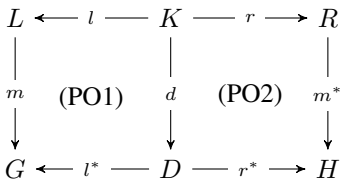


Figure 4.8: Commutative Diagram for a DPO Graph Transformation

Pushout Construction via co-product and co-equalizer. Given two morphisms $b : A \to B$, $c : A \to C$, the canonical construction of a pushout depicted in Figure 4.6b is as follows: (1) Construct the co-product $\langle f_1 : B \to P, f_2 : C \to P, P \rangle$ of the objects B and C. (2) Construct the co-equalizer $\langle d : P \to D, D \rangle$ of $p_1 := f_1 \circ b$, $p_2 := f_2 \circ c$. (3) Define the morphisms $c' := d \circ f_1$ and $b' := d \circ f_2$ [195].

In the following, we discuss the two main representative approaches of algebraic graph transformation based on pushouts, namely the double pushout approach and the single pushout approach.

Double Pushout Approach. Historically, algebraic graph transformation is first based on the *double pushout* approach (DPO) defined in the category of graphs and *total* morphisms denoted by $\boldsymbol{Graph}$ [51]. Thus, the partial mapping $p : L \to R$ is represented as span of total morphisms $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$, by introducing a gluing graph called *interface graph* K.

Definition 4.2.13: (graph production). *A* graph production $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ *is given by the left-hand side graph L, the gluing graph K, and the right-hand side graph R and two injective graph morphisms $l : K \to L$ and $r : K \to R$.*

The inverse production p^{-1} *of a production p is given by $p^{-1} = (R \overset{r}{\leftarrow} K \overset{l}{\rightarrow} L)$ [74].*

The DPO approach has its name based on the fact that the application of a production consists of two pushouts (PO1) and (PO2), as shown in Figure 4.8. The interface graph K, is a subgraph of L and R, satisfying $L \supseteq K \subseteq R$, and holds nodes and edges $\in L \cap R$, i.e., elements preserved by the application of the production. Remember the basic construction of the derived graph $H = G \backslash (L \backslash R) \cup (R \backslash L)$ stated above. The first pushout (PO1) describes the deletion of elements $\in L \backslash R$ as inverse gluing operation yielding the pushout complement object D, called *context graph*, which in turn acts as pushout complement object for the second pushout (PO2) handling the actual insertion of elements $\in R \backslash L$ in H.

Definition 4.2.14: (graph transformation). *A* direct graph transformation $G \overset{p,m}{\Longrightarrow} H$ *is given by the commutative diagram depicted in Figure 4.8, where (PO1) and (PO2) are pushouts in the category* $\boldsymbol{Graph}$*, where $p = (L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R)$ is a graph production, G the host graph, H the derived graph, and $m : L \to G$ a graph morphism denoted as match.*

A sequence of direct graph transformations $\rho = (G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n)$ is called graph

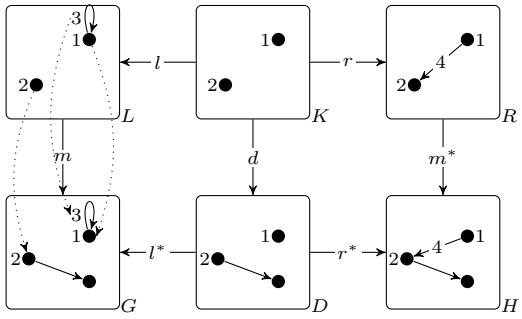


Figure 4.9: Example for a DPO Graph Transformation

transformation*, denoted by* $G_0 \overset{*}{\Rightarrow} G_n$*. For the special case* $n = 0$*, the direct graph transformation* $p_0 : G_0 \Rightarrow G_0$ *is equal or isomorphic to the identity morphism* $id : G_0 \Rightarrow G_0$*, as pushouts and consequently direct graph transformations are only unique up to isomorphism [74].*

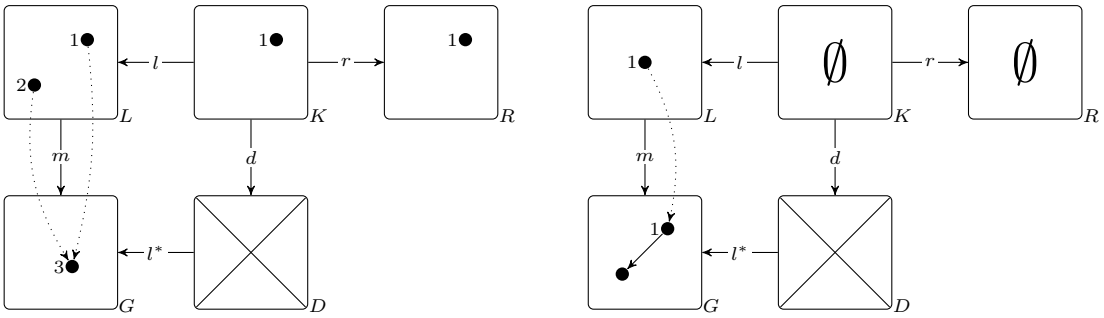
The application of a production, i.e., a direct graph transformation may be reversed via its inverse production and its co-match m^*, yielding an equal or isomorphic graph to the original host graph G.

Fact 4.2.15: (inverse graph transformation). *For a direct graph transformation* $G \overset{p,m}{\Longrightarrow} H$ *with given co-match* m^**, there exists a direct inverse graph transformation* $H \overset{p^{-1},m^*}{\Longrightarrow} G$ *[74].*

We may now refine the applicability of a production $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ to a graph G via match m such that there must exist a context graph D for a valid pushout (PO1) according to Definition 4.2.10. Such a context graph exists and is unique up to isomorphism, i.e., for the pushout object and all pushout complement objects the universal property is fulfilled, According to [51,74], this is only the case if and only if the match $m : L \rightarrow G$ satisfies the application condition called *gluing condition*. The gluing condition consists of two parts, namely the *identification condition* requiring a unique image of elements which should be deleted, and the *dangling condition* ensuring that the constructed graph has no dangling edges, i.e., in the DPO approach, a node can only be deleted, if all adjacent edges to that node are also deleted by the production.

Definition 4.2.16: (gluing condition). *A given production* $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$*, and a given match* $m : L \rightarrow G$ *with* $X = (V_X, E_X, s_X, t_X)$ *for all* $\mathrm{X} \in \{\mathrm{L}, \mathrm{K}, \mathrm{R}, \mathrm{G}\}$ *satisfy the* gluing condition *if the* dangling condition *and the* identification condition *are satisfied, where*

- *the dangling condition ensures that no edge* $e \in G_E - m_E(L)$ *is adjacent to any node in* $m_V(\ L_V - l_V(K_V))$*, i.e., no node is deleted which would cause dangling edges*
- *the identification condition requires that there is no element* $x, v \in L_V \cup L_E$ *such that* $x \neq y, m(x) = m(y)$ *and* $y \notin l(K_V \cup K_E)$*, i.e., deleted elements must have a unique preimage in L [51, 74].*



(a) Violation of the Identification Condition

(b) Violation of the Dangling Condition

Figure 4.10: Examples for Violations of the Gluing Conditions

Example 4.2.17: direct graph transformation in $Graph$ Consider the production $p : L \to R$ depicted in the diagram of Figure 4.9. The LHS L describes a patten of two nodes, one of which with self-referencing edge, which is rewritten in the RHS R such that the self-referencing edge is deleted and a new edge connecting both nodes is added. Before the production p can be applied to graph G with match m indicated by equal numbers in the diagram, we have first to check if the gluing condition is satisfied. The dangling condition is satisfied, besides due to that no node is deleted by the production, as no unmatched edge in G is source or target of a deleted node's image in G. Further, the identification condition is satisfied, as no deleted element in G has more than one preimage in L such that it is deleted and preserved at the same time. The gluing condition is satisfied and the interface graph K is constructed by $L \backslash R$. Then, the context graph D is built by performing the inverse gluing operation, i.e., the first pushout over l and m, where all elements in G with preimage in L but not in K are deleted. Elements added by the production $R \backslash L$ are finally embedded into D by the second pushout over r and d yielding the derived graph H.

Example 4.2.18: violation of the identification condition Given a production $p : L \to R$ with two nodes in L leaving only one node in R, a shown in Figure 4.10a. Applying p to G with match m, as depicted in the diagram below, is not allowed due to a violated identification condition: node 3 in G has two preimages in L, i.e., node 1 which is preserved, and node 2 which is deleted.

Example 4.2.19: violation of the dangling condition Consider the graph production of Figure 4.10b: a single node defined in L is deleted in R. The application of p to graph H with match m is not possible due to a violated dangling condition. The deletion of node 1 would leave behind a dangling edge.

Besides the DPO approach, algebraic graph transformation may be based on the single pushout approach (SPO). The main difference between the DPO and the later developed SPO is that in the more general SPO approach, the direct derivation is defined as one single pushout in the category of graphs and partial graph morphisms, resulting in a different treatise of the problematic situations addressed in Example 4.2.18 and Example 4.2.19.

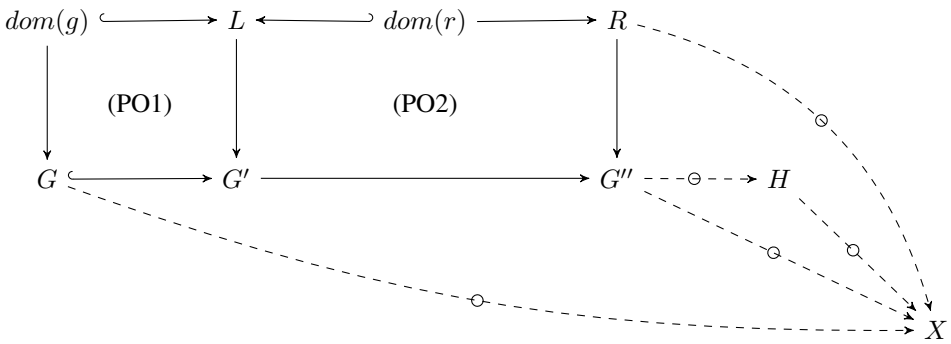


Figure 4.11: Commutative Diagram for a SPO Graph Transformation

Single Pushout Approach. The single pushout approach (SPO) is defined as a single pushout in the category of graphs and *partial* morphisms denoted by $\boldsymbol{Graph^P}$ [82, 151]. This way, the pushout construction is more general, as in the DPO approach, as it is not dependent on an interface graph. The pushout of two morphisms $L \to R$ and $L \to G$ in $\boldsymbol{Graph^P}$always exists [81], which enables the application of graph transformation rules in the SPO either in cases, where the gluing condition of the DPO does not hold. Hence, the main difference between the SPO and the DPO approach is the way how problematic situations are treated: While in the DPO approach, a production is not applicable if the gluing condition is not satisfied, in the SPO approach (1) nodes with adjacent edges to other nodes, not matched in the LHS, are deleted, and (2) deletion outpaces preservation for elements without unique preimage relation. As deletion has priority over preservation, which is due to the order of operations in the SPO construction, the graph structure may get destroyed. Then, in order to obtain a valid graph, dangling edges are removed. In cases where such side-effects are undesired, user-defined contextual application conditions to restrict *when* rules may be applied may be introduced [115]. These *positive* and *negative application conditions* specify the required existence and non-existence of certain nodes, edges, or subgraphs. Application conditions may be used to express gluing conditions, equivalent to the gluing conditions of the DPO approach. Thus, the DPO approach with built-in application condition may be considered as special case of the SPO approach. An exhaustive comparison of the SPO approach with the DPO approach is given in [81].

The SPO is given by one single pushout $\langle H, G \to G' \to G'' \to H, R \to G'' \to H \rangle$ of the partial morphisms $L \to R$ and $L \to G$ in $\boldsymbol{Graph^P}$. As depicted in the diagram of Figure 4.11, the SPO construction consists of three steps and is performed as follows.

PO 1. The first gluing step is obtained by the pushout construction $\langle G', L \to G', G \to G' \rangle$ of the total morphisms $dom(g) \to G$ and $dom(g) \to L$ in $\boldsymbol{Graph}$.

PO 2. The second gluing step is the pushout construction $\langle G'', R \to G'', G' \to G'' \rangle$ of the total morphisms $dom(r) \to L \to G'$ and $dom(r) \to R$ in $\boldsymbol{Graph}$.

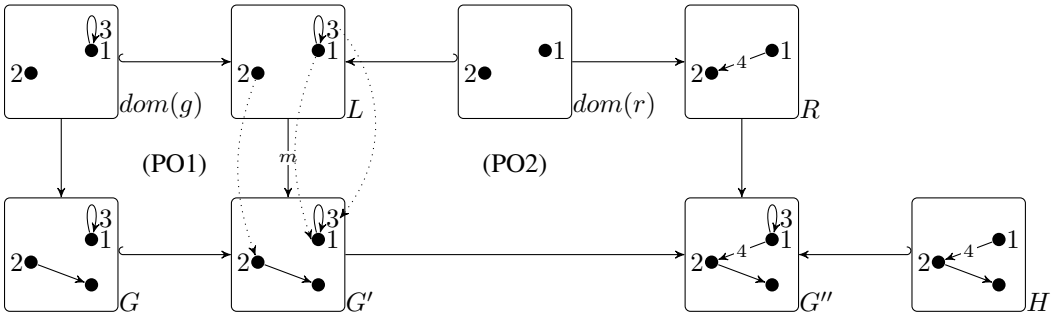


Figure 4.12: Example for a SPO Graph Transformation

Deletion. Finally, the co-equalizer $\langle H, G'' \to H \rangle$ of the partial morphisms $L \to R \to G''$ and $L \to G \to G' \to G''$ yields the derived Graph H.

Example 4.2.20: direct graph transformation in $Graph^P$ Consider the production $p : L \to R$ of Example 4.2.17. The application of the production at match m to graph G' leads to the direct derivation depicted in Figure 4.12. The first gluing step (PO1) is achieved by the pushout of the total morphisms $dom(g) \to L$ and $dom(g) \to G$. The second gluing step (PO2) yields the pushout object G'' via the total morphisms $dom(r) \to L \to G'$ and $dom(r) \to R$, where the new edge labeled 4 is added to graph G'. The deletion of edge 3 is finally performed by the co-equalizer $\langle H, G'' \to H \rangle$, as it has a preimage in L via G' but not via R.

4.3 Summary

In this chapter, we discussed different application scenarios for model transformations and introduced concepts and state-of-the-art techniques to handle this complex transaction. While endogenous model transformation describe transformations where source and target models adhere to the same metamodel, the domain of exogenous model transformation is the transformation across metamodels. A special case is bidirectional model transformation, where the same transformation definition is used to perform model transformations in either direction, for incremental change propagation, and to check for equivalence of two models. Further, whenever model transformations are expressed in terms of transformation models, higher-order transformations, i.e., transformations on top of the transformation models, may be defined.

All in all, model transformations are often expressed declaratively, in terms of pre- and post-condition patterns, similar as done in term rewriting, or graph transformation, which perfectly fit the model's graph based nature. The history of the field of graph transformation dates back to the late 1960s, what explains their well elaborated, mathematically founded theory, enabling static reasoning about the transformations.

We further highlighted the foundations of the algebraic, category-theory based approach as one major representative of graph transformation. The application of such a graph transformation production may be constructed in two ways: The double pushout approach comes

with built-in application condition ensuring valid output models only. The more general single pushout approach in turn is always applicable. However, the single pushout construction is more complex and may destroy the model's structure. Thus, we base further developments throughout this thesis on the more reliable double pushout approach.

CHAPTER 5

Conflict Aware Merge

Whenever artifacts are not locked and thus are subject to concurrent modifications as in optimistic versioning systems, isolated changes have to be integrated into one consolidated version from time to time. For the integration of those changes, several merge strategies exist, which differ in their level of automation and user friendliness. When it comes to conflicting changes, for sequential artifacts like text files, pointing the user to highlighted conflicting parts shown side by side works satisfactory well in practice. Applied to graph-based artifacts like models, this approach yields accidental complexity [28] to the user in charge of merging, especially when modelers are used to the model's concrete syntax only.

In this chapter, we first propose a generic strategy to automatically merge models in a conflict aware manner by treating conflicts as first class citizens. Secondly, we leverage the conflict aware merge strategy for the visualization of changes and conflicts. To this end, we compute *Conflict Diagrams* and employ UML profiles [113], the language inherent extension mechanism of UML. As a result, modelers may resolve conflicts in the graphical concrete syntax using their familiar UML editors without requiring tool extensions.

5.1 Related Work

Regarding our goal of computing conflict diagrams for evolving UML models managed by optimistic model versioning systems, we identify three threads of related work. First, we analyze strategies for integrating isolated changes in general. Second, we consider how changes and conflicts are detected and presented to the user in model versioning approaches. Third, we elaborate on approaches for ensuring the mental map [59], i.e., the modeler's personal view on the model across different diagram versions.

General Merge Strategies. In order to deal with conflicts when merging artifacts, several strategies are conceivable. Versioning systems for code like Subversion and CVS typically employ a manual merge strategy when it comes to integrating conflicting changes. Then, the two

parallel evolved versions of an artifact are shown to the user side by side, conflicting changes are highlighted. The user has to analyze the evolution of the artifact and to decide which changes shall be integrated into the merged version. For sequential artifacts like text files, merging works satisfactory well in practice. However, applied to the textual serialization of graph-based artifacts like models, this approach fails, as discussed in Chapter 2.

As manual conflict resolution is error-prone and cumbersome, it seems naturally, that avoiding conflicts is a preferable goal. Munson and Dewan present a flexible framework for merging arbitrary objects, which may be configured in terms of merge policies [164]. Merge policies may be tailored by users to their specific needs and include rules for conflict detection and rules for automatic conflict resolution. Actions for automatic conflict resolution are defined in merge matrices and incorporate the kinds of changes made to the object and the users who performed those changes. Thus, it may be configured, e.g., that changes of specific users always dominate changes of others, or that updates outpace deletions.

In contrast, nearly as long as collaborative systems exist, several works have been published, arguing that inconsistencies are not always a negative result of collaborative development. They propose to tolerate inconsistencies at least temporarily for several reasons [170]. Inconsistencies may identify areas of a system, where the developers' common understanding has broken down, and where further analysis is necessary. Another reason for tolerable inconsistencies arise when changes to the system are so large, that not all dependent changes can be performed at once. Further, fixing inconsistencies may be more expensive than their impact and risk costs. Tolerating inconsistencies requires the knowledge of their existence and careful management. Undetected inconsistencies in contrast, should be avoided as they cause problems. Schwanke and Kaiser [200] propose as one of the first an adapted programming environment for identifying, tracking, tolerating, and periodically resolving inconsistencies. Similarly, Balzer [13] allows to tolerate inconsistencies by relaxing consistency constraints and annotating inconsistent parts with so called pollution markers.

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

Merging in Model Versioning. While a profound discussion on model versioning systems and their features is presented in Chapter 2, in this section we focus on the merge phase, especially, how changes and conflicts are detected and presented to the user.

According to Mens [157], *state-based* approaches and *change-based* approaches may be distinguished. While state-based approaches, e.g., [40, 46, 123, 165] compute the changes between two model versions by matching and difference algorithms, change-based approaches, e.g., [140, 173, 194] record the changes directly in the modeling editor as they are applied.

Concerning the visualization of changes and merge conflicts, state-based approaches present the common ancestor model, the two revised models, the difference reports as well as the conflict report. In contrast, change-based approaches show the initial model, the two sets of changes from both modelers, respectively, and an optional conflict report stating dependencies between changes. Another possibility for change-based approaches is to show the initial model with

applied non-conflicting changes and only the conflicting changes are outstanding. However, the aforementioned approaches neither consider to represent only one integrated model and tolerate inconsistencies during the merge phase, nor enrich the integrated model with annotations for showing the changes and conflicts explicitly to the user. Only one approach allows to compute a single, conflict-free merged model. Ehrig et al. [80] produce a pre-merged version by giving updates a higher priority as deletions. By this, a first version of the merged model is generated and subsequently the user has to reason about if deletion should have actually a higher priority. Furthermore, all aforementioned approaches neglect the concrete syntax of models, thus, no attempts are made to show changes and conflicts in the concrete syntax of models.

Finally, dedicated approaches for visualizing differences between diagram versions have been proposed by Mehra et al. [156] and Ohst et al. [175] by using different coloring and highlighting techniques for changed model elements shown in so called *unified diagrams* incorporating all changes of both users. The implementation of the approach of Ohst et al. has been presented in [168], but solely considers two-way merges. Three-way merges are only mentioned as subject to future work. Thus, only update/update conflicts for attribute values and element moves are marked explicitly in the unified diagrams. Furthermore, tool-specific extensions have to be implemented for modeling editors in order to use this approach. In contrast to Ohst et al., Mehra et al. consider tree-way merges. Although conflicting changes are detected by their differentiation algorithm, no attempt is made to indicate to the user that accepting one change may invalidate another, as explicitly stated in their paper. Concerning the concrete syntax changes, a diagram with many overlapping highlighted model elements is generated in cases where a large number of changes occurred. This is because, for each movement, the origin as well as the new place of each element with a line as connector is shown. The approach has been implemented for the meta-CASE tool Pounamu [227] for providing generic visualization support for modeling languages defined in Pounamu, but for UML modeling environments there is no support available.

Preserving the Mental Map between Diagrams. There is a number of works focusing on preserving the mental map across sequences of diagrams. The mental map refers to the modeler's personal perception of the model which is manifested in the diagram layout. In general, the sequences of diagrams are created by transforming the underlying model and adjusting the diagram to the evolved model. When the mental map shall be preserved, the goal is to keep changes of the layout at a minimum such that the modeler needs not to spend much effort in realignment. This aspect is the particular focus of previous work by Jucknath-John et al. [132], Pilgrim [221], Johannes and Gaul [130], and Grimm et al. [105].

Jucknath-John et al. [132] aim at layout graphs that are transformed by a sequence of endogenous graph transformations. Their goals are: (1) achieve an optimal quality for each single graph layout, (2) retain the mental map of a graph layout, and (3) allow to identify of the changes between two succeeding graph layouts by visually emphasizing the differences. To achieve these goals, the authors introduce the concept of node aging and protection of the layout of senior nodes, i.e., nodes that have been introduced earlier than others are less likely to be repositioned by the algorithm than younger nodes.

The focus of Pilgrim [221] is to retain the mental map in exogenous model transformations.

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

Johannes and Gaul [130] considere the diagram layout when composing domain-specific models. In their approach, the layout composition information is delivered through a graphical model composition script, which specifies how models should be composed. After the composed model is created, the diagrams of the composed model are merged into a new composed diagram according to the positions in the graphical model composition script. Finally, Johannes and Gaul also apply some algorithms to adjust the final layout to remove overlaps.

Grimm et al. [105] presented an approach for tackling the challenge of preserving the mental map when UML class diagrams have to be merged. Their approach is based on using one of the concurrently edited diagrams as so called *base diagram*, in which all modifications done for creating the other diagram, the so called *fitting diagram*, are included. For merging in a metal map preserving manner, the neighborhood of model elements in the fitting diagram has to be ensured also in the merged diagram as good as possible.

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

5.2 Merging in the Presence of Conflicts

In the context of optimistic model versioning, conflicts occur when the modifications of two or more models cannot be integrated in one consistent model. The reasons of conflicts are manifold as we argued in Chapter 3 where we established a conflict categorization. Merging is the intricate task of usually one modeler, i.e., the modeler who performs the later check-in, of integrating all changes into one consolidated version of the model. To preserve intentions behind all changes, the modeler needs to understand the evolution of the other modeler's version as well as how this version contradicts her own changes to the model. Obviously, to minimize the risk of losing modifications, both modelers may resolve conflicts together in a meeting or with the help of a tool-supported collaborative setting as proposed in [38]. However, an automatically computed tentative, pre-merged version of the consolidated model would be highly appreciated as accelerator for manual conflict resolution, regardless of whether the merge is performed by one single modeler, or in collaboration. Thus, we elaborate in this chapter on a merge strategy inspired by the works in the field of tolerating inconsistencies [13, 170, 200]. Accordingly, we need a representation for managing known conflicts in order to temporarily tolerate them. Before we establish the conflict representation, we shortly recapitulate exemplarily the essential excerpt

of the conflict categorization, covering solely conflicts, model versioning systems like AMOR (cf. Chapter 3) are capable to detect.

5.2.1 Conflict Categorization Revisited

We distinguish two groups of conflicts: *conflicts due to overlapping changes* and *violations*. Whereas the former is only the result of atomic changes (insert, update, delete), for the latter, also composite operations like refactorings may be involved. To detect these, language specific knowledge is necessary, whereas overlapping changes may be detected independently of the used modeling language. Violations are usually only detectable within the merged models when it becomes obvious that for example (meta)model constraints have been violated.

We discuss the various kinds of conflicts with the help of the example shown in Figure 5.1 and aim at giving an intuition how conflict detection components like the one implemented in AMOR (cf. Chapter 3) detect and report such conflicts. Conflicts themselves are represented in terms of models as discussed in the next subsection and serve as the input data for our visualization approach presented in this chapter.

The modelers Harry and Sally work together in a project, where an event management system has to be developed. To support their collaboration, the artifacts under development are exchanged via the central repository of an optimistic model versioning system. One day, Harry checks out the model which is named Origin Model in Figure 5.1 from the repository. This model contains a UML Class Diagram as well as a UML Sequence Diagram which specifies some interactions between certain elements of the Class Diagram.

