FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Adaptable Model Versioning based on Model Transformation By Demonstration

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der technischen Wissenschaften

eingereicht von

Philip Langer
Matrikelnummer 0325934

an der
Fakultät für Informatik der Technischen Universität Wien

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

Diese Dissertation haben begutachtet:

(o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel)	(Prof. Dr. Jeff Gray)

Wien, 15.11.2011

(Philip Langer)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Adaptable Model Versioning based on Model Transformation By Demonstration

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften

by

Philip Langer
Registration Number 0325934

to the Faculty of Informatics
at the Vienna University of Technology

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

The dissertation has been reviewed by:

(o.Univ.-Prof. Dipl.-Ing. Mag. Dr. Gerti Kappel)

(Prof. Dr. Jeff Gray)

Wien, 15.11.2011

(Philip Langer)

Erklärung zur Verfassung der Arbeit

Philip Langer
Kulmgasse 32/9, 1170 Wien

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

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

This thesis would not have been possible without the valuable contributions of many people. I owe my gratitude to all those, who have supported me during the time when I worked on this thesis and who made this time to be a precious experience for me.

I am very grateful to *Dr. Gerti Kappel*, who provided me with every kind of support and freedom to explore. She constantly encouraged me to aim high and to keep pushing forward, while also giving me the guidance to get back on track when I struggled.

I am indebted to *Dr. Jeff Gray*, who supported me with valuable feedback and always kindly encouraged me to succeed with my thesis, despite the geographical distance and time difference.

My deepest gratitude is to *Dr. Martina Seidl* and *Dr. Manuel Wimmer*, who have been both great mentors and friends throughout my time working on my thesis. Martina and Manuel have always been there to listen and give advice. I am deeply grateful for their insightful comments and constructive criticisms.

I am also thankful to the entire AMOR team, *Petra Brosch*, *Dr. Gerti Kappel*, *Dr. Werner Retschitzegger*, *Dr. Wieland Schwinger*, *Dr. Martina Seidl*, *Konrad Wieland*, and *Dr. Manuel Wimmer*, who closely worked with me to build the basis of this thesis.

Especially, I wish to thank *Konrad Wieland* with whom I studied for five years and shared an office for three years. He has always been a great colleague and friend. I would not have enjoyed my time in academia so much without him.

I also would like to acknowledge all the other outstanding researchers, I had the honor to collaborate with during the last three years. My thanks go to *Dr. Jordi Cabot*, *Dr. Claudia Ermel*, *Dr. Markus Herrmannsdoerfer*, *Dr. Birgit Hofreiter*, *Dr. Christian Huemer*, *Dr. Horst Kargl*, *Dr. Maximilian Koegel*, *Christian Pichler*, *Dr. Yu Sun*, *Dr. Gabriele Taentzer*, and *Dr. Jules White*.

This thesis would not have been possible without the support of my family, *Regine* and *Hans*, *Julia* and *Peter*, and *Benjamin* and *Sara*. My family has always been an essential element of my life and is a constant source of advice, concern, and dependability.

Last but not least, I am deeply thankful to *Andrea Rucker*, who has always been there to share both my successes and my failings unconditionally. She encouraged me in difficult times and was greatly patient when I spent numerous nights and weekends working on this thesis instead of spending time with her. She always provided me with her help, advice, and understanding and constantly gave me the strength and confidence to move on.

Abstract

Model-driven engineering (MDE) is evermore adopted in academia and industry for being a new paradigm helping software developers to cope with the ever increasing complexity of software systems being developed. In MDE, software models constitute the central artifacts in the software engineering process, going beyond their traditional use as blueprints, and act as the single source of information for automatically generating executable software.

Although MDE is a promising approach to master the *complexity* of software systems, so far it lacks proper concepts to deal with the ever growing *size* of software systems in practice. Developing a large software system entails the need for a large number of collaborating developers. Unfortunately, *collaborative development of models* is currently not sufficiently supported. Traditional versioning systems for code fail for models, because they treat models just as plain text files and, as a consequence, neglect the graph-based nature of models.