Harry performs the following modifications. In the Class Diagram, he introduces a generalization relationship between the classes `Person` and `Customer`, such that `Customer` becomes the subclass of `Person`. Then he adds an association named `employedAt` between the classes `Employee` and `Office` with multiplicities `1` and `2`, respectively. Obviously, this multiplicities do not make much sense concerning the real world domain. We will come back to this issue a little later. Then he renames the operation `getInfo()` of class `Ticket` to `getTInfo()` and removes the attribute `artist` from the class `Exhibition`. Finally, he adds the two new classes `SoccerMatch` and `EventManager`. The class `SoccerMatch` gets subclass of `Event`. In the Sequence Diagram, Harry changes the message `getTInfo()` (recall that it was renamed in the Class Diagram) to an asynchronous message and adds a reply to this message which is also asynchronous. Between this request and reply messages he adds a message to an interaction partner of type `Office`. Furthermore, he introduces the call of `buy` from `Customer` to `Ticket`. He checks his modifications into the repository.

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

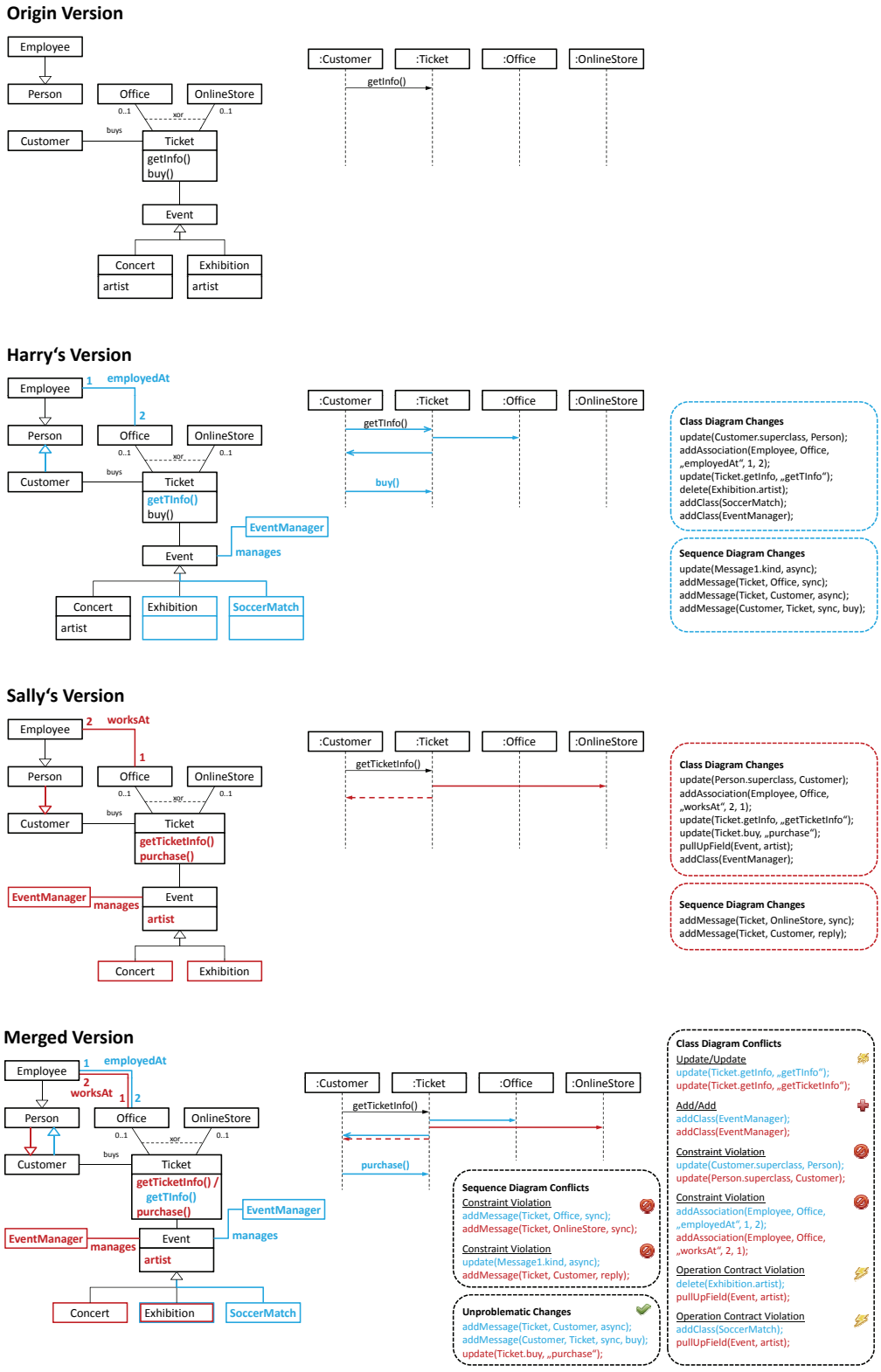


Figure 5.1: Model Versioning Example

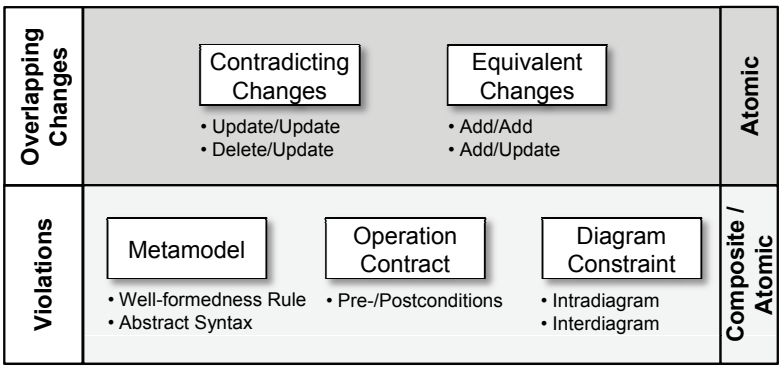


Figure 5.2: Conflict Categorization

after which she adds a reply message from `Customer` to `Ticket`.

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

1. *Contradicting Changes.* The probably most common conflict in model versioning is a conflict due to contradicting changes, where one modeler modifies a model element deleted by the other or where both modelers modify the same model element in different ways. In our example, this conflict occurs for the renaming of the operation `getInfo()`. When contradicting changes are reported strongly depends on the granularity level of the conflict detection, i.e., what kind of model element is considered to be atomic. For example if a class is considered as one atomic unit, a conflict is reported when the names of two different attributes are modified, if the granularity level is set at the level of the attributes, then no conflict is reported in this case.

2. *Equivalent Changes.* Harry and Sally both introduced a class called `EventManager`. Such a situation may be handled in various ways by the conflict detection component. (1) The two elements are considered as different elements, especially when they have internally unique IDs. Then this element is inserted twice in the merged model. (2) The two elements are considered as equal and they are merged, i.e., a model element is included in the merged model which contains all features of both. (3) A conflict is reported.

3. *Metamodel Violations.* Usually, a modeling language is specified in terms of a metamodel like the Superstructure [114] for UML. Even if the two modelers perform changes which do not impact the conformance of the model to the metamodel, under certain circumstances the combination of the modifications of both modelers might result in an invalid model. Consider for example the introduction of the generalizations in our example (the inheritance from `Person` to `Customer` for Harry, and the inheritance from `Customer` to `Person` in Sally's case). Another metamodel violation is shown in the Sequence Diagram. The message type of `getInfo()` is changed by Harry from synchronous to

asynchronous. Furthermore, an asynchronous message is added to model the reply to the request. Sally, in contrast, added an explicit reply message. In the merged version, this reply message follows an asynchronous message, what is not allowed.

4. *Operation Contract Violation.* Sally performed the refactoring `pullUpField` on the classes `Concert` and `Exhibition` on the common attribute `artist`. When combined with Harry's changes, two problems arise. (1) The newly introduced class `SoccerMatch` would inherit the attribute `artist` if the refactoring was applied. Usually few artists are involved in a soccer match, but this knowledge is seldom available to the conflict detection component. Still, this conflict may be detected and reported, because the class `SoccerMatch` violates the precondition of the refactoring, stating that all subclasses have to share the feature which shall be pulled to the superclass. (2) As Harry removes the attribute `artist` from `Exhibition`, the refactoring is not applicable to `Exhibition` anymore, as the precondition of the refactoring is not fulfilled for this class. Recently, the detection of conflicts due to refactorings and complex operations in general are for example supported by the versioning system AMOR (cf. Chapter 3).

5. *Diagram Constraint Violation.* In modeling languages like UML, models consist of different diagrams. A diagram provides a specific view on a certain aspect of the model. For example, UML provides the Class Diagram for describing the structure of a system and the Sequence Diagram for describing the interactions happening between certain modeling elements. Within a diagram constraints may be defined which hold for the diagram itself or constraints may be defined which restrict another diagram. We therefore distinguish *Intradiagram Constraints* and *Interdiagram Constraints*. A violation of an Intradiagram Constraint occurs in our example, because Harry and Sally both add an association between the classes `Office` and `Employee`, but with the opposite multiplicities. When analyzing the multiplicities, it becomes obvious that the model can never be instantiated, because one object of type `Employee` would need two objects of type `Office` and one object of type `Office` would need two objects of `Employee`. When this inconsistency is reported, it becomes also obvious that Harry has made a mistake. For the violation of Interdiagram Constraints consider the `xor`-constraint, which states that an object of type `Ticket` is either related to an object of type `Office` or an object of type `OnlineStore`, but not to both. When we merge the two different versions of the Sequence Diagram, we have a ticket that communicates with both, i.e., the `xor`-constraint is violated.

The different kinds of conflicts discussed above represent the most common conflicts identified in the recent model versioning literature and are implemented in the conflict detection components. In the categorization presented in Chapter 2, we also introduced ontological conflicts, which require domain knowledge for their detection. For example, having specific information, it could be reported that both modelers introduced a similar association between `Employee` and `Office`, one named `employedAt` and one named `worksAt`. A human modeler would probably recognize them as equal, but to the best of our knowledge no conflict detection component has been implemented so far which is able to detect and report such a conflict. Therefore, we do not consider this kind of conflict any further in this thesis.

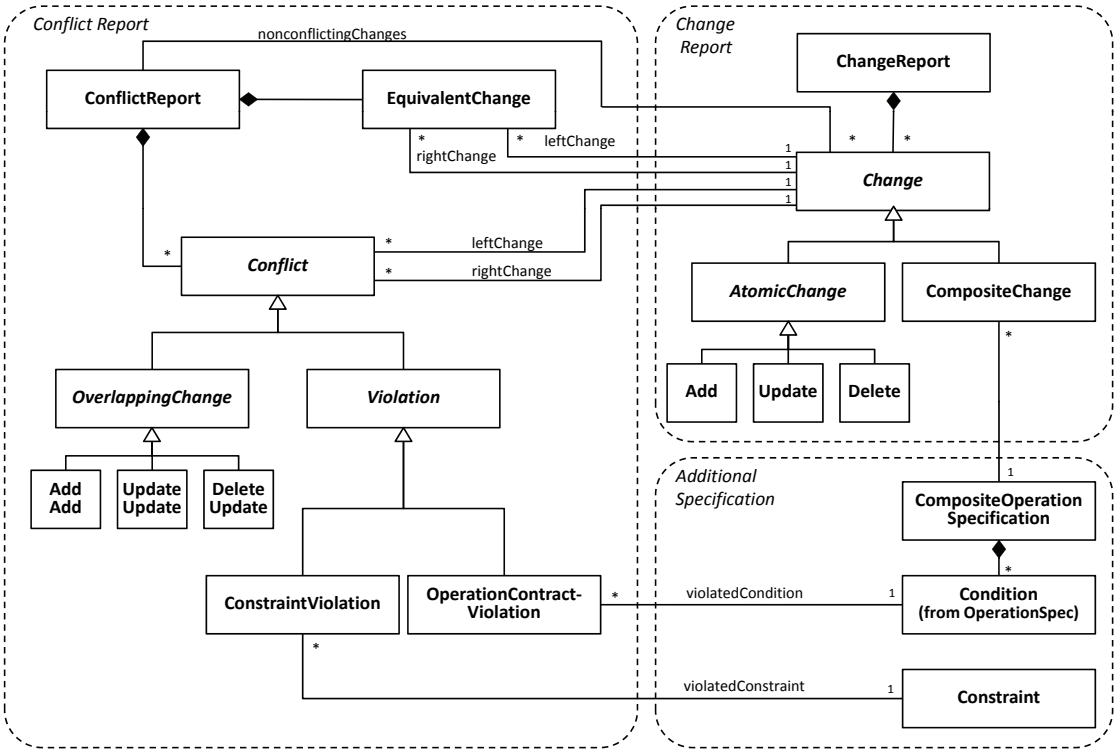


Figure 5.3: Conflict Model

5.2.2 A Model for Conflicts

The essence of a conflict are the involved model elements, the performed changes as well as the violated constraints. These constraints are either preconditions of a change or conformance rules defined in the metamodel of a modeling language. For further processing conflicts during the merge process, we need therefore an explicit representation. Similar as proposed by Cicchetti et al. [46], our means of choice is a model-based representation.

In our conflict model depicted in Figure 5.3, we assemble two sources of information for obtaining a *Conflict Report*, namely the *Change Report*, comprising the applied atomic or composite changes, and *Additional Specifications* formulating language specific operations like refactorings and conformance rules. All these information are obtained from an external conflict detection component like for example the conflict detector of the model versioning system AMOR (cf. Chapter 3).

With this conflict model we may profoundly express the kinds of conflicts shown in Figure 5.2. The conflict report consists of *Conflicts* and *Equivalent Changes*. Equivalent changes act as hint for a merge component, expressing that only one of both changes shall be included in the merged version and are not directly referred as conflict. Conflicts are either *Violations* or *Overlapping Changes*. A conflict caused by overlapping changes always references two changes which either interfere each other, or where one change makes the other change obsolete. *UpdateUpdate* conflicts are reported when two changes are not commutable, *DeleteUpdate* conflicts

occur for changes not applicable in combination; whether two overlapping changes are reported as *AddAdd* conflict, or as equivalent change strongly depends on the conflict detector. In the example of Figure 5.1 the renaming of the operation `getInfo()` is reported as overlapping change, and the duplicate introduction of the class `EventManager`, results in an equivalent change.

For conflicts due to violations we distinguish the following subclasses:

- *OperationContractViolation.* A conflict due to the violation of an operation contract always involves at least one composite change like a refactoring. This change cannot be performed because another change violates a precondition. Composite changes may be specified with a tool like the Operation Recorder as proposed in [36] or in terms of graph transformation as proposed in [213]. We distinguish two cases: a composite change, such as the `pullUpField` refactoring, is either not applicable because a model element violating the refactoring's precondition has been added (e.g., class `SoccerMatch` in Figure 5.1) or an existing model element necessary for the execution has been changed or deleted (e.g., an attribute of class `Exhibition` in Figure 5.1).

- *ConstraintViolation.* Furthermore, conflicts may arise if the merged model violates metamodel constraints or constraints specified within the diagram (constraining either the diagram itself or constraining other diagrams). For example in Figure 5.1, the inheritance cycle or the violation of the xor-constraint defined in the Class Diagram fall into this category of conflicts.

Instances of this conflict model serve in the following as the data input for the merge algorithm with the aim to calculate a tentative, pre-merged version of the model to further support manual conflict resolution. Note that this representation of a conflict is not adopted for a certain modeling language and might be even applied for conflicts on the concrete syntax if the diagram is represented as a model. For example, if two modelers move the shape of a model element to different positions, we have a conflict due to overlapping changes. In the following, we discuss the technical realization in more detail and explain how this conflict model serves as basis for the representation of conflicts within modeling environments.

5.2.3 Conflict Aware Merge Strategy

Resolving conflicts by manually exploring the common ancestor model and the two changed models in combination with the change and conflict reports shows to be cumbersome and error-prone in practice. Thus, the conflict aware merge strategy acts as accelerator for manual conflict resolution. It takes the common ancestor model and the conflict report as input and produces a tentative output model, unifying all changes of both modelers in a conflict aware manner. The major premise is to produce a syntactically correct model without losing model elements or changes. In this respect, only non-conflicting changes are merged. Deletions are skipped. Information on conflicting changes is incorporated directly in the merged model with the help of annotations, similar to *pollution markers* [13]. As a result, conflicting changes are not immediately *rolled-back*, but *temporarily tolerated* and reported to the user.

Merge Algorithm. The conflict aware merge strategy is shown in Algorithm 5.1. An overview of classes which are involved in the conflict report and used in the algorithm is depicted in Figure 5.3 and discussed in Subsection 5.2.2. In short, a conflict report explicates conflicts and equivalent changes based on a change report and additional specifications. Further, it is assumed, that model elements are identified by IDs and that changes are associated with the affected model element. The algorithm takes two inputs, i.e., the `originModel` and the `conflictReport`, and outputs the `mergedModel`. First, helper functions for working with the `conflictReport` are defined in Lines 1 and 2. The function `isConflicting()` checks if a change is involved in a conflict and `validate()` calls a validation engine to validate the `mergedModel` and update the `conflictReport`.

The actual merge process starts in Line 4 with copying the `originModel` to the `mergedModel`. In this way, the origin model is untouched and a new model version is created. In Lines 4 and 5, all distinct atomic changes and composite changes, i.e., all changes neglecting one of two equivalent changes, are retrieved from the `conflictReport` and stored in the respective `atomicChangeSet` and `compositeChangeSet`. The merge algorithm is then split into four phases.

1. *Merge Atomic Changes.* First, atomic changes are processed in Lines 6 to 19. Composite changes are applied thereafter. Thus, atomic changes may then be incorporated in composite changes [57, 85]. In order to retain all model elements and changes, changes of type `Add` are always applied to the `mergedModel` (Lines 7 to 9). In case of `Delete`, the change is not applied, but the deleted element is `marked` as deleted (Lines 10 to 13). `Update` changes are applied only, if they are not conflicting with other changes (Lines 14 to 18).

2. *Merge Composite Changes.* Composite changes are handled in Lines 20 to 30. If a composite change is conflicting, all involved elements are annotated (Lines 21 to 26). The composite change is only applied to the `mergedModel`, if it is not conflicting (Lines 27-29). The conflict detector reports conflicts for composite changes based on the pre-merged model, i.e., composite changes are replayed on the model including atomic changes and pre- and postconditions are validated. For more details about the conflict detector of AMOR, we kindly refer to [145].

3. *Annotate Overlapping Changes.* After executing all changes, overlapping changes are annotated in Lines 31 to 41. A conflict's elements affected by left and right changes are retrieved first. Then, if `leftElement` and `rightElement` are equal, e.g., in case of directly overlapping `UpdateUpdate` and `DeleteUpdate` conflicts, `annotate` is performed only once. Otherwise, both elements are annotated.

4. *Validate Model.* Finally, in Line 42 the `mergedModel` is validated and the `conflictReport` in enriched with validation conflicts. Those violations are retrieved in Line 43, and involved elements of each violation are annotated in Lines 44 to 49.

```
Input: originModel, conflictReport
Output: mergedModel
// Function declarations
isConflicting(Change) ↦ boolean
validate(Model, ConflictReport)

// Variable declarations
// copy the origin model as base for the new merged model
mergedModel ← originModel.clone()

// Retrieve changes from the conflict report
atomicChangeSet ← conflictReport.getDistinctAtomicChanges()
compositeChangeSet ← conflictReport.getDistinctCompositeChanges()

// Merging atomic changes
foreach change ∈ atomicChangeSet do
    if change instanceof Add then
        mergedModel.apply(change)
    end
    else if change instanceof Delete then
        element ← mergedModel
        .getElementById(change.getElement().getId)
        element.markAsDeleted()
    end
    else if change instanceof Update then
        if ¬ isConflicting(change) then
            mergedModel.apply(change)
        end
    end
end

// Merging composite changes
foreach change ∈ compositeChangeSet do
    if isConflicting(change) then
        // get elements bound to the composite change
        elementSet ← mergedModel
        .getElementsById(change.getInvolvedElements().getIds())
        foreach e ∈ elementSet do
            e.annotate(change)
        end
    end
    else
        mergedModel.apply(change)
    end
end
```

```
   // Annotating overlapping conflicts
31 foreach conflict ∈ overlappingChanges do
32 |  leftElement ← mergedModel
   |  .getElementById(conflict.getLeftChange().getElement().getId())
33 |  rightElement ← mergedModel
   |  .getElementById(conflict.getRightChange().getElement().getId())
34 |  if isEqual(leftElement, rightElement) then
   |  |  // Annotate changed Element
35 |  |  leftElement.annotate(conflict)
36 |  end
37 |  else
   |  |  // Annotate left changed Element
38 |  |  leftElement.annotate(conflict)
   |  |  // Annotate right changed Element
39 |  |  rightElement.annotate(conflict)
40 |  end
41 end
   // Validating constraints
42 validate(mergedModel, conflictReport)
43 violations ← conflictReport.getViolations()
44 foreach violation ∈ violations do
   |  // Annotate each element involved in a violation
45 |  elementSet ← mergedModel
   |  .getElementsById(violation.getInvolvedElements().getIds())
46 |  foreach e ∈ elementSet do
47 |  |  e.annotate(violation)
48 |  end
49 end
```

Algorithm 5.1: Conflict Aware Merge

Annotation Techniques. As described above, annotations are used to mark conflicting or inconsistent parts of the merged model. Those conflicts are tolerated to a certain extent and eventually corrected. In our case, conflicts are tolerated during the merge phase.

Annotations extend the model and carry information. Hence, annotations need an appropriate representation in the modeling language's abstract and concrete syntaxes. When designing a *new modeling language*, annotations supporting the merge process may be integrated from the get-go. However, creating new modeling languages goes hand in hand with building new editors and code generators, as well as preparing documentation and teaching materials, among others. Further, several modeling languages have already matured and may not be neglected when setting up versioning support. Thus, mechanisms are needed to *customize existing languages*. When directly *modifying existing languages*, the aforementioned issues remain unsolved, as newly introduced metamodel elements cannot be parsed by existing editors. According to [11], a lightweight language customization approach is desirable. For customizing immutable modeling languages like UML [114], *UML profiles* are the means of choice. UML profiles provide a language inherent, non intrusive mechanism for dynamically adapting the existing language to specific needs. As UML profiles are not only part of UML, but defined in the infrastructure specification [113], various modeling languages, which are defined as instance of the common core may be profiled and thus dynamically tailored. Recently, in an endeavor to broaden the idea of UML profiles to modeling languages based on implementations of Essential MOF [111], such as Ecore [60], several works have been published [146, 152].

5.3 Visualization of Conflicts

In the previous section, we presented a generic approach to merge conflicting models by temporarily tolerating merge conflicts with the help of annotations. We discussed, how existing modeling languages may be extended to support the proposed merge annotations. In this section, we apply the presented approach to the *Unified Modeling Language* (UML) [114], the de facto standard for MDE specified by the OMG. We make use of UML profiles and define a *Versioning Profile* for annotating the merged model and fostering visualization. In order to realize user support for the merge phase, we further extend the idea, to not only explicate conflicts, but also changes; hence, the complete evolution of a model is visualized directly in the model and is observable at a glance.

5.3.1 A UML Profile for Versioning

Our premise for visualizing conflicting UML models is that the user (1) should remain in her familiar UML tool, (2) should start with a tentative, automatically merged version comprising unproblematic changes, and (3) should be able to comprehend and reproduce changes and resulting conflicts. To this end, we adjust the generic conflict aware merge strategy shown in Subsection 5.2.3 to UML and present a dedicated *Versioning Profile* (cf. Figure 5.4). The versioning profile reflects the information available in the *Conflict Report* (cf. Figure 5.3) provided by the conflict detection component. This information allows the visualization of a model's evo-

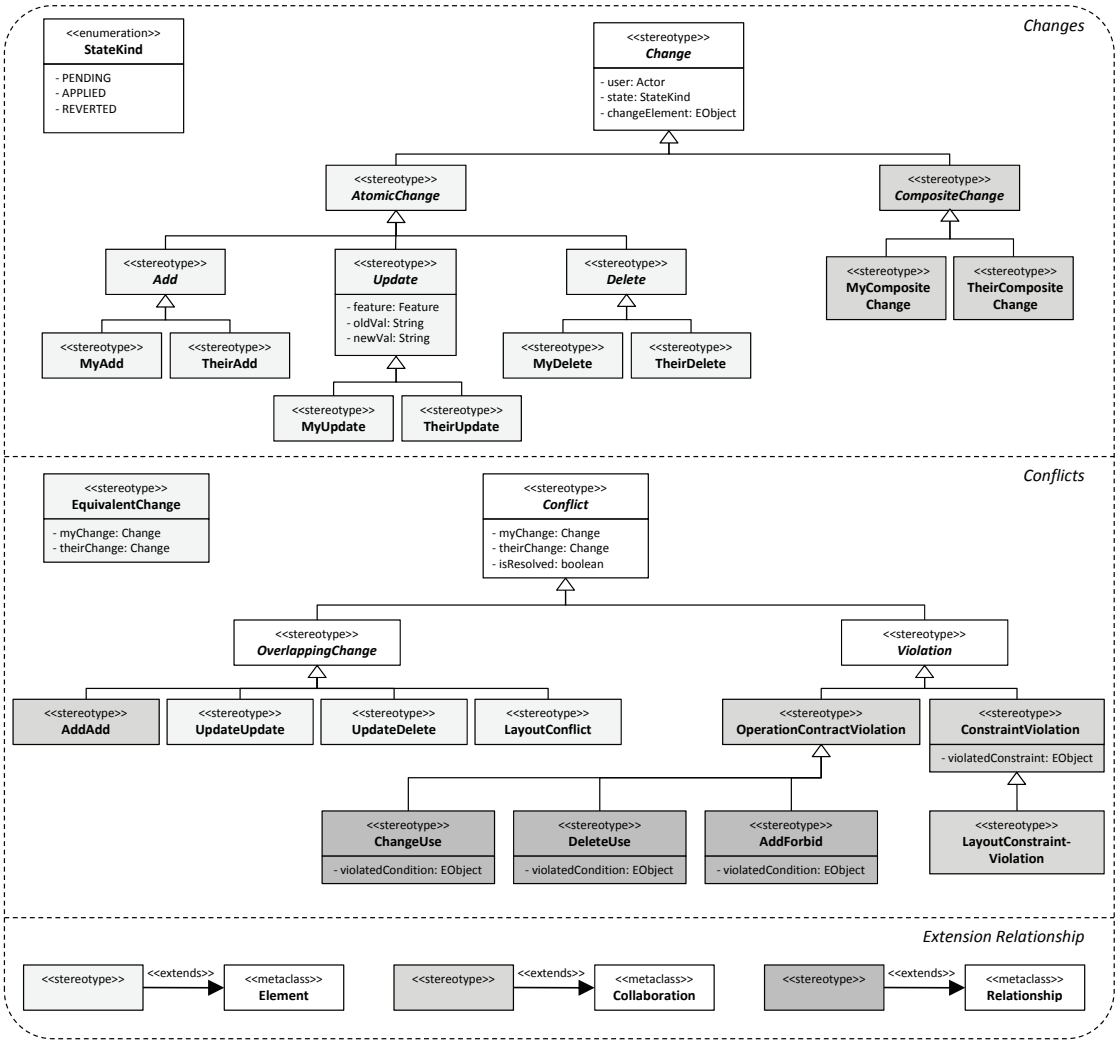


Figure 5.4: Versioning Profile

lution, i.e., the performed changes, as well as the merge conflicts directly in the UML model. Our design rationale for using UML profiles is based on the following requirements:

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

- *Integrated view:* All information necessary for the merge shall be visualized within a single diagram to provide a complete overview of the model's evolution.

- *Standard conform UML models:* The models incorporating the merge information shall be conform to the UML metamodel.

- *Model-based representation:* The merge information shall be explicitly represented as model elements to facilitate model exchange between UML tools, as well as postponing the resolution of certain conflicts.

- *No editor modifications:* The visualization of the merge information shall be possible without modifying the graphical editors of UML tools.

UML profiles define a lightweight extension to the UML metamodel and allow for customizing UML to a specific domain. UML profiles typically comprise *stereotypes*, *tagged values*, and additional *constraints* stating how profiled UML models shall be built. Stereotypes are used to introduce additional modeling concepts to extend standard UML metaclasses. Once a stereotype is specified for a metaclass, the stereotype may be applied to instances of the extended metaclass to provide further semantics. With tagged values, additional properties may be defined for stereotypes. These tagged values may then be set on the modeling level for applied stereotypes. Furthermore, syntactic sugar in terms of icons for defined stereotypes may be configured to improve the visualization of profiled UML models. The major benefit of UML profiles is, reflected by the fact that profiled models are still conforming to UML, that they are naturally handled by current UML tools. Thus, UML profiles provide a suitable mechanism to integrate annotations into UML models. In the following, we show how we map information of the conflict report to annotations. The conflict report may be obtained from any conflict detection component, but it has to obey the structure introduced in the previous section (which might be obtained by a simple adapter).

As shown in Figure 5.3, the conflict report assembles the change report, comprising all changes performed to the model and its respective views, and the conflict report, marking specific changes as equivalent, overlapping, or violating some constraint. The versioning profile reflects this separation by introducing dedicated stereotypes for changes and conflicts. Like in the conflict report, the change stereotypes comprise the information about how a specific model element has evolved, whereas conflict stereotypes are introduced to annotate merge problems and link to the respective changes. The versioning profile is derived from the previously described conflict report but adds additional information, which is only implicitly available in the conflict report.

Changes. The versioning profile depicted in Figure 5.4 provides stereotypes for each kind of change. To provide provenance information, each «*Change*» has tagged values to make the responsible user explicit. Additionally, as the stereotypes are not only used for mere visualization purposes, but also for supporting the merge process in terms of dedicated tooling (cf. Section 5.3), status information indicating whether the change is already applied, is introduced. To complement tooling related information, each change may be traced back to the corresponding change element in the change report. Changes are either atomic or composite. An «*AtomicChange*», i.e., add, delete, or update, may be applied to any concrete UML element, i.e., `Class`, `Generalization`, `Property`, etc., and thus, is defined to extend the UML metaclass `Element`. As updates are changes to existing elements, they have additionally tagged values for pointing to the affected feature of the changed element including its old and new value. Composite changes like refactorings, incorporate a set of indivisible atomic

changes. To highlight this fact and to express the interrelation of these changes, a new UML collaboration [114] is introduced and annotated with the stereotype «*CompositeChange*». The collaboration connects all model elements concerned by the composite change via UML relationships. Each specific kind of change stereotype is finally defined in the form of «MyChange» and «TheirChange» to indicate which changes were originally performed by the other user and which changes were applied by the user in charge of merging.

Conflicts. The conflict part of the versioning profile defines stereotypes for equivalent changes and the different conflict types depicted in Figure 5.3. Accordingly, «*EquivalentChange*» marks no conflict, as this change is applied only once, but gives a hint to the modeler in charge of merging. A «*Conflict*» may be either an «*OverlappingChange*» or a «*Violation*». «UpdateUpdate» and «UpdateDelete» stereotypes extend the UML metaclass `Element`, as these conflicts result from two atomic changes on the same model element. In contrast, Add/Add conflicts and violations comprise different modeling elements. Thus, we again introduce UML collaborations to hint at the involved changes. «ConstraintViolation» further states violated constraints. In case of a «OperationContractViolation», the UML relationships interlinking the involved elements to the UML collaboration, are annotated with stereotypes (inspired from graph transformation theory [143]) indicating how the contract is violated by the model element. The stereotypes «ChangeUse» and «DeleteUse» are applied on model elements already existing in the origin model, which are involved in a composite operation and changed or deleted by the other user, respectively. «AddForbid» indicates the addition of a new model element which invalidates the precondition of a composite operation. Finally, all equivalent change and conflict stereotypes refer via tagged values to the underlying change stereotypes, what makes understanding and reproducing the conflicts possible.

Please note that the type of tagged values referencing external information, i.e., `change`, `violatedConstraint`, and `violatedCondition`, is set to `EObject` for implementation reasons. However, as the versioning profile is derived from valid conflict reports only, we expect the types `DiffElement`, `OCLExpression`, and `OCLCondition`, respectively.

5.3.2 Conflict Aware Merging of UML Diagrams

With the versioning profile at hand, we are now able to adapt the conflict aware merge strategy for UML models. When merging UML models, one has to distinguish models and diagrams. UML 2 provides 14 different diagram types, each of which providing a dedicated view on special aspects of the model [114]. These diagrams have their own concrete graphical syntax and complement one another to render a coherent picture of the modeled system. Further, several diagrams of the same type may be defined to show partly overlapping parts of the model from different viewpoints or in different abstraction levels.

In the following, we consider to merge one model with arbitrarily many corresponding diagrams in a conflict aware manner. For each diagram, we generate a dedicated *Conflict Diagram*, i.e., a conflict aware merged version of the diagram, showing all relevant changes and detected conflicts at a single glance. Thus, for our motivating example presented in Figure 5.1, two conflict diagrams are generated: one for the Class Diagram and one for the Sequence Diagram.

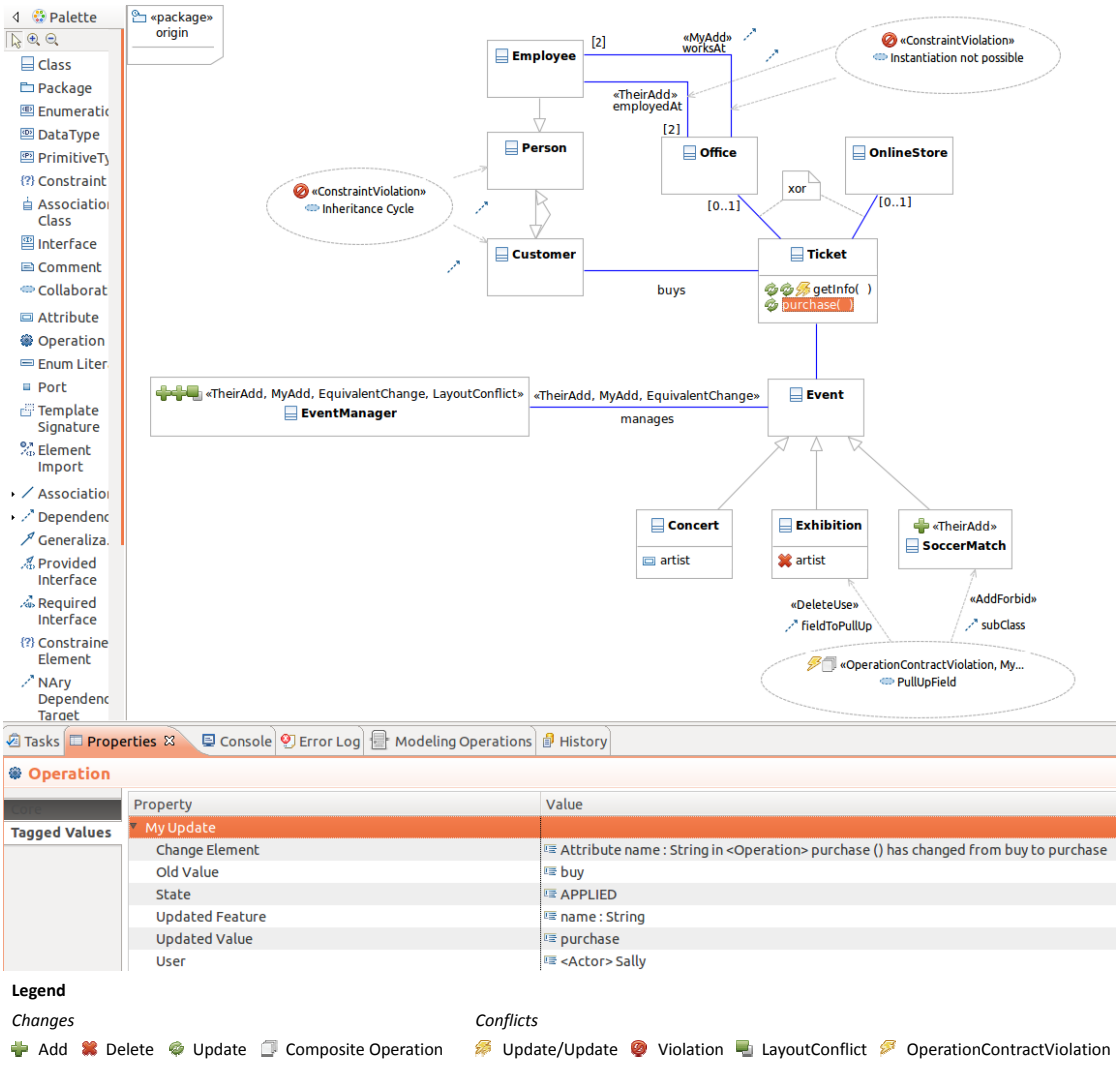


Figure 5.5: Conflict Diagram for the Running Example

The conflict diagram for the Class Diagram is depicted in Figure 5.5. The conflict diagram is enriched with change and conflict information. Though, it provides a familiar view, which the user in charge of merging can recognize as *her* diagram. It is obtained as follows:

1. All distinct additions and non-overlapping atomic updates are applied to a copy of the common base model; Equivalent changes are applied only once. Deletions are skipped, to allow annotating deleted elements with the respective stereotype (e.g., `artist` in class `Exhibition` in Figure 5.5). All composite changes are left out in this step since they are handled in Step 5. This allows for incorporating all atomic changes in the composite change (e.g., a refactoring) [57, 85]. More precisely, if the class `SoccerMatch` added by Harry had an attribute `artist`, which was shifted to the superclass by Sally with the

pullUpField refactoring, a re-execution of the refactoring would also include the new class and remove the attribute.

2. All changed elements are annotated with the corresponding change stereotypes defined by the type of change and the respective user. A link to the corresponding user and to the underlying Change of the change report is stored in tagged values. The Change element is later needed to execute or undo the change in the manual merge phase. The tagged value state indicates if a change is already applied, what is only the case for non-overlapping additions and updates. For updates, also the updated feature, old and new value are persisted. For an «Add» of Harry, cf. class SoccerMatch, annotated with «TheirAdd» as Sally did the later check-in and has now the burden of merging the models. Please note that the textual and graphical visualization of stereotypes varies for different modeling elements and editors. Thus the concrete peculiarity of «MyChange» and «TheirChange» may be visible not until visiting the property view, like in case of purchase in class Ticket in Figure 5.5.

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

4. Whenever equivalent changes occur, which cannot be reduced to one change, UML collaborations are added to connect the equal modeling elements. An «AddAdd» stereotype is added to these collaborations and again, both corresponding changes are stored in tagged values. As the class EventManager inserted by Harry and Sally independently is deep equal, i.e., all features and references are equal, no conflict is reported by our conflict detection component and both additions are marked as equivalent change. Hence, only one class is added to the model in Step 1. However, two possibilities for placing the corresponding shape in the diagram exist. As the conflict diagram shall provide a familiar view to Sally, because she is merging the model, the position of her version is applied. Further, a stereotype indicating a «LayoutConflict» is added to point her attention to Harry's different position for that shape.

5. As several modeling elements are involved in composite changes like in the pullUpField refactoring in Figure 5.5, we interlink them by introducing a UML collaboration annotated with a «CompositeChange» stereotype for each distinct composite change. Before applying composite changes, their preconditions are checked. If the preconditions are still valid, they are re-executed on the merged model. If the preconditions do not hold, an operation contract violation is at hand. Then the added collaboration is annotated with an «OperationContractViolation» stereotype and relationships to the elements, which do not longer fulfill the precondition, are marked with dedicated stereotypes. In our example, the relationship to the attribute artist is annotated with a «DeleteUse» stereotype, as the deletion of the property violates the precondition of the pullUpField refactoring, i.e., every subclass must have the field to be pulled up. Similarly, the relationship to the added class SoccerMatch is marked as «AddForbid». This also is due to the missing property artist.

6. Finally, the merged model is validated. All applied additions and updates are incorporated for validation, as well as deletions, except they are undone. Violated constraints are, again, marked by adding UML collaboration elements, interlinking the involved model elements and applying «ConstraintViolation» stereotypes. Their tagged value states the violated OCL constraint (e.g., Inheritance Cycle in Figure 5.5).

5.3.3 Making Conflicts Accessible

The tentative, automatically merged conflict diagram provides a kick-start for manually merging the models. The algorithm described above generates a *Neutral View* of all non-overlapping changes. Based on this conflict diagram, two supplementary views are supported. *My View* automatically privileges changes of the user in charge of merging the model. Thus, instead of skipping overlapping updates, the value of my change is used. *Their View* provides the opposite view, to allow the person who performs the merge to immerse herself in the situation of the other modeler. Additionally to applying values of their changes, also the layout information of the other user's diagram is used. In our example, this would affect the class `EventManager`, which was introduced by both modelers, but treated as equal by the conflict detection component. In their view, the class is visualized on the right of class `Event`. Violations are handled in my and their view as in the neutral conflict diagram, because currently we do not trace back to the exact change causing the violation. Thus, atomic changes dominate composite operations, and in case of constraint violations, both changes are applied and marked with the help of the collaboration. However, tools like the UML Analyzer [106] may be used for this task. The three conflict diagram views may be seamlessly transformed into each other and act as sandbox for checking various scenarios to better understand and resolve all conflicts.

The conflict diagram provides several benefits concerning the resolution of the conflicts. First of all, necessary information to resolve the occurred conflicts is provided at a single glance. Furthermore, different diagram filters based on the stereotypes may be used. With the help of these filters, specific kinds of stereotypes, i.e., conflicts, may be hidden enabling the user to focus on a specific conflict scenario. For example, a conflict resolution process can be supported such as firstly representing contradicting changes, subsequently, operation contract violations, and finally, constraint violation conflicts. Based on the user and state information of the stereotypes, the modeler responsible for the merge may switch between two further views, namely my view and their view to analyze different scenarios. The stereotypes enable additional mechanisms for visualizing conflicts directly supported by state-of-the-art UML modeling tools. As depicted in Figure 5.5, special icons are used for stereotyped elements. However, in case of loads of changes, the icons may quickly overwhelm the diagram. For example, the operation `getInfo` in class `Ticket` of Figure 5.5 is decorated with three icons, i.e., the change of Harry, the change of Sally, and the resulting Update/Update conflict. In such cases, information hiding would be helpful, e.g., suppressing change icons, as the conflict stereotypes incorporate change information anyway.

Going beyond visualizing the conflict diagram, extensions to the UML editor in form of dedicated *Merge Actions* may be implemented to interact with the stereotypes. Then, pending changes may be applied or reverted to resolve conflicts. Conflict stereotypes are then deactivated by setting them to `isResolved`. However, this needs re-execution of conflict detection after

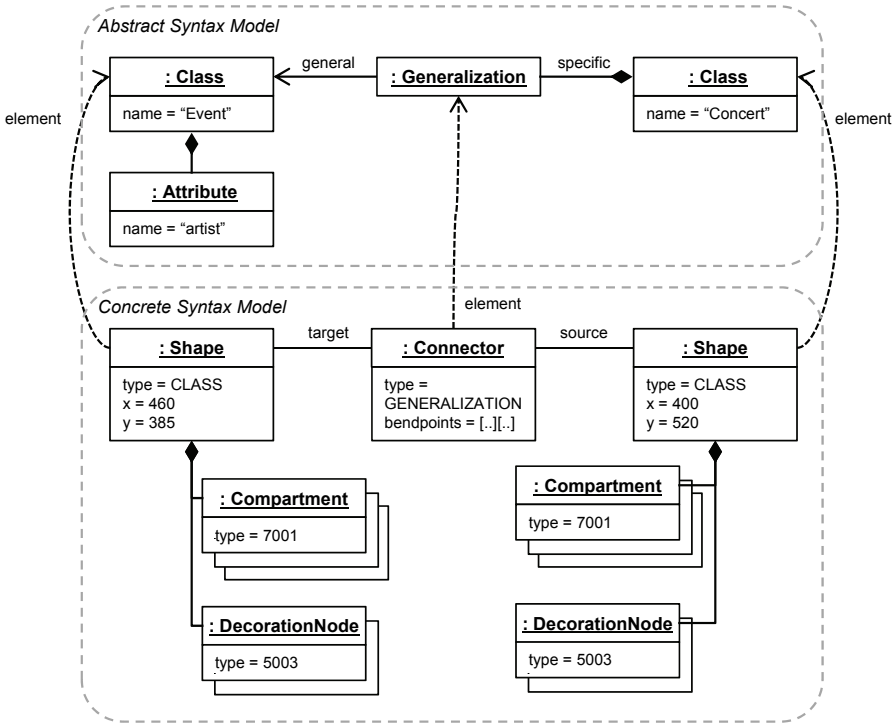


Figure 5.6: Models Representing Abstract Syntax and Concrete Syntax

each change. When checking in, elements marked for deletion are eventually deleted. Stereotypes for applied and reverted changes, as well as for resolved conflicts are removed. Pending changes and unresolved conflicts may be temporarily tolerated and checked in with the model for later processing, or handled over to another modeler as issue report.

The possibilities how to utilize the conflict diagram for manual merging are manifold. Insights about conflict resolution support implemented in AMOR are given in Chapter 6.

5.3.4 Merging Models and Diagrams

In the first part of this section, we proposed to report merge conflicts in the concrete syntax of UML by annotating the model elements with additional decorations and generating conflict diagrams for each corresponding diagram. Hence, it does not suffice to merge changes to the model only, but it is necessary to also merge diagrams. As diagrams are also models adhering to a graphical notation metamodel, differencing and merge components may be reused. Nevertheless, special care has to be taken to keeping model changes and diagram changes in sync.

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

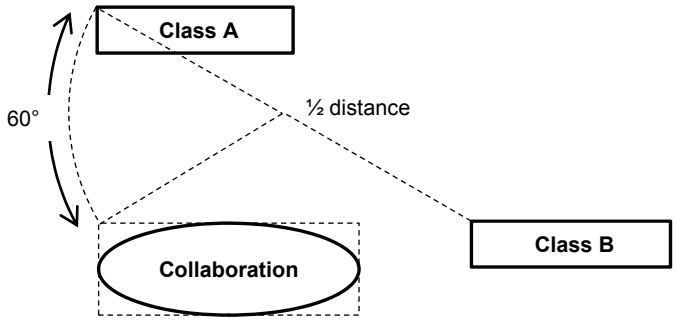


Figure 5.7: Placing Collaborations

unit. Whenever elements are added to the model and to the considered diagram, a shape should be also added to the conflict diagram.

Consider again the example shown in Figure 5.1. When switching from the concrete syntax to the abstract syntax of the diagram created with Eclipse's Class Diagram Editor UML2 Tools[1] (cf. Figure 5.6), we see that generalizations are not simple links between classes, but are model elements on their own which are mapped to connector elements in the visualization of the modeling editor. Further, primary view elements are mapped to elements of the model. Compartments and decoration nodes link only implicitly to the model elements and are part of a shape regardless of attributes or operations exist in the model. In contrast, decoration nodes for multiplicities of associations are only set for multiplicities different to `1`, which is the default value. Yet, the concrete syntax model highly depends on the modeling editor and thus, no *perfect* generic solution is possible.

Applying Stereotypes. Even though UML allows to apply stereotypes to every concrete element, applying change and conflict stereotypes is not straight forward and additional editor related information is needed. For example, when the multiplicity of an association is changed, the stereotype indicating that change should not be applied on the association's end property, but on the association, because stereotypes on the property are not visualized in the UML2 Tools Class Diagram Editor. In contrast, the UML tool Enterprise Architect[2] has no such restrictions.

Placing Collaborations. As several new UML collaboration elements and connectors are added to the conflict diagram, a suiting position has to be found in order to keep the diagram tidy and understandable. According to [224], the esthetics of UML diagrams affected by the spatial layout of nodes is crucial for understanding a model. To foster model comprehension, collaborations should therefore be placed as near as possible by the connected elements, whereas connectors should have approximately the same length [47, 224]. Thus we calculate the position of new collaborations by rotating the coordinates of the leftmost shape by 60 degrees, taking the center of the shapes to be connected as anchor, as depicted in Figure 5.7.

[1] `http://wiki.eclipse.org/MDT-UML2Tools`
[2] `http://www.sparxsystems.eu`

Handling Layout Conflicts. Like on the abstract syntax of the model, also on the concrete syntax of the model may occur conflicts. For example, if both modelers move a class in different directions, a conflict on the diagram occurs, even without any change to the model itself. Such kind of layout conflicts are currently automatically resolved, as the conflict diagram is generated using the layout of *MyUser* by default. Additionally, a «LayoutConflict» stereotype is applied.

5.4 Summary and Future Work

To summarize, we presented a generic approach for merging conflicting changes in a conflict aware manner. Conflicts are not neglected, but are treated as first class citizens by annotating the merged model. We further showcased the feasibility of the presented approach and applied the conflict aware merge strategy to UML models. For realizing annotations, we leveraged UML profiles, the powerful and dynamic extension mechanism of UML. Annotated models are still valid and may be opened by UML editors. We comprehensively discussed how the UML profile is derived from the conflict report and showed how the information about conflicts may be directly incorporated into UML diagrams.

As UML is a visual modeling language, we extended the conflict aware merge strategy to merge models and their accompanying diagrams. Hence, in contrast to other model versioning systems, we do not visualize conflicts on the abstract syntax of a model, but in the concrete syntax as used by the modelers. This allows the modelers to stick to their familiar notation when they are forced to resolve merge conflicts. Then the modeler in charge of conflict resolution may focus better on integration of the changes by interpreting only the changes of the others, whereas in the abstract syntax representation it would first be necessary to identify the own changes because the mental map, i.e., the personal view of the modeler on the model, is destroyed.

Our hypothesis is that with this approach model merging needs less time and is less error-prone than model merging on the abstract syntax. Although we think that our support will result in substantial improvement of the merge process, we see that there is the potential danger of overloading the model with too much additional information. Therefore, we focus our future work on developing methods to filter information in order to avoid confusion and disorientation. Additionally, we plan to further explore the advantages of stereotypes to realize an assistant for the manual conflict resolution process, by providing dedicated merge actions to resolve conflicts.