A few dedicated *model versioning approaches* have been proposed, which directly operate on the models and not on the models' textual representation. However, these approaches suffer from four major deficiencies. First, they either support only one modeling language or, if they are *generic*, they do not consider important specifics of a modeling language. Second, they do not allow the specification of *composite operations* such as refactorings and thus, third, they *neglect* the importance of respecting the original intention behind composite operations for detecting conflicts and constructing a merged model. Fourth, the types of *detectable conflicts* among concurrently applied operations is *insufficient* and *not extensible by users*.

To address these deficiencies, we present four major contributions in this thesis. First, we introduce an *adaptable model versioning framework*, which aims at combining the advantages of two worlds; the proposed framework is generic and offers out-of-the-box support for *all modeling languages* conforming to a common meta-metamodel, but also allows to be *adapted* for enhancing the versioning support for *specific modeling languages*. Second, we propose a novel technique, called *model transformation by demonstration*, for easily specifying composite operations. Besides being executable, these composite operation specifications also constitute the *adaptation artifacts* for enhancing the proposed versioning system. More precisely, with our third contribution, we present a novel approach for *detecting applications of specified composite operations* without imposing any dependencies on the employed modeling environment. Fourth, we present a novel approach for detecting *additional types of conflicts* caused by *concurrently applied composite operations*. Furthermore, we contribute additional techniques for revealing potentially obfuscated or unfavorable merge results. Besides introducing the contributions from a conceptual point of view, we provide an open source implementation of these concepts and present empirical case studies and experiments for evaluating their usefulness and ease of use.

Kurzfassung

Model-driven engineering (MDE) findet als neues Softwareentwicklungsparadigma sowohl in der Wissenschaft und als auch in der Industrie immer mehr Anwendung. Dabei werden Modelle als zentrale Artefakte der Softwareentwicklung angesehen und dienen nicht nur als Skizze oder Entwurf, sondern stellen zur Generierung von lauffähiger Software die einzige und vollständige Spezifikation dar.

Auch wenn MDE ein vielversprechender Ansatz ist, der EntwicklerInnen dabei unterstützt die steigende *Komplexität* von Softwaresystemen zu meistern, fehlen derzeit Mittel und Wege mit der wachsenden *Größe* der zu entwickelnden Softwaresystemen umzugehen. Die Entwicklung großer Softwaresysteme erfordert die Zusammenarbeit vieler EntwicklerInnen. *Kollaborative Entwicklung von Modellen* wird jedoch derzeit nur unzureichend von MDE-Werkzeugen unterstützt. Herkömmliche Versionierungssysteme, eines der wichtigsten Werkzeuge für Softwarecode, sind für Modelle ungeeignet, da diese Systeme nur die textuelle Repräsentation von Modellen betrachten und die graphenähnliche Struktur von Modellen unberücksichtigt lassen.

Um dieses Problem zu lösen wurden einige *speziell für Modelle zugeschnittene Versionierungssysteme* vorgestellt, die direkt mit Modellen und nicht mit ihrer textuellen Repräsentation arbeiten. Aktuelle Systeme weisen jedoch einige Mängel auf. Erstens unterstützen aktuelle Systeme entweder nur *eine spezielle* Modellierungssprache oder sie sind generisch und lassen daher die *Besonderheiten von Modellierungssprachen* gänzlich unberücksichtigt. Zweitens lassen existierende Modellversionierungssysteme die wichtige Bedeutung von zusammengesetzten Operationen wie z.B. *Refactorings* außer Acht. Drittens verabsäumen diese Systeme die Erkennung einiger wichtiger Konfliktarten und sind nicht von BenutzerInnen erweiterbar.

Um die Mängel aktueller Systeme zu beseitigen, stellen wir ein *adaptierbares Modellversionierungssystem* vor, das die Vorteile von generischen und sprachspezifischen Versionierungssystemen vereint, indem es einerseits generisch ist, jedoch von BenutzerInnen in Hinsicht auf die Besonderheiten der Modellierungssprachen erweitert werden kann. Dafür stellen wir eine neue Technologie namens *Model Transformation By Demonstration* vor, die es auf einfache Weise erlaubt zusammengesetzten Operationen zu spezifizieren. Diese Spezifikationen sind nicht nur automatisch anwendbar, sondern dienen auch zur Erweiterung unseres Versionierungssystem. Einerseits ermöglichen sie die *Erkennung von Anwendungen* der spezifizierten Operationen. Andererseits ermöglichen sie die *Erkennung spezieller Konflikte*, die sich aus der gleichzeitigen Anwendungen von zusammengesetzten Operationen ergeben. Darüber hinaus behandlet diese Arbeit auch die Erkennung von weiteren potentiell unerwünschten Auswirkungen gleichzeitiger Änderungen. Die in dieser Arbeit vorgestellten Konzepte wurden in Form einer quelloffenen Implementierung veröffentlicht und mit empirischen Fallstudien und Experimenten evaluiert.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Software engineering [NRB69, PI82], being the systematic discipline of building high quality software systems, has a long history going back to the late 1960s. Since then, researchers and practitioners have been struggling to cope with the ever growing *complexity* and *size* of the developed systems. One way of coping with the complexity of a system has been raising the level of abstraction in the languages used to specify a system. As stated by Smith and Stotts, "the history of programming is an exercise in hierarchical abstraction. In each generation, language designers produce explicit constructs for conceptual lessons learned in the previous generation, ..." [SS02]. Besides dealing with the complexity of software systems under development, also managing the size of software systems constitutes a major challenge. As stated by Ghezzi et al., "software engineering deals with the building of software systems that are so large or so complex that they are built by teams of engineers" [GJM02]. Orthogonal to the challenge entailed by the complexity and size of software systems, dealing with the demand to *evolve* constantly a system, in order to meet ever changing and growing requirements, constitutes an additional major challenge. To summarize, Parnas defines software engineering as the "multi-person construction of multi-version software" [Par75].

More recently, *model-driven engineering* (MDE) has been proposed as a new paradigm for raising the level of abstraction once again [Béz05, GS03, Sch06]. In MDE, models, being an abstraction of the real world, are considered as central artifacts in the software engineering process, going beyond their traditional use as sketches and blueprints. Models constitute the basis and the single source of information to specify and automatically generate an executable system. Thereby, developers may build models that are less bound to an underlying implementation technology and are much closer to the problem domain [Sel03]. Consequently, developers are enabled to focus on modeling the problem domain instead of worrying about implementation details of a solution domain. As an ultimate result, MDE promises to decouple the developed solution from implementation-specific platforms, to raise the efficiency and ease of developing software, and, by implication, to achieve a software of higher quality. In the context of MDE,

the Object Management Group[1] (OMG) has set the prerequisites for the adoption of MDE in practice by standardizing first the Unified Modeling Language (UML) [OMG03] and later the language for defining modeling languages, called *Meta-Object Facility* (MOF) [OMG04], as well as the common model exchange format *XML Metadata Interchange* (XMI) [OMG07] in the course of the *Model Driven Architecture* initiative [KWB03, Mel04, OMG05b].

Although MDE is a promising approach to cope with the ever growing *complexity* of systems, so far it lacks proper concepts to deal with the ever growing *size* of systems being built in practice. This, however, is crucial for MDE to succeed as a new paradigm in software engineering [FR07]. Developing a large system entails the need for a large number of developers who collaborate to succeed in creating a large system. Unfortunately, *collaborative development of models* is not sufficiently supported yet by current modeling tools [ABK+09]. As in traditional code-centric software engineering, *versioning systems* [CW98, Men02] are required, which allow for concurrent modification of the same model by several developers and which are capable of merging the operations applied by all developers to obtain ultimately one consolidated version of a model again.