Another challenging open issue is handling layout conflicts. Similar to violations of the model, layout conflicts may not be obvious until the merge is performed. Additions or moves of different shapes may result in (partly) hidden shapes or crossing edges in the merged model. In order to preserve the mental map, we do not change the layout of the existing elements. Currently, it is up to the editor to avoid crossing edges and element overlaps due to the merge and the new collaborations. However, there are two main directions how to deal with layout conflicts in future work. (1) For hidden, or partly overlapping modeling elements, OCL constraints may be established. Then, layout constraint conflicts are handled like diagram constraint conflicts, i.e., with the help of UML collaborations. (2) To avoid diagram overload, a mental map preserving algorithm similar to the work of Jucknath et al. [132] may be implemented, where element overlaps are avoided by rearranging younger nodes and retaining the position of senior nodes.

CHAPTER 6

Conflict Resolution Support

Despite steady advancements in managing the evolution of software models, merging conflicting models remains a challenging undertaking. The modeler in charge of merging has to navigate through several artifacts to collect information necessary to effectually reproduce changes and conflicts. Recently, conflict aware merge strategies have been proposed, which integrate those solitary artifacts and provide a unified, tentatively merged model. Even if this model turned out to be helpful for analyzing the evolution, resolution support moves hardly beyond the choices "take my change", "take their change", or "abandon all changes and provide a new manual solution". As the same types of conflicts are likely to reoccur, mechanisms providing more sophisticated resolution support would be appreciated.

In this chapter, we learn from related work how recommendation systems assist software developers in implementing, debugging, and maintaining large code bases. We then analyze the task of conflict resolution from a technical point of view and elaborate on a recommendation system for conflict resolution in model versioning.

6.1 Related Work

The need for maintenance of software written in the 1960s and 1970s and the accompanying growth from several-thousand-line to multimillion-line code bases, led to observe program understanding as *the challenge* for the 1990s [50]. Tools emerged which help programmers to read and run the code by effectively presenting static and dynamic data on the programs to the user. To date, emerging recommender systems specifically tailored to software engineering, guide programmers to deal with the ever increasing complexity of software systems [189]. Tools supporting MDE are currently in the state of the early 1990s compared to tools for traditional software engineering. However, the exciting evolution of tool support for software engineering motivates us to strive for similar achievements in MDE.

In the following, we shortly review recommender systems in general, and give then an overview on recommender systems for software engineering.

Recommender Systems. With the upcoming mass of electronic documents, especially mails, several works on information filtering have been published in the 1980s [154, 180]. Inspired by these works, Goldberg et al. [103] coined the term *collaborative filtering* in the early 1990s, as they proposed to collect reactions of eager readers of newsgroup mails and use them as input for filters of more casual readers. Since then, exciting developments moved the field forward [141, 188]. Resnick et al. [187] proposed with the tool GroupLens an automated system for collaborative filtering based on ratings and the calculation of personalized predictions. Shardanand and Maes [204] proposed a similar method. Although they showcased their approach with the Ringo system, which makes personalized recommendations for music albums and artists, they mentioned the potential of their algorithms to recommend books, movies, news articles, and products, as well. Burke et al. [41] employed a knowledge-based approach and implemented a series of FindMe systems, assisting users browsing through large multi-dimensional information spaces. According to [141], the consensus that these new and emerging systems are all *recommender systems*, was not reached until the first Collaborative Filtering Workshop in 1996 and the resulting special issue of the Communications of the ACM [188].

Reflecting the diverse origins, to date there are four groups of recommender systems [129]. *Collaborative recommender systems* [125] purely rely on user profiles, ratings, and on the assumption that user A is interested in an item user B is recently interested in, if (1) user A and user B have strongly overlapping user profiles, and (2) user A is currently not aware of that item. Such systems have the advantage, that no information of the recommended items is needed. The disadvantages are that a large user base is essential, and that it is hard to deal with (new) items, having no or only few ratings. These problems are referred to the *cold-start problem* and *data-sparsity*. Approaching the drawback of content-agnostic systems, *content-based recommender systems* [126], rooted in the fields of information retrieval and information filtering, exploit specific characteristics and recommend items, similar to those a user was interested in the past. Content-based systems do not need a large user base and new items may be immediately recommended. However, item descriptions must be available—either extracted automatically or entered manually. Although the border is blurred, in cases, where no user profile is available and user preferences are derived in interactive dialogs, recommender systems are called *knowledge-based recommender systems* [128]. They are further distinguished in constraint-based systems and case-based systems. *Constraint-based systems* select and recommend items fulfilling given constraints. *Case-based systems* use similarity metrics and recommend items, which are to a certain extent similar to the given constraints. Finally, *hybrid recommender systems* [127] combine different approaches to mitigate problems of only one approach. Commonly, content-based techniques assist collaborative approaches during ramp-up.

Recommender Systems for Software Engineering. Although most prominent recommender systems are employed in online shops for marketing purposes, the application field of such systems is much broader. Several sophisticated recommender systems for software engineering (RSSE) emerged over the last few years, assisting software engineers in various task, such as reusing code, adopting APIs, and debugging, just to name a few. While RSSE may fall in any of the four groups mentioned above, their distinguishing characteristic is, that they are tailored to specific needs of software engineers.

RSSEs are defined as follows:

> *"An RSSE is a software application that provides information items estimated to be valuable for a software engineering task in a given context"* [189].

It is commonly accepted that clean, readable code reduces maintenance costs. Thus, it is recommended to refactor the code base from time to time, i.e., to improve the internal structure of existing code without altering the external behavior [94]. Even if object-oriented design patterns are mentioned as target of refactorings [98], performing refactorings is still challenging. The challenge is to identify the context of bad design and transform the code to a better design. Beck and Fowler [16] identify *bad smells*, discovering those critical parts of bad designed code, signalizing *when* and *where* to refactor. Further, they provide a refactoring catalog, summarizing valuable information *what* to do in such a situation. The authors retain human intuition and refrain from giving precise criteria or metrics for identifying the need of refactoring. Anyhow, even if not automatically executable, this framework may be seen as a simple form of a knowledge-based recommender system for software engineering, as it triggers context aware recommendations valuable for refactoring. Consequently, Tourwé and Mens [217] propose semi-automatic support for the refactoring process and use logic meta programming to automatically detect bad smells and recommend adequate refactorings to the user. Similarly, Kataoka et al. [134] discover refactoring candidates by revealing program invariants, which are dynamically derived by their Daikon tool [88, 89].

Additionaly, RSSE may actively support developers while programming. CodeBroker [225] is an RSSE for supporting code reuse. The development environment is proactively augmented with reusable component information queried from an annotated reuse repository. Code Conjurer [120] combines keyword search and test-driven search to assess the fittingness of reuse candidates without depending on annotations. Strathcona [118] assists developers using large APIs and recommends relevant examples by heuristically matching the code under development to a repository already using the API. FRAN [191] suggests related functions by purely exploiting structural information in the call graph, i.e., the graph representing the calling relationships between functions of a program. Zimmermann et al. [228] and Ying et al. [226] apply independently of each other data mining to change histories to detect changes often applied in combination. This information is then used to guide developers through changing the code by suggesting related changes and preventing the developer from performing incomplete changes. Hipikat [220] goes one step further and includes not only version histories in its mining process, but also electronic communication channels like newsgroup articles, bug reports, and forums. Hipikat analyzes relationships between these artifacts and establishes a project memory acting as knowledge base for recommending a developer related artifacts to her current task. Kim et al. [137] propose to learn project-specific bug and fix pairs from version histories and set up a bug fix memory as RSSE supporting debugging. Similarly, DebugAdvisor [10] assumes that the same bug or a very similar bug may be already fixed in another code branch and provides a systematic search tool to improve debugging productivity. To express the context of the current bug, it allows the user to define a fat query, i.e., a query containing structured and unstructured data including stack traces, debugger output, and natural language text. The search tool queries diverse data repositories associated with the project, such as bug reports, logs of interac-

tive debugger sessions, information on related code changes, and people, who can be consulted. Bettenburg et al. [19] address the problem that bug reports vary in their quality and implemented the recommender system Cuezilla to assist bug reporters to improve their bug reports.

Recently, RSSEs are also successfully applied in the field of MDE. Mens and Van Der Straeten [161] aim at incrementally resolving model inconsistencies by recommending resolution patterns to the user. They propose a tool on top of the graph transformation tool AGG [212] to manage and execute graph transformation rules to annotate and resolve model inconsistencies. Resolution rules are specified to first match undesired inconsistent situations in models and then transform the model in such a manner, that the inconsistency is resolved. They allow to define multiple resolution rules for one kind of inconsistency, which are recommended to the user who decides one resolution rule. Further, they analyze sequential dependencies between rules to inform the user about (1) inducing new inconsistencies when resolving an inconsistency, (2) invalidating resolution rules for other inconsistencies, and (3) reintroducing an already resolved inconsistency by resolving other inconsistencies. Similarly, the tool Model/Analyzer [68] is currently extended to not only incrementally check violated design rules, but also to generate appropriate fixing actions [185].

While current RSSE already provide great support for developers mastering their primary programming and debugging tasks, secondary development tasks like versioning are neglected. To the best of our knowledge, there is no dedicated RSSE supporting versioning of software artifacts, more precisely, supporting the merge process in the presence of conflicting changes, as we are interested in. Only in the field of ontology engineering, one tool, PROMPT [169], provides interactive support and recommendations to resolve conflicts resulting from the alignment and merge of two ontologies. However, the recommendations are quite limited, as there are only few predefined pairs of merge conflict type and corresponding suggestion. Further, the focus of PROMPT is not directly on merging two parallel evolved ontologies stemming from a common ancestor, but on integrating two independent ontologies.

The most related approach to our work is of Cicchetti [45], who defines conflict patterns for parallel evolving models going beyond directly overlapping changes. Those conflict patterns consist of two change patterns to express atomic or composite changes such as refactorings, which are not allowed to be applied to the merged version in combination. These patterns may further be augmented with a reconciliation strategy, indicating one of the change patterns to prevail the merge process, while the other change pattern is ignored. Cicchetti focuses on the automatic resolution of recurring conflict patterns assigning as less work as possible to the user in charge of merging the models. While automatic conflict resolution may reduce user effort to a minimum, the quality of the merge may get impaired—especially when not all intentions behind the changes are covered in the automatic resolution but only one change pattern is applied to the merged version. Yet inspired by this approach and the works in the RSSE field, we develop a recommendation system for model versioning suggesting not only one of the previously applied change patterns, but also completely new resolution patterns.

In the remainder of this chapter, we first discuss conflict resolution in model versioning in terms of model transformations and show further, how reusable patterns may be described. We then assemble the conflict resolution recommender system which is integrated in the model versioning tool AMOR (cf. Chapter 3), suggesting reusable patterns in recurring conflict situations.

6.2 Guiding Modelers Through Conflict Resolution

In order to resolve merge conflicts in state-of-the-art model versioning systems, the modeler in charge of merging has to manually remodel the conflicting model in such a manner, that the non commutative changes become commutative. To accelerate conflict resolution, we introduced a conflict-free, tentatively merged model in Chapter 5. However, even if single operations of pending conflict pairs may be re-executed automatically, reasonable conflict resolution integrating all intentions behind conflicting changes still remains a manual task. Yet, conflicts obeying the same structure may be resolved by the same pattern. Thus, to approach our goal of establishing a recommender system for conflict resolution, we need (1) a technique to identify and compare the structure of a conflict, and (2) describe and automatically execute the manual remodeling activity in terms of reusable patterns to get rid of the manual effort. In the following, we describe those patterns by means of graph transformations in the sense of the algebraic, category theoretical double-pushout (DPO) approach introduced in Section 4.2. Before we go into theoretical details, we present an informal overview of the Conflict Resolution Recommender System.

6.2.1 Overview on Conflict Resolution Recommendation

As discussed in Section 6.1, several works in the field of RSSE attempt to leverage existing knowledge implicitly available in software artifacts and repositories, to support developers in various tasks. Our vision of conflict resolution recommendation is similar to the ideas behind the RSSE tool DebugAdvisor [10], which assists developers in fixing recurring bugs; We aim at assisting modelers in resolving recurring merge conflicts. While DebugAdvisor searches loads of structured and unstructured data to compare bugs leading to irregularities regarding the quality of information about recommended bug fixes eventually presented to the user, we prefer to exploit purely structured information enabling the direct execution of found conflict resolution patterns to automatically reconcile a conflict. In order to reuse a conflict resolution pattern for future conflict situations, the pattern needs appropriate abstractions from certain values in the model, yet describing the necessary change for reconciliation. Consequently, we describe conflict resolution patterns in terms of model transformations, which perfectly meet this requirement. However, to make conflict situations comparable, the conflict's essence, i.e., the exact cause for the conflict needs to be explicit. Fortunately, the *Conflict Report* introduced in Section 5.2, used as representation for conflicts in AMOR and exchange format between conflict detection components and conflict resolution components, reflects this information in a structured manner. Such conflict descriptions indicate the conflict type, e.g., UpdateUpdate, OperationContractViolation, etc., and their corresponding changes, e.g., Add, Update, or the occurrence of a certain composite operation, effectively linking to the affected elements in the model. Our conflict resolution recommender system is solely based on this information. The workflow depicted in Figure 6.1 is as follows. As mentioned above, the conflict report is AMOR's exchange format between conflict detection and resolution components. Thus it acts as input for the conflict resolution workflow, effectively decoupling the proposed conflict resolution recommender system from a certain conflict detection component. The first step performed by the Conflict Aware Merger is orchestrating change and conflict information linked in the conflict report, yet dis-

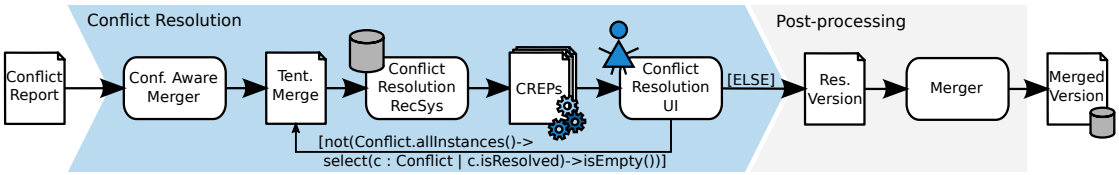


Figure 6.1: Conflict Resolution Workflow

tributed over several files, into one pre-merged model, called Tentative Merge. The tentative merge, introduced in Chapter 5, unifies all but deleting changes of both modelers in a conflict aware manner. As non-conflicting changes are already applied to the tentative merge, it acts as accelerator for semi-automatic conflict resolution. Conflicting changes are not ignored, but temporarily tolerated and marked with special annotations, leading to a holistic, model-based conflict representation. Those conflict annotations are then exploited by the Conflict Resolution Recommender System to search for existing, automatically executable Conflict Resolution Patterns (CREPs). The conflict resolution patterns are then attached to their respective conflict annotations and visualized in the Conflict Resolution UI. The user in charge of merging chooses a suggested conflict resolution pattern to reconcile one conflict after another. As long as unresolved conflicts remain in the tentative merge, the conflict resolution recommender system is triggered to lookup conflict resolution patterns. For conflicts where no or no suitable conflict resolution pattern is found, the resolution is performed manually and a new pattern is stored in the conflict resolution pattern repository. The Resolved Version is yield after reconciling all conflicts. Finally, a cleansed version without annotations called Merged Version is created in a post-processing step.

From a technical viewpoint, the conflict resolution recommender system is a graph transformation system. Conflict resolution patterns are defined as graph transformation rules, identifying a conflict's context in the LHS and providing a resolution in the RHS. Alternative resolution patterns are defined by stating rules with equal LHS graphs producing different RHS graphs. Instead of non-deterministically applying one of the matching rules, the user interacts with the conflict resolution UI and chooses one of the suggested rules or provides a new rule for resolving the conflict, which is then stored in the conflict resolution recommender system's pattern storage. Details are given in the next sections.

6.2.2 Description of Conflict Resolution Patterns

Equipped with the basic ideas on the conflict resolution recommender system and the knowledge on model transformation and graph transformation discussed in Chapter 4, we now elaborate on the description of reusable patterns for conflict resolution. As we have learned in Section 4.2, graph transformation rules are a useful mean to describe changes from one snapshot to another snapshot. In the model versioning system AMOR, conflict resolution is performed on the tentative merge obtained by the conflict aware merge strategy presented in Chapter 5. Thus, a resolution pattern's LHS graph should match an appropriate cutout of the tentative merge reflecting a certain conflict's context, i.e., its involved elements and annotations.

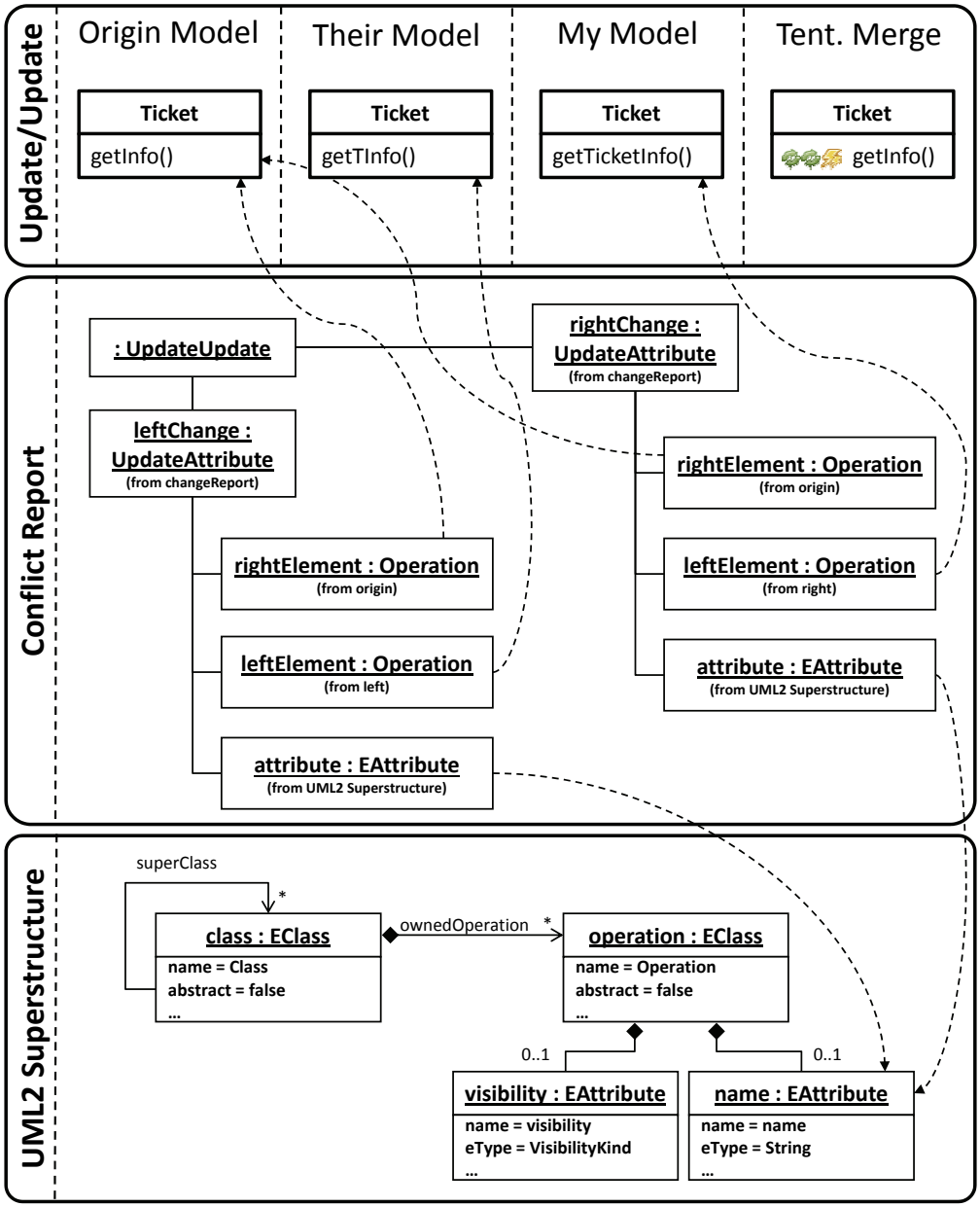


Figure 6.2: Conflict Model for an Update/Update Conflict

Consider for example an Update/Update conflict as occurred in the versioning example depicted in Figure 5.1. The operation `getInfo()` in class `Ticket` of the origin model is concurrently renamed by two modelers resulting in `getTInfo()` and `getTicketInfo()`, in their model and my model, respectively. Recall that the tentative merge keeps the origin name for the operation, as the changes are conflicting. The conflicting excerpt of the model and the resulting instance of the conflict model is shown in Figure 6.2. As discussed in Section 5.2, the

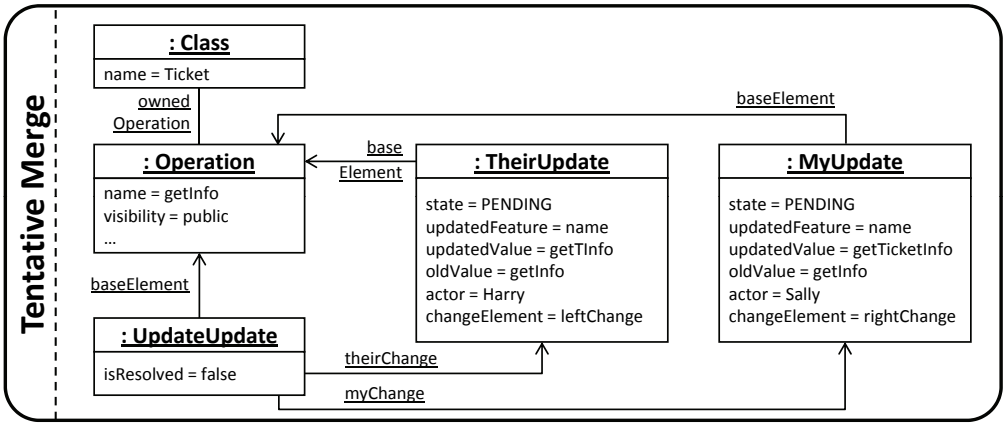


Figure 6.3: Tentative Merge for an Update/Update Conflict

conflict report consists of several `Conflict` elements, which link to `Change` elements of the change report. In AMOR, the change report is obtained by the differencing engine provided by EMF Compare [62] and enhanced afterwards. The result still conforms to the EMF DiffModel[1]. EMF Compare is not limited to specific modeling languages and provides generic differencing facilities for any Ecore based modeling language. In the conflict report for the example, the conflict object is of type `UpdateUpdate` and links to `leftChange` and `rightChange`—two instances of `UpdateAttribute` of the change report adhering to the EMF DiffModel. The `UpdateAttribute` element describes the actual change to an attribute in a generic manner by referencing to an `EAttribute attribute` as well as to an `EObject leftElement`, linking to the changed element, and an `EObject rightElement`, referring to the origin element. In this way, it is not tailored to the change of a specific attribute and is able to cover changes to any `EAttribute` of any `EObject`, defined in any metamodel based on the metametalanguage Ecore. In an instance of `UpdateAttribute`, the type of the changed attribute is defined by referencing to an `EAttribute` of the metamodel, the changed model conforms to. In our example, the operation's name is changed. Hence, the objects `leftChange` and `rightChange` refer both to the `EAttribute` with name `name` of the `EClass Operation` of Eclipse's Ecore based reference implementation of the UML Superstructure [63]. The `rightElement` points in both changes to the operation `getInfo()` of the origin model. The `leftElement` of the `leftChange` is a reference to the operation `getTInfo()` in their model, while the `leftElement` of the `rightChange` is a reference to the operation `getTicketInfo()` of my model. This probably misleading naming convention is due EMF Compare's capability of performing two-way and three-way comparisons, which results are expressed using the same EMF DiffModel. Nevertheless, with this information, the origin and changed values may be extracted. We use these values for the construction of the tentative merge and further in our conflict resolution patterns.