1.2 Model Versioning in its Infancy

In traditional code-centric software engineering, text-based versioning systems, such as Git[2], Subversion[3], and CVS[4], have been successfully deployed to allow for collaborative development of large software systems. The reason for their success probably is that they can be used independently from the used programming language and integrated development environment (IDE). Especially, *optimistic versioning systems* gained remarkable popularity because they enable several developers to work concurrently on the same artifacts instead of pessimistically locking each artifact for the time it is changed by one developer. The price to pay for being able to work in parallel when using optimistic versioning systems is that after all developers finished their work, the operations of all developers have to be merged again. Merging is sometimes a tedious task because in case of spatially overlapping modifications *conflicts* are raised, which have to be manually resolved.

To enable *collaborative modeling* among several team members, optimistic text-based versioning systems have been reused for models. Unfortunately, it turned out quickly that applying text-based comparison and conflict detection is inadequate for models and leads to unsatisfactory results [ABK+09]. This is because such versioning systems consider only text lines in a text-based representation of a model as, for instance, the XMI serializations. As a result, the information stemming from the model's graph-based structure is destroyed and associated syntactic information is lost. Furthermore, obtained textual differences between the serialization of two versions of a model strongly differ from the operations actually performed by developers in their modeling environments. In other words, one operation applied to a model in a modeling editor often causes several scattered operations of multiple lines across the model's textual rep-

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

resentation. Consequently, the correct identification of the actual model operations is impeded and hardly comprehensible when applying text-based versioning systems to models. However, correctly obtaining and understanding the actual model operations is crucial for detecting the effective conflicts and for creating a correctly merged model unifying all original operations. Nguyen et al. used the term *impedance mismatch* [NMB04] to refer to this unfavorable mismatch between an artifact's representation put under version control and the representation users usually work with. This mismatch constitutes the root of the aforementioned drawbacks.

To overcome these drawbacks caused by the impedance mismatch of text-based versioning systems used for models, dedicated *model versioning* approaches have been recently proposed (cf. Section 2.1.2 for a survey). Comparable to *syntactic merge approaches* [Men02], such approaches do not operate on the textual representation. Instead, they directly operate on the model's graph-based structure to obtain applied operations, detect conflicts, and to eventually create a merged version. However, after carefully surveying these approaches the following major deficiencies have been identified in current approaches mitigating their use in practice.

Deficiency 1: Dependency on the modeling editor versus imprecise versioning. The first task to be achieved when merging two concurrently modified versions is to obtain the operations that have been applied by developers in parallel. There are two approaches for obtaining operations. On the one hand, they may be identified using model differencing algorithms[5], which take two versions of a model as well as their common base version as input and compute the model differences by comparing these three states. On the other hand, operations between two versions of a model may be directly recorded[6] in the modeling environment as they are performed by the user. In comparison to model differencing approaches, operation recording is, in general, more precise than model differencing. However, operation recording approaches inherently put *restrictions on the used modeling editors* because the editor used for modifying the model has to be capable of recording operations and represent them in a commonly processable format. However, recalling successful versioning systems for code, such as SVN and Git, only approaches that are independent from the used editor gained significant adoption in practice. Thus, we may draw the conclusion that a versioning system having an inherent dependency on the used editor might not find broad adoption in practice. In particular, as long as no standardized format for representing operations is available and as long as available modeling environments implement this standardized format, operation recording being the basis for collaborative modeling is in severe contradiction to the inherent vendor-neutral approach followed by the MDA initiative. Model differencing on the contrary is in tune with the goal of vendor independence but it *lacks the precision of computed operations*. The availability of precise operations, however, is crucial for a proper quality of the merge result.

Furthermore, existing approaches are inflexible with respect to the trade-off between generic (i.e., language-independent) versioning and language-specific versioning. *Generic versioning systems* are applicable for all modeling languages conforming to a common meta-metamodel. However, such versioning systems are characterized by deficient versioning support because they neglect language-specific operations and conflicts. In contrast, *language-specific versioning sys-*

[5] Also referred to as *state-based versioning* [BP08, CW98, Men02].