Figure 6.3 shows the tentative merge in its abstract syntax to illustrate how the conflict model and additional state information are incorporated. As discussed in Section 5.3, when using UML

[1] `http://www.eclipse.org/emf/compare/diff/1.1`

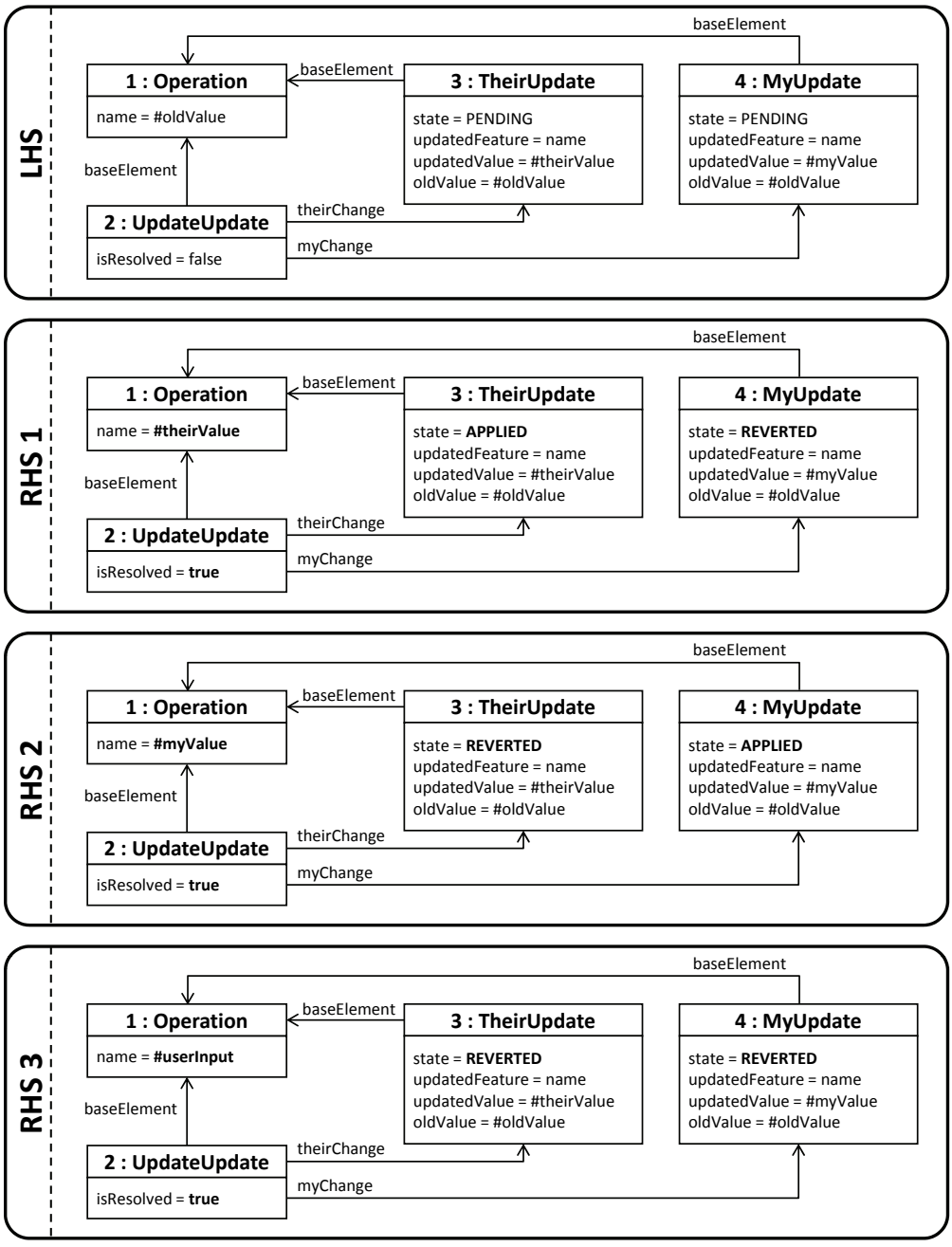


Figure 6.4: Resolution Patterns for an Update/Update Conflict

profiles as annotation mechanism, stereotypes for change and conflict annotations directly extend the UML metamodel and are thus instantiated as concrete elements at the model level. The stereotypes `TheirUpdate`, `MyUpdate`, and `UpdateUpdate` are applied to the concurrently renamed operation expressed by the `baseElement` references. Further information of the conflict model and resolution state is stored in so called tagged values, i.e., attributes of the stereotypes. Additionally, the stereotype `UpdateUpdate` refers to the conflict's underlying changes.

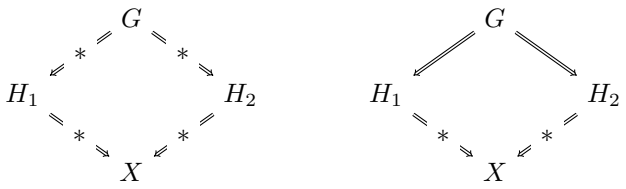
We now exploit this holistic structure for describing conflict resolution patterns to eventually establish a conflict resolution recommender system. As depicted in Figure 6.4, there exist several strategies how this conflict may be resolved. The LHS describing the pattern for matching the conflicting part in the model is shown only once, as it is the same for all replacement graphs. The search pattern describes an `Operation` with applied annotations `UpdateUpdate`, `TheirUpdate`, and `MyUpdate`, where `UpdateUpdate` has an attribute `isResolved` with value `false`, i.e., the conflict still exists. Note that only the attributes `state` and `updatedFeature` are set to fixed values. The name of operation `1` is not restricted to "`getInfo`", but set to a variable `#oldValue` indicating the origin value, which has to be equal to the attribute `oldValue` in the change annotations `3` and `4`[2]. For referring to the changed values of their and my version, the variables `#theirValue` and `#myValue` are introduced. Additionally, a variable `#userInput` is used to provide user defined values. Changed values in the replacement graphs are printed bold face for reading convenience. The strategy for resolving the Update/Update conflict proposed in RHS 1 in Figure 6.4 is to replace the origin operation name with the changed operation name of their version by setting the name to `#theirValue`. RHS 2 replaces the origin name with `#myValue`, and RHS 3 uses `#userInput` to set a new, user defined value to the name of the operation. Obviously, another resolution strategy is to keep the origin name or to combine my and their name.

The use of variables in the graph transformation based resolution descriptions allows for reusing the patterns for resolving other Update/Update conflicts; the Update/Update conflict's intrinsic structure is matched by a fixed pattern, while actual values are abstracted. In the following, we discuss the tentative merge construction and conflict resolution patterns as formal framework.

6.3 A Recommender based on Graph Transformation Theory

Having already motivated the idea of recommending conflict resolution patterns for conflict resolution in the previous sections, we now concentrate on the technical details of the conflict resolution recommender system developed in AMOR. The conflict resolution recommender system is implemented as *graph transformation system* based on the double pushout (DPO) approach (cf. Section 4.2). We use graph transformation, because of its preciseness given by its formal semantics. Further, the inherent properties of the DPO approach ensure the validity of produced graph structures and invertibility of direct graph transformations by construction. In the context of our conflict resolution recommender system, the DPO approach ensures that conflict resolution patterns never destroy the structure of a model and that a pattern may always be reverted.

[2]We use the prefix # to distinguish variables and explicit values



(a) Confluence (b) Local Confluence

Figure 6.5: Commutative Diagrams for Confluence and Local Confluence of GTS

Before we head to the actual conflict resolution, we summarize important properties of graph transformation systems needed for establishing a graph transformation system for conflict resolution. We then discuss the algebraic construction of the tentative merge and show that the resulting model yields a perfect basis for the conflict resolution recommender system, i.e., it exists, and is unique up to isomorphism.

6.3.1 Properties of Graph Transformation Systems

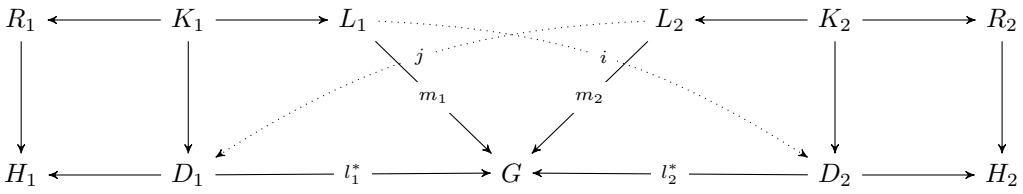
The properties of existence and uniqueness of an outcome may be achieved by a terminating and confluent graph transformation system.

Definition 6.3.1: (graph transformation system, graph grammar, and language). *A* graph transformation system $\mathcal{GTS}$ *is given by a set of productions* p_i *for* $i \in \{1 \dots n\}$*. A graph transformation system* $\mathcal{GTS}$ *with a given start graph* G_0 *defines a* graph grammar $\mathcal{G} = (\mathcal{GTS}, G_0)$*. The graph transformation* $\rho = (G_0 \overset{p_1}{\Longrightarrow} G_1 \overset{p_2}{\Longrightarrow} \dots \overset{p_n}{\Longrightarrow} G_n)$ *forms the* derivation *of the graph grammar, denoted by* $G_0 \overset{*}{\Rightarrow} G_n$ *for short. A graph language* $\mathcal{L}(\mathcal{G})$ *is given by the set of all graphs* G_n*, such that* $G_0 \overset{*}{\Rightarrow} G_n$ *[51, 74].*

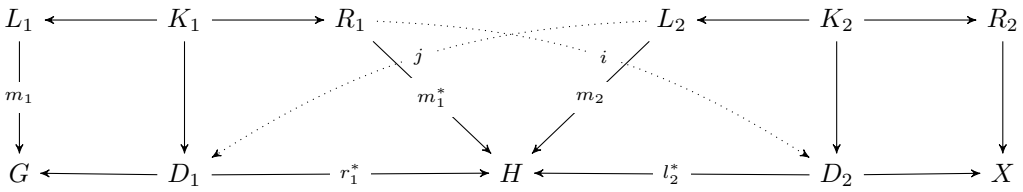
As discussed in Section 4.2, the application of productions is non-deterministic resulting in non-deterministic derivations of the graph grammar. However, global determinism of graph transformation systems yielding a unique or isomorphic result for each given start graph may be achieved under certain conditions, which are discussed in the following.

Definition 6.3.2: (confluence, local confluence). *A graph transformation system* $\mathcal{GTS}$ *is* confluent*, if for each graph transformation* $G \overset{*}{\Rightarrow} H_1$ *and* $G \overset{*}{\Rightarrow} H_2$*, there exists a graph* X *with* $H_1 \overset{*}{\Rightarrow} X$ *and* $H_2 \overset{*}{\Rightarrow} X$ *as depicted in Figure 6.5a.* Local confluence*, also denoted by weak confluence, is given, if there is a graph* X *with* $H_1 \overset{*}{\Rightarrow} X$ *and* $H_2 \overset{*}{\Rightarrow} X$ *for all pairs of direct graph transformations* $G \Rightarrow H_1$ *and* $G \Rightarrow H_2$*, as shown in Figure 6.5b [74].*

A graph transformation $G \Rightarrow H$ is terminating, if there is no more production p in the set of productions of the corresponding graph transformation system $\mathcal{GTS} = (P)$, which is applicable to H [74]. If all graph transformations within a graph transformation system $\mathcal{GTS}$ are terminating, then $\mathcal{GTS}$ is terminating.



(a) Parallel Independence



(b) Sequential Independence

Figure 6.6: Commutative Diagrams for Independent Direct Graph Transformations

Definition 6.3.3: (termination). *A graph transformation system $\mathcal{GTS}$ is said to be terminating if all sequences of graph transformations $(\rho_n : G_0 \overset{n}{\Longrightarrow} G_n)$ are finite, such that there is no $\rho_{n+1} = G \overset{\rho_n}{\Longrightarrow} G_n \Longrightarrow G_{n+1}$ [74].*

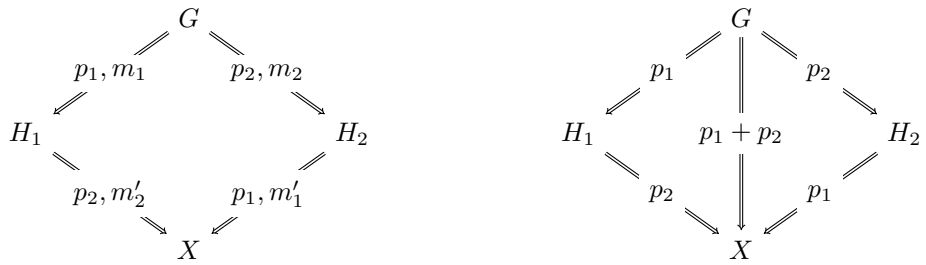
Remark 6.3.4: According to [119] a terminating and locally confluent rewriting system is confluent, which also holds for graph transformation systems [74].

In the following, we focus on a special case of confluence based on parallel independence, as it satisfies the requirements for constructing the tentative merge. In general, confluence is not restricted to parallel independence, but parallel independence of each pair of direct graph transformation is a sufficient condition for confluence of a $\mathcal{GTS}$.

Definition 6.3.5: (parallel independence). Parallel independence *is given for two direct graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ if their matches overlap in gluing items only, such that the diagram shown in Figure 6.6a commutes, i.e., there exist morphisms $i : L_1 \to D_2$ and $j : L_2 \to D_1$ with $l_2^* \circ i = m_1$ and $l_1^* \circ j = m2$ [74].*

Besides the parallel computation model, where parallel independent direct graph transformations are executed simultaneously, there is a second computation model for the parallel application of productions; the sequential model describes consecutive productions executed in arbitrary order, leading to a unique result if they are sequentially independent [51].

Definition 6.3.6: (sequential independence). *Two direct graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H \overset{p_2,m_2}{\Longrightarrow} X$ are sequentially independent, if the co-match m_1^* and the match m_2 overlap in gluing items only with respect to both transformations, i.e., there exist morphisms $i : R_1 \to D_2$ and $j : L_2 \to D_1$, such that $l_2^* \circ i = m_1^*$ and $r_1^* \circ j = m_2$, as depicted in Figure 6.6b [74].*



(a) Local Church-Rosser Theorem for $\mathcal{GTS}$

(b) Parallelism Theorem

Figure 6.7: Commutative Diagrams for Local Church-Rosser and Parallelism Theorems

If the direct graph transformations $G \overset{p1}{\Longrightarrow} H_1$ and $G \overset{p2}{\Longrightarrow} H_2$ are parallel independent, they are also sequentially independent, i.e., $G \overset{p1}{\Longrightarrow} H_1 \overset{p2}{\Longrightarrow} X$ and $G \overset{p2}{\Longrightarrow} H_2 \overset{p1}{\Longrightarrow} X$ yield the same object X [71]. This symmetry is shown in the Local Church-Rosser Theorem.

Theorem 6.3.7: (Local Church-Rosser Theorem for $\mathcal{GTS}$). *For two given parallel independent direct graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$, there exist direct graph transformations $H_1 \overset{p_2,m'_2}{\Longrightarrow} X$ and $H_2 \overset{p_1,m'_1}{\Longrightarrow} X$ yielding the same graph X, such that $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m'_2}{\Longrightarrow} X$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2 \overset{p_1,m'_1}{\Longrightarrow} X$ are sequentially independent.*

For two given sequentially independent direct graph transformations $G \overset{p_1,m_1}{\Longrightarrow} H_1 \overset{p_2,m'_2}{\Longrightarrow} X$, there exist two direct graph transformations $G \overset{p_2,m_2}{\Longrightarrow} H_2 \overset{p_1,m'_1}{\Longrightarrow} X$ such that $G \overset{p_1,m_1}{\Longrightarrow} H_1$ and $G \overset{p_2,m_2}{\Longrightarrow} H_2$ are parallel independent. The commutative diagram is given in Figure 6.7a [74].

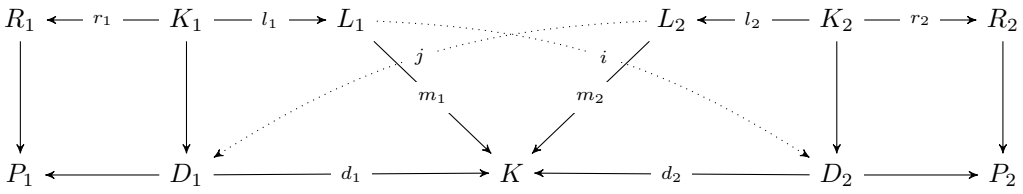
Parallelism may be represented more explicitly by abstracting from any possible application order and resulting intermediate graphs. This abstraction is modeled by a *parallel production* simulating the simultaneous application of single productions [51].

The single productions p_1 and p_2 of a parallel production $p_1 + p_2$ may overlap in gluing items, as this kind of overlapping is implicitly handled by a non-injective match of the parallel production [211].

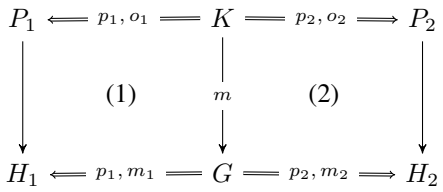
Definition 6.3.8: (parallel production, parallel direct graph transformation). *The parallel production $p_1 + p_2$ of two given productions $p_1 = (L_1 \overset{l_1}{\leftarrow} K_1 \overset{r_1}{\rightarrow} R_1)$ and $p_2 = (L_2 \overset{l_2}{\leftarrow} K_2 \overset{r_2}{\rightarrow} R_2)$ is defined by the co-product construction, i.e., disjoint union of the corresponding objects and morphisms: $p_1 + p_2 = L_1 + L_2 \overset{l_1+l_2}{\longleftarrow} K_1 + K_2 \overset{r_1+r_2}{\longrightarrow} R_1 + R_2$.*

The application of a parallel production is denoted parallel direct graph transformation *[71].*

Parallel productions $p_1 + p_2 + \ldots + p_n$ for $n > 2$ may be constructed analogously by iteration [51]. A parallel production may further be constructed and de-constructed bijectively as shown in the parallelism theorem.



(a) Critical Pair



(b) Completeness of Critical Pairs

Figure 6.8: Commutative Diagrams for Critical Pair and Completeness of Critical Pairs

Theorem 6.3.9: (Parallelism Theorem). *For a given sequentially independent graph transformation $G \overset{p1}{\Longrightarrow} H_1 \overset{p2}{\Longrightarrow} X$, there exists a parallel graph transformation $G \overset{p_1+p_2}{\Longrightarrow} X$, called* synthesis construction.

For a given parallel graph transformation $G \overset{p_1+p_2}{\Longrightarrow} X$, there is a construction called analysis construction*, leading to two sequentially independent graph transformations $G \overset{p1}{\Longrightarrow} H_1 \overset{p2}{\Longrightarrow} X$ and $G \overset{p2}{\Longrightarrow} H_2 \overset{p1}{\Longrightarrow} X$. Synthesis and analysis constructions are bijectively correspondent, such that the diagram in Figure 6.7b commutes [71].*

If two direct graph transformations are not parallel or sequentially independent, they are denoted parallel or sequentially dependent, respectively. The minimal context of a parallel dependency is given by a critical pair. Two parallel dependent direct graph transformations forming a critical pair are considered to be in conflict.

Definition 6.3.10: (critical pair, completeness of critical pairs). *A* critical pair *is given by two direct graph transformations $P_1 \overset{p_1,o_1}{\Longleftarrow} K \overset{p_2,o_2}{\Longrightarrow} P_2$, where $\nexists i : L_1 \to D_2$ such that $d_2 \circ i = m_1$ and (m_1, m_2) jointly surjective or $\nexists j : L_2 \to D_1$ such that $d_1 \circ j = m_2$ and (m_1, m_2) jointly surjective, as depicted in Figure 6.8a [74, 143].*

For each pair of parallel dependent direct graph transformations $H_1 \overset{p_1,m_1}{\Longleftarrow} G \overset{p_2,m_2}{\Longrightarrow} H_2$, there is a critical pair $P_1 \overset{p_1,o_1}{\Longleftarrow} K \overset{p_2,o_2}{\Longrightarrow} P_2$, such that (1) and (2) of Figure 6.8b are extension diagrams and $m : K \to G$ an injective morphism [74].

As each pair of parallel dependent direct graph transformations yields a critical pair, local confluence of a $\mathcal{GTS}$ may be implied if the corresponding set of all critical pairs is empty.

In the following, we make use of the presented properties of graph transformation systems and establish a graph transformation system for conflict resolution. The graph transformation

based conflict resolution recommender system is built upon four layers reflecting the priorities of rule application. Merge rules are obtained by analyzing changes and conflicts of the evolved models. Conflict resolution patterns are globally available in an extensible repository.

Layer 0. Layer 0 has highest priority and its rules are applied first. At this layer, the conflict-free tentative merge is constructed and annotated automatically. Annotations are inserted to changes as well as to conflicts.

Layer 1. Layer 1 contains merge rules which are applicable with slightly different match. At this layer, rules are applied automatically with a warning annotation. Additionally, this layer enforces metamodel constraints and contains rules for adding annotations to violated parts in the model.

Layer 2. Layer 2 forms the actual conflict resolution recommender system and is connected to the conflict resolution pattern repository. Conflict resolution patterns may match annotations inserted in layers 0 and 1 and are suggested automatically, but selected and applied manually. Complete manual conflict resolution is also possible by providing a new resolution rule which is then stored in the repository. The conflict resolution $\mathcal{GTS}$ is based on the DPO approach and hence, any automatically or manually applied graph transformation may be reverted in this layer.

Layer 3. Finally, layer 3 performs a cleansing step and contains rules for removing annotations, non-reverted deletions, and reverted additions.

We start with analyzing merge rules and establish layers 0 and 1 in the next section.

6.3.2 Algebraic Construction of the Tentative Merge

As already presented in Chapter 5, conflict resolution is performed on the basis of a tentatively pre-merged version obtained by applying additions, non-conflicting atomic updates, and non-conflicting composite operations to the common ancestor model. Deletions and conflicting modifications are omitted and eventually reported to the user in charge of merging. Please recall that change and conflict detection is not scope of this thesis, as we rely on the conflict detection engine presented in [145]. Hence, we are directly working with a *Conflict Report* introduced in Subsection 5.2.2 where conflicting and non-conflicting modifications of both sides are explicitly available.

When relying on a state-based comparison of evolved models as we do, the actual production of the performed graph transformation is not available. However, the performed modifications may be calculated by comparing the evolved models with their ancestor model as for example done by EMF Compare for Ecore based models. In order to lift those modifications to graph transformation theory, the notion of *graph modification* is introduced in [213] as generalization of graph transformations to graph changes. According to [80, 213], graph modifications are not necessarily rule-based changes to graphs. However, the definition of graph modifications is heavily inspired by the definition of graph transformation rules of the DPO approach. In fact, a graph modification is considered as graph transformation without its rule and match. As a rule in

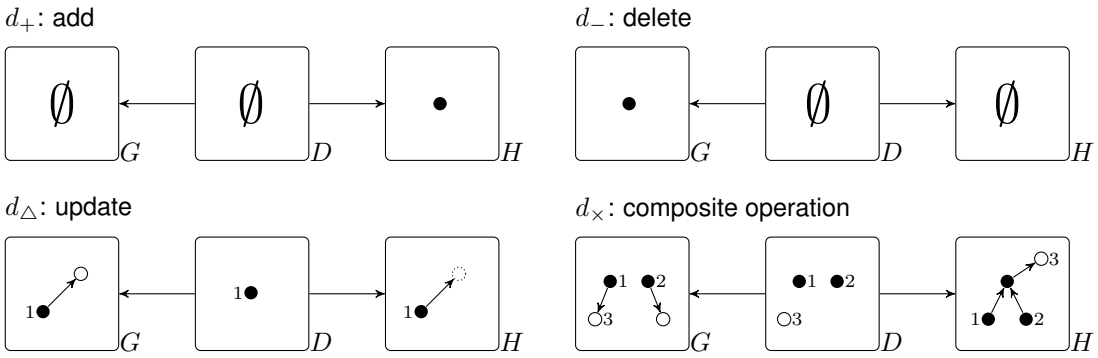


Figure 6.9: Exemplary Graph Modifications for Change Types

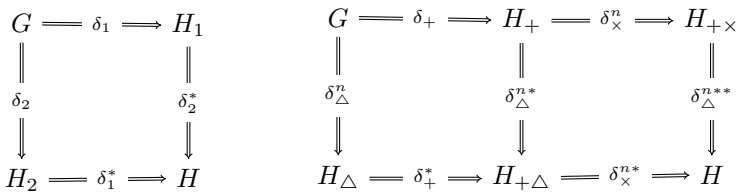
the DPO approach, a graph modification is given by a partial injective graph mapping expressed by introducing an interface graph and a span of two total injective graph mappings.

Definition 6.3.11: (direct graph modification, graph modification). *A* direct graph modification $G \Rightarrow H$ *between two graphs* G *and* H *is given by a span of injective morphisms* $G \overset{g}{\leftarrow} D \overset{h}{\rightarrow} H$. *A* graph modification $G \overset{*}{\Rightarrow} H$ *is given by a sequence* $G = G_0 \Rightarrow G_1 \Rightarrow \cdots \Rightarrow G_n = H$ *of direct graph modifications [213].*

The interface graph D *satisfies* $G \supseteq D \subseteq H$*, i.e., deletions are already applied, while additions are not applied.*

We distinguish the two groups of graph modifications: atomic changes add, update, delete, and composite changes denoted composite operation, framing user defined combinations of atomic changes.

Example 6.3.12: Graph Modifications for Change Types We summarize atomic and composite changes in Figure 6.9 showing exemplary graph modifications using a simplified notation: black circles represent elements, white circles attributes, and $\emptyset$ an empty set. The direct graph modification d_+ shows the addition of an element in graph H. As the element is newly inserted, there is no correspondence in the origin graph G, and therefore none in the gluing graph D. Consequently, such an addition may never be in conflict with any other change. However, if the added element is embedded into a certain container element, graphs G and D hold the context graph, which may overlap with other graph modifications. A delete operation is depicted as direct graph modification d_-. The element matched in graph G is deleted and has no correspondence in graphs D and H. An overlapping of the deleted element with another graph modification preserving that element results in a critical pair, i.e., a conflict. The direct graph modification $d_\triangle$ shows an update of an element's attribute. The attribute matched in graph G is deleted and substituted by a new attribute in graph H. Hence, the attribute has no counterpart in the gluing graph D. Finally, the direct graph modification $d_\times$ represents the application of a user defined composite operation. Two elements with certain attributes are matched in graph G. One attribute and all links between the matched elements and attributes are deleted and have consequently no counterpart in the gluing graph D. In the resulting graph H, a new element is



(a) Merge by User

(b) Merge by Change Type

Figure 6.10: Parallel Graph Modification

inserted linking to the preserved attribute. The new element in turn, is linked by the preserved elements.

Graph modifications derived from state-based comparison of two models are sufficient for merging models. Taentzer et al. [213] and Ehrig et al. [80] already use graph modifications in the context of model merging. In [213] changes from the common ancestor model to each parallel evolved model are considered as graph modification. Conflicts between both graph modifications are detected by reconstructing their respective minimal rule [24] and checking for parallel independence, i.e., whether the overlapping of both matches is preserved (cf. Definition 6.3.5). If the minimal rules are not conflicting, the graph modifications of both users $G \xrightarrow{\delta_i} H_i$ for $i \in \{1, 2\}$ are merged analogously to the construction of parallel productions (cf. Definition 6.3.8) as shown in Figure 6.10a. In case of conflicts, the merge cannot be constructed and the merged model does not exist. Ehrig et al. [80] extend this approach to merge any parallel modification irrespectively whether the modification is conflicting or not. They extend the parallel construction of $\delta_1 + \delta_2$ to a general merge construction, which formally resolves merge conflicts by prioritizing updates over deletions.

We pursue the ideas of [80] and construct a tentatively merged model even in the presence of conflicts. However, our goal is not to automatically resolve conflicts by fixed priorities, but to provide a conflict-free basis for a recommendation system supporting semi-automatic conflict resolution. As discussed in Subsection 6.2.2, conflict resolution is based on conflict resolution patterns matching certain annotations. Hence, we need a merge construction where the result is (1) maximal in the sense that no elements are deleted, (2) unique such that there is no combination of direct graph modifications leading to a different result, and (3) always exists. While in [80, 213] the complete modification of each user is treated as graph modification, we refrain from separating modification sequences by users, but separate them by their kind of modification as shown in Figure 6.10b. In the model versioning system AMOR, the sequences of both users' graph modifications are analyzed and de-constructed in occurrences of composite operations and atomic modifications to perform a more fine-grained conflict detection. We then obtain conflicting and non-conflicting changes by their type from the conflict report provided by the conflict detection engine. The fine-grained treatment of separated atomic and composite changes renders a more sophisticated merge construction possible. Our premise for the tentative merge is to embrace non-conflicting changes such that the resulting model is maximized.

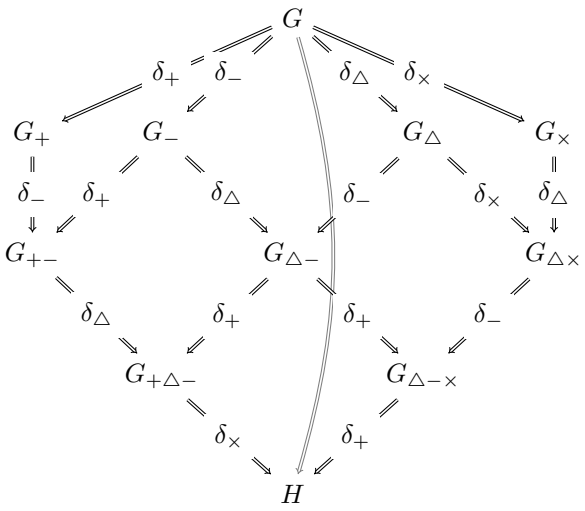


Figure 6.11: Analysis Construction of the Parallel Graph Modification $G \overset{*}{\Rightarrow} H$

Proposition 6.3.13: (existence and uniqueness of tentative merge). *For a given ancestor model G and graph modifications $\delta_1 : G \overset{*}{\Rightarrow} H_1$ and $\delta_2 : G \overset{*}{\Rightarrow} H_2$, there is a parallel construction $G \overset{\delta^t}{\Rightarrow} H$ denoted* tentative merge construction *yielding a unique result called* tentative merge*, where*

1. *$\delta_i = \delta_{i+} + \delta_{i\triangle} + \delta_{i-} + \delta_{i\times}$ for $i \in \{1, 2\}$ is the analysis construction of each parallel graph modification δ_i separating graph modifications in single add, update, delete, and composite operations,*
2. *$\delta^n = \langle \delta^n_+, \delta^n_\triangle, \delta^n_-, \delta^n_\times \rangle$ is the set of all non-conflicting, i.e., parallel independent graph modifications of δ_1 and δ_2, and $\delta^c = \langle \delta^c_+, \delta^c_\triangle, \delta^c_-, \delta^c_\times \rangle$ consists all conflicting graph modifications of δ_1 and δ_2, such that the disjoint union $\delta^n \dot\cup \delta^c = \delta_1 \dot\cup \delta_2$.*
3. *the parallel graph modification $\delta^t = \delta_+ + \delta^n_\triangle + \delta^n_\times$ exists and yields a unique graph, where the number of nodes and edges is maximal.*

Proof Idea. As a state-based comparison reflects effective changes only, changes within one derived graph modification sequence δ_1 or δ_2 do not overlap with one another, i.e., they are parallel independent and (1) follows from Theorem 6.3.9. The analysis construction is depicted in Figure 6.11. (2) follows from Definition 6.3.10, as two direct graph modifications either form a critical pair and are in the set δ^c of conflicting graph modifications, or are in the set δ^n of non-conflicting graph modifications otherwise. The disjoint union $\delta^n \dot\cup \delta^c$ of non-conflicting graph modifications and conflicting graph modifications yields the disjoint union of all graph modifications $\delta_1 \dot\cup \delta_2$. The existence and uniqueness of $G \overset{\delta^t}{\Rightarrow} H$ stated in (3) may be ensured by showing confluence of $\delta^t = \delta_+ + \delta^n_\triangle + \delta^n_\times$. Intuitively, the set of additions, non-conflicting updates, and non-conflicting composite operations is confluent. The maximal number of nodes

and edges in the resulting graph H is given, if there is no graph modification $G \overset{\delta^*}{\Rightarrow} H'$ for $\delta^* \in \delta_1 \dot{\cup} \delta_2$, such that $H \subset H'$. δ^t already includes all additions but no deletions, consequently H is maximal.

In the following, we rely on the existence, maximality, and uniqueness of the tentative merge and head for conflict resolution. Taking basic graph transformation theory discussed in Section 4.2 and Subsection 6.3.1 together with the algebraic tentative merge construction as building blocks, the conflict resolution recommender system is a natural consequence. In the following, we assemble the conflict resolution recommender system with the help of the model versioning example introduced in Chapter 5.

6.3.3 Conflict Resolution Recommendation in Action

Recall our two modelers Harry and Sally working together on an event management system depicted in Figure 5.1. Both modelers work in parallel on the same version of a model. Harry commits his changes first to the common repository and is immediately done. However, when Sally tries to check in her changes, she has to resolve several conflicts. For her convenience, she is supported by AMOR's conflict resolution recommender system. Hence, Sally interacts with the generated tentative merge visualized as conflict diagram (cf. Figure 5.5). Behind the scenes, the conflict resolution recommender system is set up and calculates the tentative merge at which we now draw our attention.

As discussed above, the conflict resolution recommender system is established as graph transformation system. In fact, there is not *the* conflict resolution recommender system for all conflict resolution scenarios, as it is based on twofold production sets. The *individual production set* is automatically constructed based on a specific conflict report and contains rules for building the tentative merge. Additionally, a *global production set* embracing rules for matching and annotating metamodel constraint violations and rules for matching and resolving conflicts is given by a common repository for all individual conflict resolution recommender systems. We now discuss the conflict resolution recommender system Sally is interacting with.

Layer 0. The production set of layer 0 has highest application priority and is thus executed first. This production set is individually assembled based on a specific conflict report to generate the tentative merge for the common ancestor model acting as input model for the conflict resolution recommender system. In Sally's case, the addition of the classes `EventManager` and `SoccerMatch`, the references `manages`, `employedAt`, and `worksAt`, the inheritance relationships `Person` to `Customer` and vice versa, as well as the update of a method's name to `purchase` are considered non-conflicting in the first place. Before constructing the tentative merge, only conflicts due to overlapping changes are detectable. Thus, only two conflicts are reported: the Update/Update conflict of the method `getInfo` concurrently renamed in `getTInfo` and `getTicketInfo` and the overlapping deletion of the attribute `artist` as atomic change performed by Harry and in Sally's composite operation. The rules for the tentative merge construction is solely based on this information.

As discussed in Subsection 6.3.1, non-conflicting, i.e., confluent graph transformations may be executed in arbitrary order yielding the same result. Therefore, to construct the tentative

merge, we first merge all non-conflicting direct atomic graph modifications except deletions. Remember from Chapter 5, that every graph modification is automatically complemented with a change annotation. Hence, deletions are in their atomic form and as part of a composite operation only marked as deleted in terms of an annotation to preserve the information for the user in charge of merging and for the conflict resolution recommender system. As deletions are neglected, additions are always applicable.

Composite operations are specifically treated in the model versioning system AMOR, as their corresponding production may be retrieved a-posteriori from an operation repository based on the direct graph modification (cf. Chapter 3 and [145]). Having the production of composite operations at hand is useful for two reasons: First, the production's application conditions may be evaluated for detecting conflicts. Second, the match of re-applying the production after applying atomic changes may be compared with the direct graph modification extracted from the user's modification. If the original match is equal to the recalculated one, the composite operation is treated as conflict-free and is applied in layer 0. In cases of increased match sizes, the rule for the composite operation is shifted to layer 1.

Thus, the property of confluence is ensured for layer 0, i.e., for the set $\delta^t = \langle \delta_+, \delta^n_\triangle, \delta^n_\times \rangle$ consisting all modifications of type addition denoted by δ_+, all non-conflicting update modifications $\delta^n_\triangle$, and $\delta^n_\times$ all non-conflicting composite operations with unchanged match size. Consequently, the tentative merge construction $G \overset{\delta^t}{\Rightarrow} H$ yields a unique result (cf. Figure 6.10b), which is eventually annotated with conflict markers.

Layer 1. When no rules of layer 0 are applicable any more, i.e., when the tentative merge is created, rules of layer 1 are automatically executed. This layer consists of an individual production set and a global production set. The individual production set contains composite operations with increased match sizes. For example, if Sally performs the composite operation pullUpField while Harry introduces a new subclass of the class `Event` possessing the pulled up attribute, the composite operation would still be applicable to the tentative merge. In such a case, the match size is increased compared to the original application, meaning that other changes, more precisely new elements, are included in the refactoring. While the re-application of the composite operation in such situations may often reflect the intuition of the modelers, it is yet annotated with a warning, as it is sequentially dependent to changes already applied and should therefore attract the modeler's attention.

The global production set of layer 1 holds patterns for matching metamodel constraint violations and producing corresponding annotations. In Sally's case, the inheritance cycle between `Person` and `Customer` is detected and annotated.

Optionally, external constraint checkers taking the tentative merge as input and returning a set of individual productions for annotating violated constraints may be plugged into this layer. In our example, an instantiation check is performed, recognizing that the inverse multiplicities of the new inserted references `employedAt` and `worksAt` cannot be fulfilled.

Layer 2. Layer 2 is built on top of layers 0 and 1 and is responsible for the actual conflict resolution recommendation. The production set of this layer is solely global and retrieved from a common conflict resolution pattern repository. This repository is used by all individual conflict

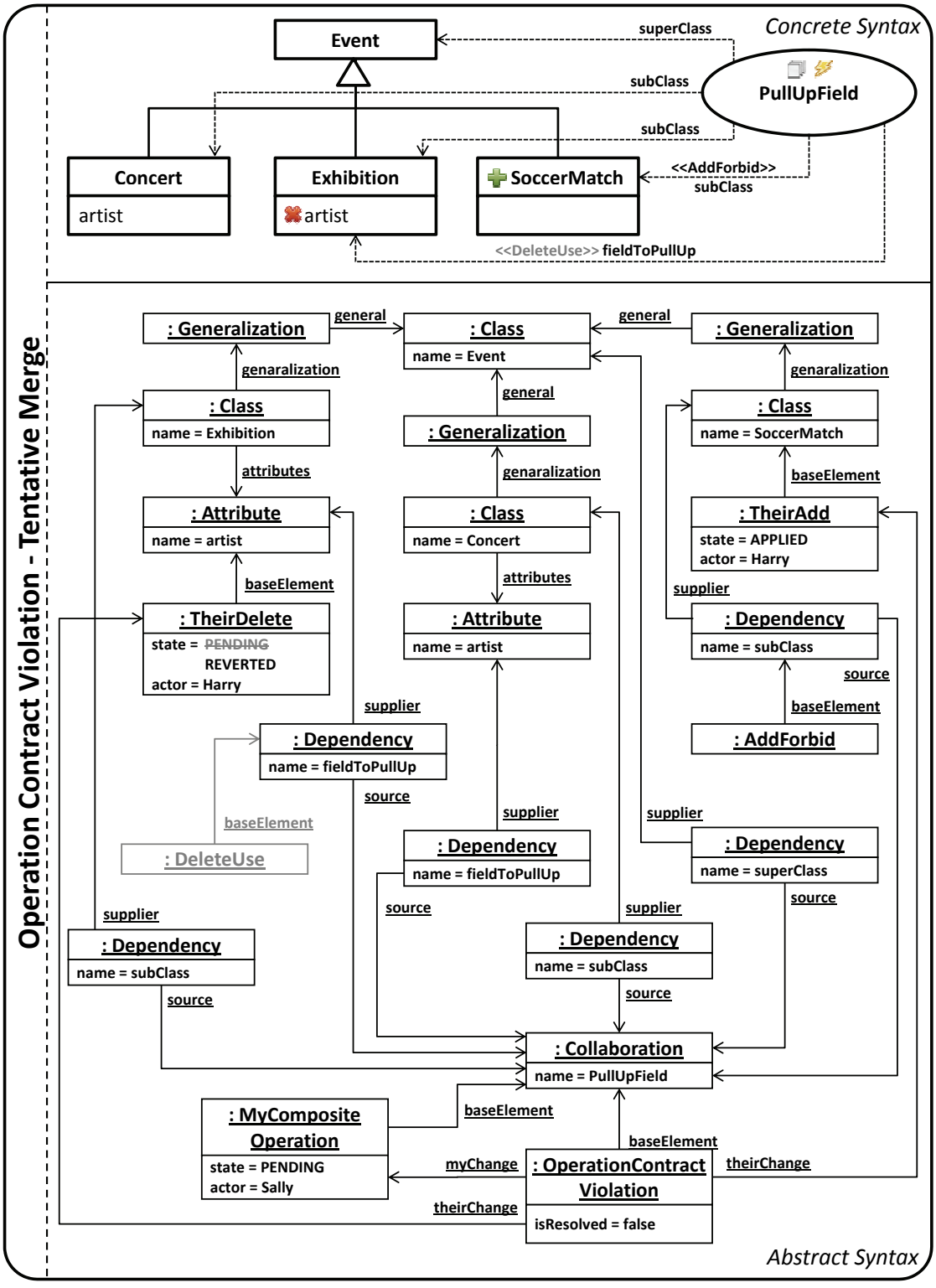


Figure 6.12: Tentative Merge for an Operation Contract Violation

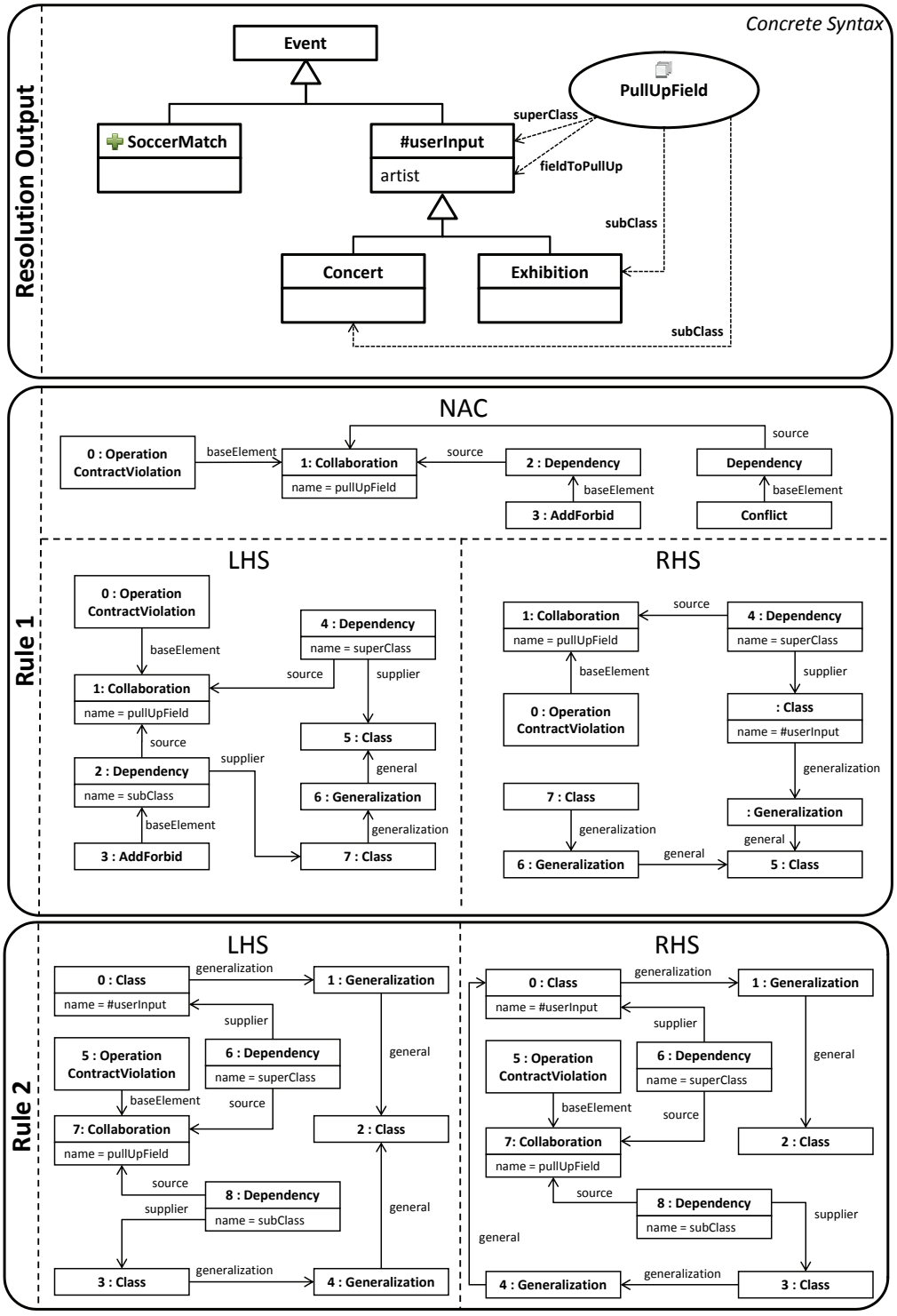


Figure 6.13: Resolution Output for the Add/Forbid Conflict and Rules 1+2

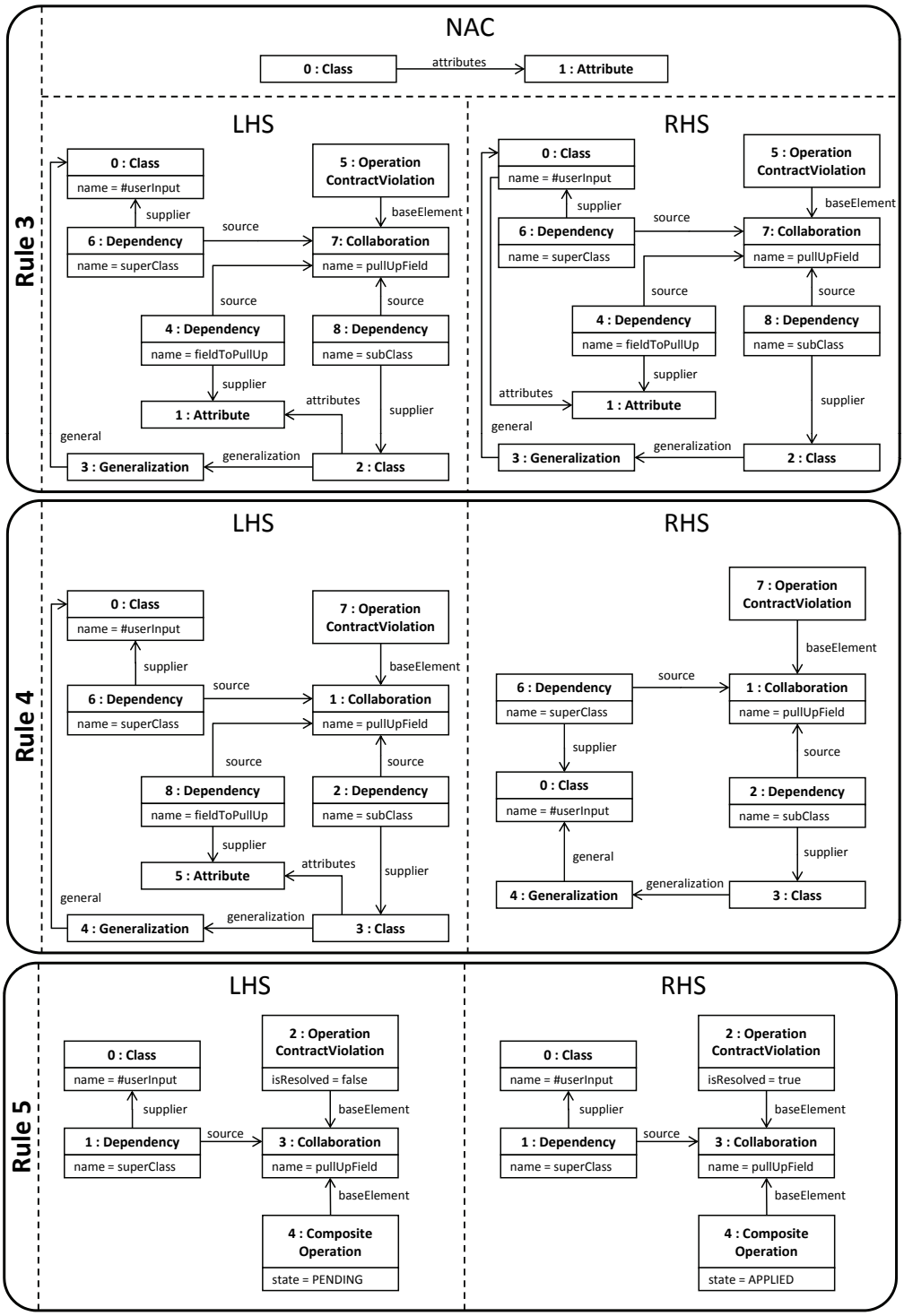


Figure 6.14: Resolution Rules 3-5 for the Add/Forbid Conflict

resolution recommender systems and provides reusable patterns applicable in recurring conflict situations. However, the repository is extensible and new conflict resolution patterns may be stored and immediately used in all conflict resolution recommender systems. In contrast to all other layers where rules are automatically applied without user intervention, productions in layer 2 are selected by the user. Thus, layer 2 reflects the conflict resolution user interface the modeler in charge of merging is interacting with. Starting with the output of layer 1 (cf. Figure 5.5), Sally selects rules for resolving the conflicts from the suggested set. The suggested set contains all applicable productions from the repository.

Sally has to resolve the Update/Update conflict concerning the method `getInfo` for which she gets the suggestions to take either the new name provided by Harry, her new name, or a completely different name, as depicted in Figure 6.4. Further, she has to resolve two violations, namely the metamodel violation caused by the inheritance cycle and the instantiation violation due to the multiplicities of the new references between `Office` and `Employee`. Sally resolves both violations by reverting one of the overlapping changes. Finally, she has to resolve the operation contract violation resulting from her composite change `pullUpField` and Harry's atomic changes add class `SoccerMatch` and delete attribute `artist` in class `Exhibition` taking shape in the Add/Forbid conflict and the Delete/Use conflict, respectively. The part of the tentative merge comprising this conflict is depicted in Figure 6.12. Unfortunately, there is no suggestion for resolving this complex kind of conflict yet. Hence, Sally starts to analyze the problem and browses through the change annotations linked to this conflict. She recognizes, that Harry deleted the attribute `artist`, which she shifted to the common superclass. As she does not agree with Harry's opinion that exhibitions do not have an artist, she reverts his change. The revert results in changing the state value to `REVERTED` in annotation `theirDelete` and the removal of the `DeleteUse` annotation from the corresponding `Dependency`. The changes resulting from this revert are visualized gray in Figure 6.12. The Delete/Use conflict is resolved, but the operation contract violation remains with the Add/Forbid conflict. If Sally would later decide to apply Harry's delete, the rule of layer 0 responsible for annotating the conflict would immediately match. As layer 0 has higher priority than layer 2, the conflict gets again automatically annotated. The Delete/Use conflict is resolved now and a new conflict resolution pattern for resolving the remaining Add/Forbid conflict is found and suggested to Sally (cf. Figure 6.13 and Figure 6.14). The upper part of Figure 6.13 shows the resulting model after applying this conflict resolution pattern. As stated in the negative application condition of rule 1, the conflict resolution pattern found for resolving the Add/Forbid conflict is only applicable if no further conflict is taking place in the operation contract violation. Rule 1 matches the class causing the Add/Forbid conflict and the target class for the pullUpField refactoring together with the corresponding collaboration and annotations. It further introduces a new intermediate layer in the inheritance hierarchy to separate classes exhibiting the common attribute and the new added class without the attribute. The new class is then used as superclass for the refactoring, while the added class of the Add/Forbid conflict is excluded from the refactoring. Rule 2 updates the inheritance relationships for all remaining subclasses to the newly introduced intermediate class. Subsequently, rule 3 performs the actual shift of the attribute to the intermediate class. It is executed only once as the negative application condition ensures that the shifted attribute does not exist in the new super class beforehand. Rule 4 is applied as long as subclasses pos-

sess the shifted attribute and deletes it. Finally, rule 5 updates the tagged values of the conflict stereotypes; in fact, the state of the composite operation annotation is set to `APPLIED` and the operation contract violation is set to resolved. Sally is satisfied with this resolution pattern and confirms that she is finished and that the model should be committed.