[6] Also referred to as *change-* or *operation-based versioning* [CW98, KHWH10, LvO92, Men02].

tems are tightly bound to a modeling language and, therefore, usually provide better versioning quality for that specific language. However, this inflexibility poses a major drawback because of the rapidly growing number of domain-specific modeling languages (DSMLs) [GTK+07,KT08]. Moreover, it is very likely that several modeling languages are concurrently applied within one single project. Using language-specific versioning systems would entail using several versioning systems—one for each set of supported modeling languages—in one single project, which is usually infeasible. The issue of *generic versus language-specific merging* has already been posed by Westfechtel in [Wes91]: "On the one hand, the merge tool should be general, i.e., it should be applicable to arbitrary software documents. [...] On the other hand, the merge tool should be intelligent, i.e., it should be based on a high-level concept of change in order to produce a result, which makes sense." However, current model versioning approaches still offer no adequate solution to this issue yet.

To summarize, the challenge is to achieve a high-quality operation and conflict detection without imposing dependencies regarding the used modeling editor and the supported modeling languages. How may the impreciseness of state-based model differencing be overcome? How may the quality of the conflict detection in generic versioning systems be increased in order to achieve the quality offered by versioning systems that also incorporate language-specific knowledge? Which language-specific knowledge is necessary for that? How may this knowledge be represented and plugged into a generic model versioning system?

Deficiency 2: Tedious specification of composite operations. As in traditional code-centric software development, also models are often subjected to composite operations [SPLTJ01]. A composite operation is a set of cohesive atomic operations that are applied within one transaction to achieve ultimately one common goal. The most prominent class of such composite operations are *refactorings* as introduced by Opdyke [Opd92] and further elaborated by Fowler et al. [FBB+99]. Refactorings have well-defined preconditions, which specify whether a refactoring may be applied to a current state of an artifact and comprise a set of actions describing how to modify or "refactor" the current state to obtain an improved structure. As stated by Dig et al. [DMJN08], the knowledge and consideration of applied refactorings in the versioning process significantly improves the quality of the merge because the intention behind those operations constituting the refactoring can be considered while merging. The importance of considering refactorings in the context of the parallel evolution of software has also been stressed by Mens et al. [MTR05]; the knowledge on applied refactorings enables to detect so-called "structural refactoring conflicts". Furthermore, the information on the applied composite operations helps other developers to better understand the evolution of a software artifact [KHvW+10]. Current model versioning approaches, however, largely neglect the importance of considering composite operations in model versioning.

To enable model versioning systems to consider composite operations in the merge process, composite operations first have to be specified clearly. This specification must include the operation's *precise preconditions as well as their mechanics* (i.e., the comprised atomic operations). Composite operations are inherently specific to a certain modeling language. Supporting many modeling languages requires to specify clearly all relevant composite operations for the languages of interest. When keeping domain-specific languages in mind, a pantheon of composite

operations have to be developed manually. As it seems to be impossible to pre-specify all combinations of composite operations, ideally, developers themselves should be enabled to specify composite operations on their own. Developers, however, are usually not trained to develop composite operations, or in more general terms *model transformations*, comprising explicit preconditions using currently existing model transformation techniques [SW08, Var06].

Therefore, the challenge is to develop an approach that eases the burden of creating well-defined specifications of composite operations. How may developers who are not trained to use model transformation techniques be enabled to develop model transformations on their own?

Deficiency 3: Absence of information on applied composite operations. For taking composite operations into account, applications of composite operations have to be available explicitly in the list of obtained operations that have been applied between two versions of a model. One way to explicate applications of composite operations is to record the operations directly in the modeling editor. However, such recording approaches strongly depend on the modeling editor (cf. Deficiency 1). Moreover, a set of manually applied atomic operations, having together the intent of a composite operation (which is indeed frequently happening in practice [MHPB09]), cannot be identified by operation recording approaches because no explicit action has been executed in the modeling editor. When refraining from recording operations directly, state-based model differencing approaches have to be used. However, current model differencing approaches are not capable of detecting applications of composite operations because, so far, the *a posteriori detection of applications of composite operations* is an open issue. As a result, the information on applied composite operations is unavailable, which is, however, the crucial prerequisite for considering them subsequently in the merge process.

To this end, the challenge is to build a model differencing approach that is capable of detecting applications of composite operations a posteriori. Enabling the detection of composite operation applications, which inherently are language-specific, is even more challenging, when aiming to apply a generic model differencing algorithm for the sake of language-independence. How may applications of composite operations be identified by a generic algorithm that solely analyzes two subsequent versions of a model? How may developers easily extend the set of detectable composite operations?

Deficiency 4: Insufficient conflict detection. Existing model versioning systems fail to detect correctly all relevant conflicts [ABK+09]. Admittedly, in the model versioning research community, no full consensus has been established yet concerning the conflicts that are indeed relevant. Whether a scenario should be classified as a conflict often depends on how a modeling language is used, the goal of the modeling project, the phase of a project, or even on personal preferences. Hence, adaptability of the conflict detection component is all the more important because it enables developers to decide, depending on their use case, for which scenarios a conflict should be reported. However, current model versioning systems mostly provide no means for adaptability. Only very few systems support some basic configurations, such as the unit of comparison, but they do not allow users to perform more sophisticated customizations with respect to language-specific knowledge. Consider, for instance, an operation in a UML class diagram, which is primarily signified by its name, its return type, and its parameters. If developer 1

modifies an operation's return type and developer 2 changes the name of the same operation, it is very likely that naively merging both modifications leads to an unfavourable result because both developers modified the primary meaning of the same operation, whereas they were not aware of the opposite modification. Aggravatingly, both modifications are not spatially overlapping, which is why current model versioning would not raise a conflict.

As mentioned earlier, composite operations often have specific preconditions restricting the scenarios in which they may be applied. If developer 1 performs operations that violate the preconditions of an application of a composite operation that was performed by developer 2, a conflict should be raised; otherwise, the composite operation fails to be applied correctly in the merge process, which might lead to an erroneously merged model. Moreover, the knowledge on composite operations give a set of atomic operations a superior meaning reflecting the original intention of the modeler performing those operations more precisely. Being aware of this superior meaning, a model versioning system should regard the intention while merging and, for instance, incorporate also model elements in the application of the composite operations while merging that have been concurrently added by the another developer. However, current model versioning systems fail to raise conflicts with respect to composite operation's preconditions and neglect the original intention behind applied composite operations.

Addressing this deficiency poses several challenges, especially when aiming to use a generic conflict detection component for the sake of language-independence. What are the specifics of a modeling language that should be considered by a model versioning system in order to increase the quality of the conflict detection? How can these specifics be configured by users? How can a generic conflict detection component be designed to take those configured specifics into account? In the context of composite operations, it is currently unclear when to raise a conflict with respect to composite operations. Which types of conflicts may occur in scenarios that involve composite operations? How may such conflicts be detected by a generic conflict detection component for a user-extensible set of custom composite operations? How may a generic model versioning system also incorporate the original developer's intention behind composite operations in the merge process?

1.3 Contributions

The overall goal of this thesis is to provide precise operation and conflict detection in the context of model versioning without imposing dependencies regarding the used modeling editor or modeling language. Nevertheless, language-specific composite operations should be considered and therefrom resulting merge conflicts should be detected.