Layer 3. Before the model is committed to the repository, a cleansing step is performed in layer 3. All elements marked for deletion, reverted additions, change and conflict annotations, and collaborations are removed from the model. After cleansing, the model is committed.

6.4 Summary and Future Work

Graph modifications are the key resource in state-based model versioning systems as they effectively represent the transition from one state of a model to another. Non-conflicting graph modification sequences derived from two parallel edited models may be automatically merged to one parallel graph modification [213]. If they are conflicting, they may nevertheless be automatically resolved by a special merge construction where updates outpace deletions [80]. However, the information on applied changes and ignored deletions gets lost. In this chapter, we first motivated that explicit change and conflict information renders the recommendation of conflict resolution patterns possible. After a brief overview on graph transformation systems and certain properties they may obey, we presented a Conflict Resolution Recommender System as graph transformation system. The basis for conflict resolution is a tentative merge construction embracing change and conflict annotations. The conflict resolution recommender system actually recommends conflict resolution patterns in terms of graph transformations matching the tentative merge and rewriting the conflicting part such that the conflict gets resolved. Hence, existence, uniqueness, and maximality of the tentative merge are crucial properties, which we showed for our construction.

The presented conflict resolution recommender system effectively exploits change and conflict annotations introduced in Chapter 5 to merge concurrently evolved models in a conflict aware manner and thereby supports the whole merge and conflict resolution process. Combined with the conflict diagram, the conflict resolution recommender system provides visual and interactive assistance in form of reusable patterns. Nevertheless, there is room for improvement; in particular, we focus our future work on the example driven derivation of conflict resolution patterns and on the application of conflict resolution patterns in similar conflict situations instead of exactly fitting ones.

Learning of Conflict Resolution Patterns by Example. The automatic derivation of conflict resolution patterns based on a concrete conflict resolution scenario would significantly improve the usability of populating the pattern repository. Currently, the user in charge of merging selects a recommended conflict resolution pattern or creates a new conflict resolution pattern in terms of graph transformations in cases where no suitable pattern is available. In future work, we plan to ease the specification of conflict resolution patterns by employing a by-demonstration approach as already implemented in EMF Modeling Operations (EMO) for the by-demonstration specification of composite modeling operations [36, 145]. However, there are still two challenges

to solve. First, the EMO environment is designed to be used in isolation. The EMO user usually provides an initial model exactly covering the minimal context necessary as application condition for the specified transformation. If EMO is integrated in the conflict resolution recommender system, the initial model comprises the whole tentative merge and should be tailored to cover the essential part describing the conflict. We plan to leverage information of the conflict report describing the conflict's essence to prune the initial model. The second challenge is to port EMO's by-demonstration approach implemented in Eclipse's modeling framework to the formal basis of algebraic graph transformation. First investigations showed the feasibility to align the by-demonstration approach with the double-pushout graph transformation theory [30, 97]. While the derivation of EMO's operation specification based on differences between the initial model and the revised model may be directly assigned to the minimal rule extraction [24], the manual configuration step is not covered yet. However, an unconfigured rule is tightly bound to the demonstrated transformation and is thus too specific for reuse. We plan to leverage information of the conflict report combined with heuristics to identify generalization potential to free the user in charge of merging from any post-processing and configuration efforts. In this way, the derivation of new conflict resolution patterns is completely hidden from the user.

Similarity Aware Conflict Resolution. Another extension of the conflict resolution recommender system is to expand the applicability of conflict resolution patterns. This widening may be considered in two directions. First, if different modeling languages share similar concepts, similar resolution strategies may be applied in case of conflicts. Thus, conflict resolution patterns defined for one modeling language may be reasonably adapted and applied for another modeling language. Second, conflict resolution patterns may also apply to similar concepts within a modeling language. For example, the restructuring of the inheritance hierarchy defined in the resolution rule for resolving the AddForbid conflict resulting from the parallel applied pullUpField refactoring and the introduction of a new class as depicted in Figure 6.13 and Figure 6.14 would also make sense for solving a similar conflict are not attributes but methods were target of the refactoring. A similarity aware conflict resolution recommender system should then suggest not only exact matching conflict resolution patterns, but also patterns which may be adapted to the similar situation. Those patterns may then be ranked based on their distance to the actual situation. In previous work, we already presented two methods for calculating similarities of elements within one metamodel [32,37]. While we follow in [32] a pure object-oriented approach where we exploit the fine-grained inheritance relationships of the UML Superstructure, we borrow in [37] a technique from information retrieval and calculate similarity measures based on the *term frequency-inverse document frequency* algorithm. In future work, we plan on the one hand to fine-tune the similarity calculation by combining both approaches. On the other hand, we plan to work on a formal framework realizing the actual adaptation step. Our design rationale for this framework is (1) to employ a formal, graph transformation based technique coherently complementing the existing conflict resolution recommender system, and (2) to calculate the similarity measures once and declaratively specify them in terms of relations which are then used for translating the specific transformations. Thus, triple-graph grammars (TGGs) [197] seam to be well suited.

CHAPTER 7

Conclusion

In this thesis, we elaborated on model versioning—a prerequisite for widespread adoption of model-driven engineering (MDE). While current model versioning systems mostly concentrate on comparing models and detecting conflicts in between, we focused our work on conflict resolution support. In the following, we briefly revisit our problem domain and critically reflect our contributions. We then conclude with an outlook on future research directions.

7.1 Contributions

According to Brooks [28], software is inherently complex. This complexity is split into the essential complexity of software, which comes from the problem domain itself and the accidental complexity emerging from using inadequate techniques to represent the problem domain. While the essential complexity is enclosed in the very nature of software, and is thus not reducible, several efforts, such as high-level programming languages, unified programming environments, and object-oriented analysis and design methods including visual modeling languages like the Unified Modeling Language (UML) [114], successfully mitigate accidental complexity [25, 28]. However, the overall complexity of software engineering is not only ascribed to software. As stated by Parnas almost 40 years ago, software engineering denotes the multi-person construction of multi-version software [178], again leading to certain complexity. To cope with the difficulties of the software engineering process, dedicated tools supporting team work and change management evolved [48, 215]. Optimistic version control systems (VCS) aim at tackling the complexity of team-work and the complexity of change management at once. VCSs (1) enable parallel read and write access to an artifact by comparing and merging independently evolved versions stemming from a common ancestor, and (2) track the evolution of those artifacts.

With the increasing complexity of modern software projects, software engineering practice shifts from code-centric paradigms to model-centric, and even model-driven engineering paradigms [20]. Visual modeling languages are more and more employed not only as informal design sketches or for documentation purposes, but as first-class citizen in the whole engineering process, inducing the multi-person construction of multi-version models. Consequently, models

need to be put under version control. Due to the graph-based nature of models, existing VCSs successfully employed for source code are only limited suited for the versioning of models and dedicated model versioning systems are necessary. State-of-the-art model versioning systems already operate on the model's graph structure and are able to detect fine-grained conflicts. However, visual models have a dual representation in form of their abstract syntax used for serialization and their graphical concrete syntax acting as visual interface to the user. Current model versioning systems neglect this fact and present conflicts in the abstract syntax only, introducing again accidental complexity [31]. Even worse, current systems provide hardly any support to resolve detected conflicts and shift the responsibility of merging completely to the user.

This work is embedded in the research project AMOR [32] with the overall goal of realizing generic, yet adaptable model versioning support. The main focus of this work is conflict resolution. In order to improve conflict resolution support in model versioning, we elaborated on three contributions throughout this thesis, which we briefly discuss in the following.

Contribution 1: Conflict Categorization. The prerequisite for improving conflict resolution support is a thorough understanding on merge conflicts. We therefore started our studies with analyzing changes and possible merge conflicts stated in literature as well as from an exhaustive example catalog covering several modeling languages. Besides related works on model merging, the fields of software merging and graph transformation theory influenced our categorization. We found out, that establishing a generic conflict categorization is no straightforward task, as the notions of *change* and *conflict* are heavily overloaded in literature. We identified quite different, yet limited types of conflicts, not reducible to one singe definition. Similar to existing categorizations in the field of software merging, we finally aligned our conflict categorization with the merge process and distinguish two main categories, namely *overlapping changes*, which are detectable by solely analyzing the respective change sets, and *violations*, not becoming evident before the merged model is created.

Contribution 2: Conflict Diagram. As modelers are used to the concrete syntax only, detected conflicts need to be visualized and resolved graphically. Conflict resolution is the complex and intricate task of integrating conflicting changes partly performed by other developers to achieve a valid version reflecting the intentions behind all changes. According to our overall design principles of developing a generic solution, we refrained from creating specific editor extensions for visualizing conflicting models. Instead, we developed a *conflict aware merge strategy*, jointly merging the model's abstract syntax representation and its corresponding concrete syntax, yielding a dedicated *conflict diagram* as tentative starting point for visual merging. The conflict diagram unifies all non-conflicting changes and fosters comprehension on the model's evolution by integrating dedicated annotations in form of a *conflict profile*. The annotation mechanism allows a holistic, model-based representation of changes and conflicts, directly reflecting the conflict categorization established in Contribution 1. We showcased the feasibility of this approach for UML models using UML profiles.

Contribution 3: Conflict Resolution Recommender System. To further improve conflict resolution support, we worked on guiding modelers through conflict resolution in a semi-auto-

matic manner. We observed, that similar structured conflicts are resolved using recurring resolution patterns. Hence, we established a conflict resolution recommender system, which suggests automatically executable conflict resolution patterns. We elaborated on the feasibility of such a system in a formal graph transformation theoretic manner, by showing that the conflict aware construction of the tentative merge is (1) maximal with respect to the parallel performed changes, (2) yields a unique result, and (3) always exists, building thus a perfect basis for the conflict resolution recommender system. Combined with the annotated conflict diagram elaborated in Contribution 2, the conflict resolution recommender system enables the conflict resolution using the concrete graphical syntax.

7.2 Outlook

The presented conflict aware merge strategy together with the conflict profile turned out to be the cornerstone throughout this thesis, as it enables visual conflict resolution in one unified conflict diagram as well as the recommendation of recurring conflict resolution patterns. The contributions of this thesis focused on the feasibility of conflict resolution support in model versioning, which was successfully showcased. Nonetheless, there is room for improvement.

Usability. Visual conflict resolution does not automatically guarantee usability. Especially when merging large diagrams manifesting many conflicts, the usability of the conflict resolution user interface is crucial to not overwhelm the modeler in charge of merging. Conceivable are, e.g., special filtering or zooming mechanisms allowing to concentrate on certain conflicts by hiding all others. Further, animation techniques may be used to replay the model's evolution.

Learning of conflict resolution patterns. Currently, the conflict resolution recommender system suggests a predefined set of conflict resolution patterns. In conflict cases where no suggestion is available, this set may be extended by manually specifying new rules. To ease the specification of conflict resolution patterns, techniques from the field of by-demonstration model transformation may be employed to derive patterns from a specific conflict resolution.

Similarity aware conflict resolution. Heuristics and similarity measures may be elaborated to extend the applicability of stored conflict resolution patterns. Then, the conflict's structure for which a conflict resolution pattern is defined may be relaxed or adapted. Further, conflict resolution patterns may be transformed to resolve conflicts occurred in different modeling languages.

List of Figures

Bibliography

[1] Aditya Agrawal. Graph Rewriting and Transformation (GReAT): A Solution for the Model Integrated Computing (MIC) Bottleneck. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ASE'2003, pages 364–368, 2003.

[2] Aditya Agrawal, Gabor Karsai, and Akos Ledeczi. An End-to-End Domain-Driven Software Development Framework. In *Companion of the 18th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, OOPSLA'03, pages 8–15. ACM, 2003.

[3] Marcus Alanen and Ivan Porres. Difference and Union of Models. In *Proceedings of UML 2003 Conference*, pages 2–17. Springer, 2003.

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

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

[6] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Model Driven Architecture – Foundations and Applications*, volume 4066 of *LNSC*, pages 361–375. Springer, 2006.

[7] AMOR Project. http://www.modelversioning.org. Accessed: 2011-12-20.

[8] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. JDiff: A Differencing Technique and Tool for Object-oriented Programs. *Automated Software Engineering*, 14(1):3–36, 2007.

[9] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.

[10] B Ashok, Joseph Joy, Hongkang Liang, Sriram K. Rajamani, Gopal Srinivasa, and Vipindeep Vangala. DebugAdvisor: A Recommender System for Debugging. In *Proceedings of the the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE'09, pages 373–382. ACM, 2009.

[11] Colin Atkinson and Thomas Kühne. A Tour of Language Customization Concepts. *Advances in Computers*, 70:105–161, 2007.

[12] Daniel Balasubramanian, Anantha Narayanan, Chris van Buskirk, and Gabor Karsai. The Graph Rewriting and Transformation Language: GReAT. *Electronic Communications of the EASST*, 1, 2006.

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

[14] Roswitha Bardohl, Mark Minas, Gabriele Taentzer, and Andy Schürr. Application of Graph Transformation to Visual Languages. Part I: Visual and Object-Oriented Languages. In Ehrig et al. [79], chapter 3, pages 105–180.

[15] Stephen Barrett, Patrice Chalin, and Greg Butler. Model Merging Falls Short of Software Engineering Needs. In *Proceedings of the 2nd Workshop on Model-Driven Software Evolution @ MoDELS'08*, 2008.

[16] Kent Beck and Martin Fowler. Bad Smells in Code. In *Refactoring: Improving the Design of Existing Code* [94], chapter 3.

[17] Lars Bendix and Pär Emanuelsson. Collaborative Work with Software Models–Industrial Experience and Requirements. In *Proceedings of the 2nd International Conference on Model Based Systems Engineering*, pages 2–6. IEEE, 2009.

[18] TU Berlin. EMF Tiger. `http://user.cs.tu-berlin.de/~emftrans/`. Accessed: 2012-01-16.

[19] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. What Makes a Good Bug Report? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT'08/FSE-16, pages 308–318, USA, 2008. ACM.

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

[21] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frederic Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations? Transformation Models! In *Model Driven Engineering Languages and Systems (MoDELS'06)*, volume 4199 of *LNCS*, pages 440–453. Springer, 2006.

[22] Jean Bézivin and Olivier Gerbé. Towards a Precise Definition of the OMG/MDA Framework. In *Proceedings of the 16th Annual International Conference on Automated Software Engineering, 2001*, ASE'2001, pages 273–280, 2001.

[23] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. Formal Foundation of Consistent EMF Model Transformations by Algebraic Graph Transformation. *Software and Systems Modeling*, pages 1–24, 2011. Online First.

[24] Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Verification of Architectural Refactorings: Rule Extraction and Tool Support. *Electronic Communications of the EASST*, 16, 2009.

[25] Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. Complexity. In *Object-Oriented Analysis and Design with Applications* [26], chapter 1.

[26] Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional, 2nd edition, 2007.

[27] Wilfried Brauer, Grzegorz Rozenberg, and Arto Salomaa, editors. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[28] Frederick P. Brooks, Jr. No Silver Bullet—Essence and Accidents of Software Engineering. *Computer*, 20(4):10–19, 1987.

[29] Petra Brosch, Uwe Egly, Sebastian Gabmeyer, Gerti Kappel, Martina Seidl, Hans Tompits, Magdalena Widl, and Manuel Wimmer. Towards Semantics-Aware Merge Support in Optimistic Model Versioning. In *Models in Software Engineering - Workshops and Symposia at MODELS 2011, Reports and Revised Selected Papers*, volume 7167 of *LNCS*, pages 246–256. Springer, 2012.

[30] Petra Brosch, Sebastian Gabmeyer, Martina Seidl, and Gerti Kappel. On Formalizing EMF Modeling Operations with Graph Transformations. In *5th International Workshop UML and Formal Methods @ FM2012*, to be published.

[31] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. The Past, Present, and Future of Model Versioning. In Rech and Bunse [184], chapter 15, pages 410–443.

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

[33] Petra Brosch, Horst Kargl, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Gerti Kappel. Conflicts as First-Class Entities: A UML Profile for Model Versioning. In *Models in Software Engineering - Workshops and Symposia at MODELS 2010,*

Reports and Revised Selected Papers, volume 6627 of *LNCS*, pages 184–193. Springer, 2011.

[34] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. Colex: A Web-based Collaborative Conflict Lexicon. In *Proceedings of the 1st International Workshop on Model Comparison in Practice @ TOOLS 2010*, pages 42–49. ACM, 2010.

[35] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Gerti 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)*, pages 129–144. Springer, 2010.

[36] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In *Model Driven Engineering Languages and Systems (MoDELS'09)*, volume 5795 of *LNCS*, pages 271–285. Springer, 2009.

[37] Petra Brosch, Martina Seidl, and Gerti Kappel. A Recommender for Conflict Resolution Support in Optimistic Model Versioning. In *Proceedings of the International Conference Companion on Object Oriented Programming Systems Languages and Applications*, Onward! @ SPLASH'10, pages 43–50. ACM, 2010.

[38] Petra Brosch, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Philip 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, pages 207–214. Springer, 2009.

[39] Petra Brosch, Konrad Wieland, and Gerti Kappel. Conflict Resolution in Model Versioning. In *1st International Master Class on Model-Driven Engineering, Poster Session Companion*, pages 17–18, 2010.

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

[41] Robin D. Burke, Kristian J. Hammond, and Benjamin C. Young. Knowledge-Based Navigation of Complex Information Spaces. In *Proceedings of the thirteenth National Conference on Artificial intelligence - Volume 1*, AAAI'96, pages 462–468. AAAI Press, 1996.

[42] Sven Burmester, Holger Giese, Jörg Niere, Matthias Tichy, Jörg P. Wadsack, Robert Wagner, Lothar Wendehals, and Albert Zündorf. Tool Integration at the Meta-Model Level: The Fujaba Approach. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3):203–218, 2004.

[43] Scott Chacon. Git. `http://git-scm.com/`. Accessed: 2011-12-05.

[44] Noam Chomsky. On Certain Formal Properties of Grammars. *Information and Control*, 2(2):137–167, 1959.

[45] Antonio Cicchetti. *Difference Representation and Conflict Management in Model-Driven Engineering*. PhD thesis, Univeritá dell'Aquilla, 2008.

[46] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. Managing Model Conflicts in Distributed Development. In *Model Driven Engineering Languages and Systems (MoDELS'08)*, volume 5301 of *LNCS*, pages 311–325. Springer, 2010.

[47] Michael K. Coleman and D. Stott Parker. Aesthetics-based Graph Layout for Human Consumption. *Software: Practice and Experience*, 26(12):1415–1438, 1996.

[48] Reidar Conradi and Bernhard Westfechtel. Version Models for Software Configuration Management. *ACM Computing Surveys*, 30(2):232–282, 1998.

[49] World Wide Web Consortium. Extensible Markup Language (XML) 1.0 (Fifth Edition). `http://www.w3.org/TR/xml/`, November 2008.

[50] Thomas A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[51] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic Approaches to Graph Transformation. Part I: Basic Concepts and Double Pushout Approach. In Rozenberg [190], chapter 3, pages 163–245.

[52] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

[53] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. ADAMS: Advanced Artefact Management System. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, CSMR'06, pages 349–350. IEEE, 2006.

[54] Andrea De Lucia, Fausto Fasano, Giuseppe Scanniello, and Genoveffa Tortora. Concurrent Fine-Grained Versioning of UML Models. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*, CSMR'09, pages 89–98. IEEE, 2009.

[55] Ernst Denert, Reinhold Franck, and Wolfgang Streng. PLAN2D - Towards a Two-Dimensional Programming Language. In *GI - 4. Jahrestagung*, pages 202–213, London, UK, 1975. Springer.

[56] Prasun Dewan and Rajesh Hegde. Semi-Synchronous Conflict Detection and Resolution in Asynchronous Software Development. In *Proceedings of the 10th European Conference on Computer-Supported Cooperative Work (ECSCW'2007)*, pages 159–178. Springer, 2007.

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

[58] F. Drewes, H.-J. Kreowski, and A. Habel. Hyperedge Replacement Graph Grammars. In Rozenberg [190], chapter 2, pages 95–162.

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

[60] Eclipse. Eclipse Modeling Framework Project (EMF). `http://www.eclipse.org/modeling/emf`. Accessed: 2011-11-04.

[61] Eclipse. Eclipse Modeling Framework Technology. `http://www.eclipse.org/modeling/emft`. Accessed: 2011-12-05.

[62] Eclipse. EMF Compare. `http://www.eclipse.org/emf/compare/#compare`. Accessed: 2011-12-05.

[63] Eclipse. EMF UML2. `http://www.eclipse.org/modeling/mdt/?project=uml2`. Accessed: 2011-12-05.

[64] Eclipse. Henshin. `http://www.eclipse.org/modeling/emft/henshin/`. Accessed: 2012-01-16.

[65] Eclipse. M2M/Operational QVT Language (QVTO). `http://wiki.eclipse.org/M2M/Operational_QVT_Language_%28QVTO%29`. Accessed: 2011-11-01.

[66] Eclipse. M2M/QVT Declarative (QVTd). `http://wiki.eclipse.org/M2M/QVT_Declarative_%28QVTd%29`. Accessed: 2011-11-01.

[67] Eclipse. Xtext. `http://www.eclipse.org/Xtext/`. Accessed: 2011-12-05.

[68] Alexander Egyed. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Transactions on Software Engineering*, 37(2):188–204, March-April 2011.

[69] Hartmut Ehrig and Karsten Ehrig. Overview of Formal Concepts for Model Transformations Based on Typed Attributed Graph Transformation. *Electronic Notes in Theoretical Computer Science*, 152:3–22, 2006.

[70] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. A Short Introduction to Category Theory. Appendix A. In Brauer et al. [27], pages 329–352.

[71] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Adhesive High-Level Replacement Systems. Part II: Adhesive High-Level Replacement Categories and Systems. In Brauer et al. [27], chapter 5, pages 101–124.

[72] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Case Study on Model Transformation. Part IV: Case Study on Model Transformation, and Tool Support by AGG. In Brauer et al. [27], chapter 14, pages 287–304.

[73] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. General Introduction. Part I: Introduction to Graph Transformation Systems. In Brauer et al. [27], chapter 1, pages 5–20.

[74] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Graph Transformation Systems. Part I: Introduction to Graph Transformation Systems. In Brauer et al. [27], chapter 3, pages 37–75.

[75] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Graphs, Typed Graphs, and the Gluing Construction. Part I: Introduction to Graph Transformation Systems. In Brauer et al. [27], chapter 2, pages 21–35.

[76] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Typed Attributed Graph Transformation Systems. Part III: Typed Attributed Graph Transformation Systems. In Brauer et al. [27], chapter 9, pages 181–205.

[77] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Typed Attributed Graph Transformation with Inheritance. Part III: Typed Attributed Graph Transformation Systems. In Brauer et al. [27], chapter 13, pages 259–285.

[78] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. Typed Attributed Graphs. Part III: Typed Attributed Graph Transformation Systems. In Brauer et al. [27], chapter 8, pages 171–180.

[79] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 2. Applications, Languages and Tools*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, October 1999.

[80] Hartmut Ehrig, Claudia Ermel, and Gabriele Taentzer. A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications. In *Fundamental Approaches to Software Engineering (FASE'11)*, volume 6603 of *LNCS*, pages 202–216. Springer, 2011.

[81] Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic Approaches to Graph Transformation. Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In Rozenberg [190], chapter 4, pages 247–312.

[82] Hartmut Ehrig, Martin Korff, and Michael Löwe. Tutorial Introduction to the Algebraic Approach of Graph Grammars based on Double and Single Pushouts. In *Graph Grammars and Their Application to Computer Science*, volume 532 of *LNCS*, pages 24–37. Springer, 1991.

[83] Hartmut Ehrig, Michael Pfender, and Hans Jürgen Schneider. Graph-Grammars: An Algebraic Approach. In *Proceedings of the 14th Annual Symposium on Switching and Automata Theory*, SWAT'1973, pages 167–180, Washington, DC, USA, 1973. IEEE Computer Society.

[84] Karsten Ehrig, Esther Guerra, Juan De Lara, Laszlo Lengyel, Tihamer Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model Transformations by Graph Transformations: A Comparative Study. In *International Workshop on Model Transformations in Practice*, MTiP @ MoDELS 2005, page 48, 2005.

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

[86] Joost Engelfriet and Grzegorz Rozenberg. Node Replacement Graph Grammars. In Rozenberg [190], chapter 1, pages 1–94.

[87] Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. The AGG Approach: Language and Environment. In Ehrig et al. [79], chapter 14, pages 551–603.

[88] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington, 2000.

[89] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically Discovering Likely Program Invariants to Support Program Evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE'99, pages 213–224, USA, 1999. ACM.

[90] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy 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.

[91] Anthony C. W. Finkelstein, Dov Gabbay, Anthony Hunter, Jeff Kramer, and Bashar Nuseibeh. Inconsistency Handling in Multiperspective Specifications. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

[92] International Organization for Standardization and International Electrotechnical Comission. Information Technology—Syntactic Metalanguage—Extended BNF 1.0. `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`, December 1996.

[93] Apache Software Foundation. Apache Subversion. `http://subversion.apache.org/`. Accessed: 2011-12-05.

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

[95] Robert France and Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Proceedings of Future of Software Engineering*, FOSE'07, pages 37–54. IEEE Computer Society, 2007.

[96] Sebastian Gabmeyer. Formalization of the Operation Recorder. In *Formalization of the Operation Recorder based on Graph Transformation Theory* [97], chapter 4, pages 59–80.

[97] Sebastian Gabmeyer. Formalization of the Operation Recorder based on Graph Transformation Theory. Master's thesis, Vienna University of Technology, 2011.

[98] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Professional, 1995.

[99] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2nd edition, 2002.

[100] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. Management of Software Engineering. In *Fundamentals of Software Engineering* [99], chapter 8.

[101] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. Software Engineering: A Preview. In *Fundamentals of Software Engineering* [99], chapter 1.

[102] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. Software Engineering Tools and Environments. In *Fundamentals of Software Engineering* [99], chapter 9.

[103] David Goldberg, David Nichols, Brian M. Oki, and Douglas Terry. Using Collaborative Filtering to Weave an Information Tapestry. *Communications of the ACM*, 35(12):61–70, 1992.

[104] Joel Greenyer and Ekkart Kindler. Comparing Relational Model Transformation Technologies: Implementing Query/View/Transformation with Triple Graph Grammars. *Software and Systems Modeling*, 9(1):21–46, 2010.

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

[106] Iris Groher and Alexander Egyed. Selective and Consistent Undoing of Model Changes. In *Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6395 of *LNCS*, pages 123–137. Springer, 2010.

[107] Fujaba Core Development Group. Fujaba. `http://fujaba.de/`. Accessed: 2011-12-05.

[108] Object Management Group. Common Warehouse Metamodel (CWM) Specification V1.1. `http://www.omg.org/spec/CWM/1.1/`, March 2003.

[109] Object Management Group. Model Driven Architecture (MDA) Guide V1.0.1. http://www.omg.org/mda/, January 2006.

[110] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation V1.1. http://www.omg.org/spec/QVT/1.1/, January 2011.

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

[112] Object Management Group. OMG MOF 2 XMI Mapping Specification V2.4.1. http://www.omg.org/spec/XMI/2.4.1, August 2011.

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

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

[115] Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, 26(3-4):287–313, June 1996.

[116] David Harel and Bernhard Rumpe. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer*, 37(10):64–72, 2004.

[117] Reiko Heckel. Graph Transformation in a Nutshell. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 148(1):187–198, 2006.

[118] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate Structural Context Matching: An Approach to Recommend Relevant Examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006.

[119] Gérard Huet. Confluent Reductions: Abstract Properties and Applications to Term Rewriting Systems: Abstract Properties and Applications to Term Rewriting Systems. *Journal of the ACM*, 27(4):797–821, October 1980.

[120] Oliver Hummel, Werner Janjic, and Colin Atkinson. Code Conjurer: Pulling Reusable Software out of Thin Air. *IEEE Software*, 25(5):45 –52, 2008.

[121] James W. Hunt and Malcolm Douglas McIllroy. An Algorithm for Differential File Comparison. Technical report, AT&T Bell Laboratories Inc., 1976.

[122] IBM. Comparing and merging UML models in IBM Rational Software Architect. http://www.ibm.com/developerworks/rational/library/05/712_comp/index.html. Accessed: 2011-12-05.

[123] IBM Rational Software Architect. http://www.ibm.com/developerworks/rational/products/rsm. Accessed: 2011-07-25.

[124] ikv++ technologies ag. medini QVT 1.7.0. `http://projects.ikv.de/qvt`. Accessed: 2011-11-01.

[125] Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. Collaborative Recommendation. In *Recommender Systems—An Introduction* [129], chapter 2, pages 13–50.

[126] Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. Content-based Recommendation. Part I: Introduction to Basic Concepts. In *Recommender Systems—An Introduction* [129], chapter 3, pages 51–80.

[127] Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. Hybrid Recommendation Approaches. In *Recommender Systems—An Introduction* [129], chapter 5, pages 124–142.

[128] Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. Knowledge-based Recommendation. In *Recommender Systems—An Introduction* [129], chapter 4, pages 81–123.

[129] Dietmar Jannach, Markus Zanker, Alexander Felfernig, and Gerhard Friedrich. *Recommender Systems—An Introduction*. Cambridge University Press, 2010.

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

[131] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 128–138. Springer, 2006.

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

[133] Josef Kallrath, editor. *Modeling Languages in Mathematical Optimization*. Applied Optimization. Springer, 2004.

[134] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David Notkin. Automated Support for Program Refactoring using Invariants. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM'01, pages 736–743, USA, 2001. IEEE Computer Society.

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

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

[137] Sunghun Kim, Kai Pan, and E. E. James Whitehead, Jr. Memories of Bug Fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT'06/FSE-14, pages 35–45, USA, 2006. ACM.

[138] Ekkart Kindler and Robert Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical report, Department of Computer Science, University of Paderborn, 2007.

[139] Peter Klein, Manfred Nagl, and Andy Schürr. IPSEN Tools. Part III: Applications to Software Engineering. In Ehrig et al. [79], chapter 5, pages 105–180.

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

[141] Joseph A. Konstan. Foreword. In *Recommender Systems—An Introduction* [129], pages ix–xii.

[142] Thomas Kühne. Matters of (Meta-) Modeling. *Software and Systems Modeling*, 5:369–385, 2006.

[143] Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Conflict Detection for Graph Transformation with Negative Application Conditions. In *Graph Transformations (ICGT'06)*, volume 4178 of *LNCS*, pages 61–76. Springer, 2006.

[144] Leen Lambers, Hartmut Ehrig, and Fernando Orejas. Efficient Detection of Conflicts in Graph-based Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:97–109, March 2006.

[145] Philip Langer. *Adaptable Model Versioning based on Model Transformation By Demonstration*. PhD thesis, Vienna University of Technology, 2011.

[146] Philip Langer, Konrad Wieland, Manuel Wimmer, and Jordi Cabot. From UML Profiles to EMF Profiles and Beyond. In *Objects, Models, Components, Patterns (TOOLS'11)*, volume 6705 of *LNCS*, pages 52–67. Springer, 2011.

[147] Akos Ledeczi, Miklos Maroti, Gabor Karsai, and Greg Nordstrom. Metaprogrammable Toolkit for Model-Integrated Computing. In *Proceedings of the IEEE Conference and Workshop on Engineering of Computer-Based Systems*, ECBS'99, pages 311–317, mar 1999.

[148] Martin Lefering and Andy Schürr. Specification of Integration Tools. Chapter 3: Internal Conceptual Modeling: Graph Grammar Specifications. In Nagl [167], chapter 3.4, pages 324–334.

[149] Ernst Lippe and Norbert van Oosterom. Operation-Based Merging. In *Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environment*, SDE'05, pages 78–87. ACM, 1992.

[150] Roberto Lopez-Herrejon and Alexander Egyed. Detecting Inconsistencies in Multi-View Models with Variability. In *Modelling Foundations and Applications (ECMFA'10)*, volume 6138 of *LNCS*, pages 217–232. Springer, 2010.

[151] Michael Löwe. Algebraic Approach to Single-Pushout Graph Transformation. *Theoretical Computer Science*, 109(1-2):181–224, March 1993.

[152] Frédéric Madiot and Grégoire Dupé. EMF Facet: A Non-Intrusive Tooling to Extend Metemodels. `http://www.eclipse.org/modeling/emft/facet/`, 11 2010.

[153] Miroslaw Malek. The Art of Creating Models and Models Integration. In *Model-Based Software and Data Integration*, volume 8 of *Communications in Computer and Information Science*, pages 1–7. Springer, 2008.

[154] Thomas W. Malone, Kenneth R. Grant, Franklyn A. Turbak, Stephen A. Brobst, and Michael D. Cohen. Intelligent Information-Sharing Systems. *Communications of the ACM*, 30(5):390–402, 1987.

[155] Jean-Pierre Marquis. Category Theory. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Center for the Study of Language and Information, Stanford University, Stanford, CA, USA, spring 2011 edition, 2011.

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

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

[158] Tom Mens. Introduction and Roadmap: History and Challenges of Software Evolution. In Mens and Demeyer [159], chapter 1.

[159] Tom Mens and Serge Demeyer, editors. *Software Evolution*. Springer, 2008.

[160] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

[161] Tom Mens and Ragnhild Van Der Straeten. Incremental Resolution of Model Inconsistencies. In *Recent Trends in Algebraic Development Techniques*, volume 4409 of *LNCS*, pages 111–126. Springer, 2007.

[162] Mark Minas and Hans Schneider. Graph Transformation by Computational Category Theory. In *Graph Transformations and Model-Driven Engineering*, volume 5765 of *LNCS*, pages 33–58. Springer, 2010.

[163] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. Layout Adjustment and the Mental Map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[164] Jonathan P. Munson and Prasun 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.

[165] Leonardo Murta, Chessman Corrêa, João Gustavo Prudêncio, and Cláudia 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 @ ICSE'08*, pages 25–30. ACM, 2008.

[166] John Mylopoulos. On the Relation of Graph Grammars and Graph Automata. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory*, SWAT'1972, pages 108–120, Washington, DC, USA, 1972. IEEE Computer Society.

[167] Manfred Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *LNCS*. Springer, Berlin, Germany, September 1996.

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

[169] Natalya F. Noy and Mark A. Musen. The PROMPT Suite: Interactive Tools for Ontology Merging and Mapping. *International Journal of Human-Computer Studies*, 59(6):983 – 1024, 2003.

[170] Bashar Nuseibeh, Steve M. Easterbrook, and Alessandra Russo. Making Inconsistency Respectable in Software Development. *Journal of Systems and Software*, 58(2):171–180, 2001.

[171] Object Management Group. Diagram Definition (DD). `http://www.omg.org/spec/DD/1.0/Beta2/`. Accessed: 2012-02-21.

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

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

[174] University of Paderborn. TGG-Interpreter. `http://www.cs.uni-paderborn.de/fachgebiete/fachgebiet-softwaretechnik/forschung/projekte/tgg-interpreter.html`. Accessed: 2012-01-24.

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

[176] Hamilton Oliveira, Leonardo Murta, and Cláudia Werner. Odyssey-VCS: A Flexible Version Control System for UML Model Elements. In *Proceedings of the 12th International Workshop on Software Configuration Management*, SCM'05, pages 1–16. ACM, 2005.

[177] Oracle. Java Metadata Interface (JMI). http://java.sun.com/products/jmi/, June 2002.

[178] David Parnas. Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs. In *Programming Methodology*, volume 23 of *LNCS*, pages 225–235. Springer, 1975.

[179] John L. Pfaltz and Azriel Rosenfeld. Web Grammars. In *Proceedings of the 1st international joint conference on Artificial intelligence*, pages 609–619, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[180] Stephen Pollock. A Rule-Based Message Filtering System. *ACM Transactions on Information Systems*, 6(3):232–254, 1988.

[181] Terrence W. Pratt. Pair Grammars, Graph Languages and String-to-Graph Translations. *Journal of Computer and System Sciences*, 5(6):560–595, 1971.

[182] GNU Project. Bazaar. http://bazaar.canonical.com/. Accessed: 2011-12-05.

[183] GNU Project. Concurrent Versions System. http://cvs.nongnu.org/. Accessed: 2011-12-05.

[184] Jörg Rech and Christian Bunse, editors. *Emerging Technologies for the Evolution and Maintenance of Software Models*. IGI Global, 2011.

[185] Alexander Reder and Alexander Egyed. Model/Analyzer: A Tool for Detecting, Visualizing and Fixing Design Errors in UML. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE'10, pages 347–348. ACM, 2010.

[186] Thomas Reiter, Kerstin Altmanninger, Alexander Bergmayr, Wieland Schwinger, and Gabriele Kotsis. Models in Conflict – Detection of Semantic Conflicts in Model-based Development. In *Proceedings of 3rd International Workshop on Model-Driven Enterprise Information Systems @ ICEIS'07*, pages 29–40, 2007.

[187] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In *Proceedings of the 1994 ACM Conference on Computer Supported Cooperative Work*, CSCW'94, pages 175–186, USA, 1994. ACM.

[188] Paul Resnick and Hal R. Varian. Recommender Systems. *Communications of the ACM*, 40(3):56–58, 1997.

[189] Martin P. Robillard, Robert J. Walker, and Thomas Zimmermann. Recommendation Systems for Software Engineering. *IEEE Software*, 27(4):80–86, 2010.

[190] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[191] Zachary M. Saul, Vladimir Filkov, Premkumar Devanbu, and Christian Bird. Recommending Random Walks. In *Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC-FSE'07, pages 15–24, USA, 2007. ACM.

[192] Hermann Schichl. Models and History of Modeling. In Kallrath [133], chapter 2, pages 25–36.

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

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

[195] Hans Jürgen Schneider. Categorical Notions. In *Graph Transformations* [196], pages 17–58.

[196] Hans Jürgen Schneider. *Graph Transformations*. Universität Erlangen-Nürnberg, 2010.

[197] Andy Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 151–163. Springer, 1995.

[198] Andy Schürr. Programmed Graph Replacement Systems. In Rozenberg [190], chapter 7, pages 479–546.

[199] Andy Schürr, Andreas J. Winter, and Albert Zündorf. The PROGRES Approach: Language and Environment. In Ehrig et al. [79], chapter 13, pages 487–550.

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

[201] Ed Seidewitz. What Models Mean. *Software, IEEE*, 20(5):26–32, 2003.

[202] Bran Selic. The Pragmatics of Model-driven Development. *Software, IEEE*, 20(5):19–25, 2003.

[203] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20:42–45, 2003.

[204] Upendra Shardanand and Pattie Maes. Social Information Filtering: Algorithms for Automating "Word of Mouth". In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI'95, pages 210–217, USA, 1995. ACM Press/Addison-Wesley Publishing Co.

[205] David Canfield Smith, Allen Cypher, and Jim Spohrer. KidSim: Programming Agents without a Programming Language. *Communications of the ACM*, 37(7):54–67, July 1994.

[206] Jonathan Sprinkle. Model-integrated Computing. *IEEE Potentials*, 23(1):28–30, 2004.

[207] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973.

[208] Ragnhild Van Der Straeten, Tom Mens, and Stefan Van Baelen. Challenges in Model-Driven Software Engineering. In *Models in Software Engineering (MoDELS'08)*, volume 6002 of *LNCS*, pages 35–47. Springer, 2009.

[209] Ivan Edward Sutherland. *Sketchpad: A Man-Machine Graphical Communication System*. PhD thesis, Massachusetts Institute of Technology, 1963.

[210] Eugene Syriani and Hans Vangheluwe. De-/Re-constructing Model Transformation Languages. *Electronic Communications of the EASST*, 29, 2010.

[211] Gabriele Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, Technische Universität Berlin, 1996.

[212] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE'2003)*, volume 3062 of *LNCS*, pages 446–453. Springer, 2004.

[213] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. Conflict Detection for Model Versioning Based on Graph Modifications. In *Graph Transformations (ICGT'10)*, volume 6372 of *LNCS*, pages 171–186. Springer, 2010.

[214] France Telecom. SmartQVT. `http://sourceforge.net/projects/smartqvt/`. Accessed: 2011-11-01.

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

[216] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In *Model Driven Architecture - Foundations and Applications (ECMFA-FA'09)*, volume 5562 of *LNCS*, pages 18–33. Springer, 2009.

[217] Tom Tourwé and Tom Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering 2003*, pages 91–100, USA, 2003. IEEE Computer Society.

[218] Javier Troya and Antonio Vallecillo. A Rewriting Logic Semantics for ATL. *Journal of Object Technology*, 10(5):1–29, 2011.

[219] TUM. UNICASE. `http://www.unicase.org/`. Accessed: 2011-12-05.

[220] Davor Čubranić, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A Project Memory for Software Development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.

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

[222] Bernhard Westfechtel. A Formal Approach to Three-way Merging of EMF Models. In *Proceedings of the 1st International Workshop on Model Comparison in Practice @ TOOLS'10*, pages 31–41. ACM, 2010.

[223] Konrad Wieland. *Conflict-tolerant Model Versioning*. PhD thesis, Vienna University of Technology, 2011.

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

[225] Yunwen Ye and Gerhard Fischer. Reuse-Conducive Development Environments. *Automated Software Engineering*, 12(2):199–235, 2005.

[226] Annie T. T. Ying, Gail C. Murphy, Raymond Ng, and Mark C. Chu-Carroll. Predicting Source Code Changes by Mining Change History. *IEEE Transactions on Software Engineering*, 30(9):574–586, 2004.

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

[228] Thomas Zimmermann, Peter Weissgerber, Stephan Diehl, and Andreas Zeller. Mining Version Histories to Guide Software Changes. *IEEE Transactions of Software Engineering*, 31(6):429–445, 2005.

APPENDIX A

Curriculum Vitae

Petra Brosch

Education

since 2007 **PhD Studies**, *Vienna University of Technology*, Vienna.
Business Informatics

2001–2006 **MSc Studies**, *Vienna University of Technology*, Vienna.
Business Informatics

May 2001 **Graduation from Secondary School**, *Commercial Academy*, Baden.

Master thesis

title *Ubiquitäre Web-Anwendungen:*
Realisierung von Adaptierung mit Hilfe aspektorientierter Programmierung

supervisors O.Univ.-Prof. Dr. Gerti Kappel, Vienna University of Technology
Dr. Andrea Schauerhuber, Vienna University of Technology

Experience

since 2008 **University Assistant**, *Vienna University of Technology*, Vienna.
full-time

2006–2008 **Project Assistant**, *Vienna University of Technology*, Vienna.
full-time

2005–2006 **Student Assistant**, *Vienna University of Technology*, Vienna.
part-time

2004 **Tutor**, *Vienna University of Technology*, Vienna.
part-time

2003 **Software development**, *Dohnal Solutions GmbH*, Vienna.
summer internship

2001–2005 **IT consulting and software development**.
freelancer

2000–2005 **Accounting**, *Royal Wirtschaftsprüfung & Steuerberatung GmbH*, Vienna.
part-time

1998–1999 **Assistance in payroll department**, *LIBRO AG*, Guntramsdorf.
part-time

Josef Madersperger Gasse 3 – 2353 Guntramsdorf
☎ *+43 (1) 58801 - 18814* • ✉ *brosch@big.tuwien.ac.at*
• *http://www.big.tuwien.ac.at*

Research Projects

since 2012 **FAME**, *Formalizing and Managing Evolution in Model-Driven Engineering*.
http://www.modelevolution.org

2009–2011 **AMOR**, *Adaptable Model Versioning*.
http://www.modelversioning.org

2006–2008 **TRACK and TRADE**, *Creating a Data Mart for Floating Car Data*.
http://www.trackandtrade.org

Teaching

- Model Engineering
- Advanced Model Engineering
- Web Engineering
- Project Lab
- Supervision of BSc and MSc Theses

Community Service

- Participating in fFORTE WIT—Women in Technology
- Instructing informatics related workshops for children and students
- Webmaster for 10th International Conference on Web Engineering (ICWE'10)

Languages

Mother tongue **German**

Other languages **English**

Publications

Peer Reviewed Book Chapters

[1] Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. The Past, Present, and Future of Model Versioning. In Jörg Rech and Christian Bunse, editors, *Emerging Technologies for the Evolution and Maintenance of Software Models*, chapter 15, pages 410–443. IGI Global, 2011.

Peer Reviewed Journal Papers

[2] Petra Brosch and Andrea Randak. Position Paper: m2n-A Tool for Translating Models to Natural Language Descriptions. *Electronic Communications of the EASST*, Software Modeling in Education at MODELS 2010(34), 2010.

[3] Petra Brosch, Martina Seidl, Manuel Wimmer, and Gerti Kappel. A Conflict Aware Merging Approach for Evolving UML Models. *Submitted for Publication*.

Peer Reviewed Conference Papers

[4] Petra Brosch, Uwe Egly, Sebastian Gabmeyer, Gerti Kappel, Martina Seidl, Hans Tompits, Magdalena Widl, and Manuel Wimmer. Towards Scenario-Based Testing of UML Diagrams. In *6th International Conference on Tests & Proofs (TAP'12)*, to be published.

Josef Madersperger Gasse 3 – 2353 Guntramsdorf
☎ *+43 (1) 58801 - 18814* • ✉ *brosch@big.tuwien.ac.at*
• *http://www.big.tuwien.ac.at*

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

[6] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Gerti 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)*, pages 129–144. Springer, 2010.

[7] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In *Model Driven Engineering Languages and Systems (MoDELS'09)*, volume 5795 of *LNCS*, pages 271–285. Springer, 2009.

[8] Petra Brosch, Martina Seidl, and Gerti Kappel. A Recommender for Conflict Resolution Support in Optimistic Model Versioning. In *Proceedings of the International Conference Companion on Object Oriented Programming Systems Languages and Applications (Onward! @ SPLASH'10)*, pages 43–50. ACM, 2010.

[9] Petra Brosch, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Philip 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)*, pages 207–214. Springer, 2009.

[10] Manuel Wimmer, Martina Seidl, Petra Brosch, Horst Kargl, and Gerti Kappel. On Realizing a Framework for Self-tuning Mappings. In *Objects, Components, Models and Patterns - Proceedings of the 47th International Conference TOOLS EUROPE 2009*, volume 33 of *LNBIP*, pages 1–16. Springer, 2009. Vortrag: 47th International Conference, TOOLS EUROPE 2009, June 29-July 3, 2009., Zurich, Switzerland; 2009-06-29 – 2009-07-03.

Peer Reviewed Workshop Papers and Poster Presentations

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

[12] Petra Brosch. A Service Oriented Approach to Traffic Dependent Navigation Systems. In *IEEE Congress on Services 2008*, pages 269–272. IEEE, 2008.

[13] Petra Brosch. Improving Conflict Resolution in Model Versioning Systems. In *Proceedings of the 31st International Conference on Software Engineering, Companion Volume (ICSE'09)*, pages 355–358, 2009.

[14] Petra Brosch, Uwe Egly, Sebastian Gabmeyer, Gerti Kappel, Martina Seidl, Hans Tompits, Magdalena Widl, and Manuel Wimmer. Towards Semantics-Aware Merge Support in Optimistic Model Versioning. In *Models in Software Engineering - Workshops and Symposia at MODELS 2011, Reports and Revised Selected Papers*, volume 7167 of *LNCS*, pages 246–256. Springer, 2012.

[15] Petra Brosch, Sebastian Gabmeyer, Martina Seidl, and Gerti Kappel. On Formalizing EMF Modeling Operations with Graph Transformations. In *5th International Workshop UML and Formal Methods @ FM2012*, to be published.

[16] Petra Brosch, Gerti Kappel, Martina Seidl, and Manuel Wimmer. Teaching Model Engineering in the Large. In *Educators' Symposium @ Models 2009*, 2009.

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

[18] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. Colex: A Web-based Collaborative Conflict Lexicon. In *Proceedings of the 1st International Workshop on Model Comparison in Practice @ TOOLS 2010*, pages 42–49. ACM, 2010.

Josef Madersperger Gasse 3 – 2353 Guntramsdorf

☎ *+43 (1) 58801 - 18814* • ✉ *brosch@big.tuwien.ac.at*

• *http://www.big.tuwien.ac.at*

[19] Petra Brosch, Philip Langer, Martina Seidl, and Manuel Wimmer. Towards End-User Adaptable Model Versioning: The By-Example Operation Recorder. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 55–60. IEEE, 2009.

[20] Petra Brosch and Andrea Randak. m2n: Translating Models to Natural Language Descriptions. In *Proceedings of the 6th Educators' Symposium @ MODELS 2010*, 2010.

[21] Petra Brosch, Martina Seidl, and Konrad Wieland. Guiding Modelers through Conflict Resolution: A Recommender for Model Versioning. In *Proceedings of the International Conference Companion on Object Oriented Programming Systems Languages and Applications (SPLASH'10)*, pages 241–242. ACM, 2010.

[22] Petra Brosch, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Philip 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)*, pages 739–740. ACM, 2009.

[23] Petra Brosch, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Philip 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)*, pages 791–792. ACM, 2009.

[24] Petra Brosch, Martina Seidl, and Manuel Wimmer. Mining of Model Repositories for Decision Support in Model Versioning. In *Second European Workshop on Model Driven Tool and Process Integration (MDTPI) @ ECMDA-FA'09*, pages 25–33, WP09-05, 2009. CTIT Workshop Proceedings. Vortrag: Workshop on Model Driven Tool and Process Integration (MDTPI) in conjunction with ECMDA-FA 2009, Enschede, Niederlande; 2009-06-24.

[25] Petra Brosch, Konrad Wieland, and Gerti Kappel. Conflict Resolution in Model Versioning. In *1st International Master Class on Model-Driven Engineering, Poster Session Companion*, pages 17–18, 2010.

[26] Philip Langer, Konrad Wieland, and Petra 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.

[27] Dieter Pfoser, Sotiris Brakatsoulas, Petra Brosch, Martina Umlauft, Nektaria Tryfona, and Giorgos Tsironis. Dynamic Travel Time Provision for Road Networks. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS '08)*, pages 475–478. ACM, 2008.

Josef Madersperger Gasse 3 – 2353 Guntramsdorf
☎ *+43 (1) 58801 - 18814* • ✉ *brosch@big.tuwien.ac.at*
• *http://www.big.tuwien.ac.at*