Before we discuss each contribution in detail, we briefly outline the applied versioning process (cf. Figure 1.1) as this process constitutes the context of the contributions presented in this thesis. In the course of this thesis, we apply a *versioning process*, which is referred to as *check-out/check-in protocol* [ELH+05] in the literature. According to this process, developers may concurrently check-out the latest version V_o of a model from a common repository at the time of t_0 (cf. Figure 1.1). Thereby, a local working copy of V_o is created. Both developers may independently modify their working copies in parallel. As soon as one developer completes the work, assume this is developer 1, she performs a check-in at t_1. Because no other developer

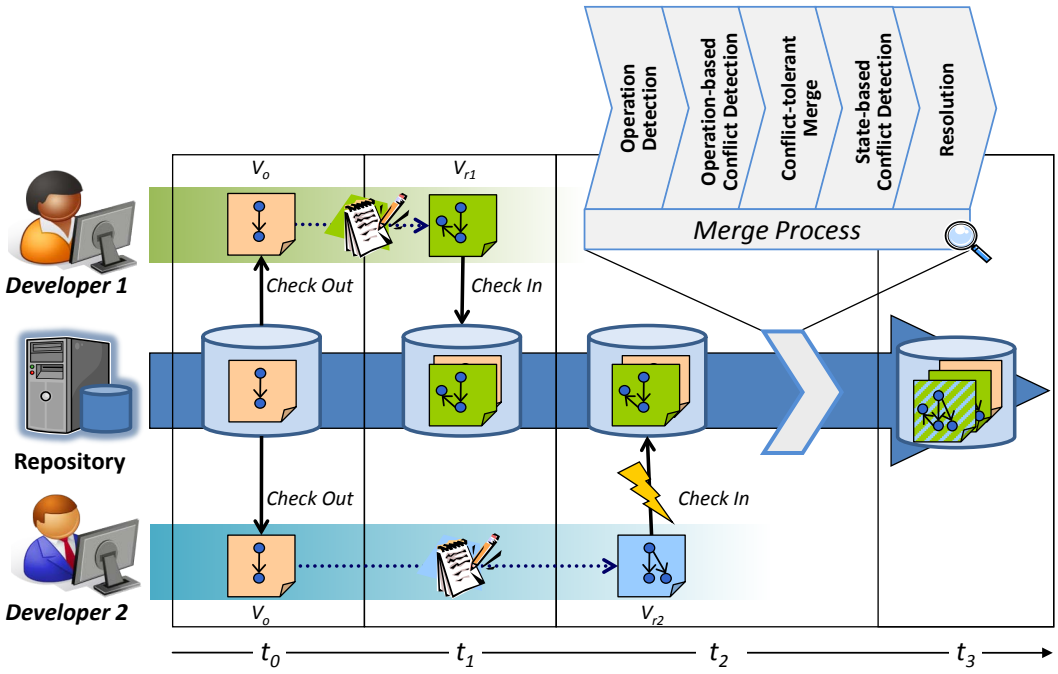


Figure 1.1: Versioning Process

performed a check-in in the meanwhile, her working copy can be saved directly as a new revised version V_{r1} in the repository. Whenever developer 2 completes her task and performs the check-in, the versioning system recognizes that a new version has been created since the check-out. Therefore, the merge process is triggered at t_2 in order to merge the new version V_{r1} in the repository with the version V_{r2} by developer 2. Once the merge is carried out, the resulting merged version, which incorporates the operations of developer 1 as well as those operations performed by developer 2, is saved in the repository.

Within the versioning process, the *merge process* comprises the most sophisticated steps with the goal of unifying all concurrently performed operations of the involved developers and obtaining a consolidated merged model version. Ideally, this merged model version reflects the intentions of all developers without introducing any errors in the merged model. The first step of the merge is the *operation detection* aiming to identify the operations that have been applied by the developers to their working copies. In the next step, namely the *operation-based conflict detection*, all concurrent operations performed by both developers are revealed that interfere with each other. Besides operation-based conflicts, we also aim to detect state-based conflicts in a model to which all operations of both developers have been applied. By state-based conflicts we refer to violations of the modeling language's validation rules coming from the metamodel and additional context conditions expressed using the Object Constraint Language (OCL) [OMG10]. To reveal such conflicts, we first have to compute a merged model version to be checked against the validation rules. As we might have identified operation-based conflicts previously, we apply a *conflict-tolerant merge*, which is capable of tolerating operation-

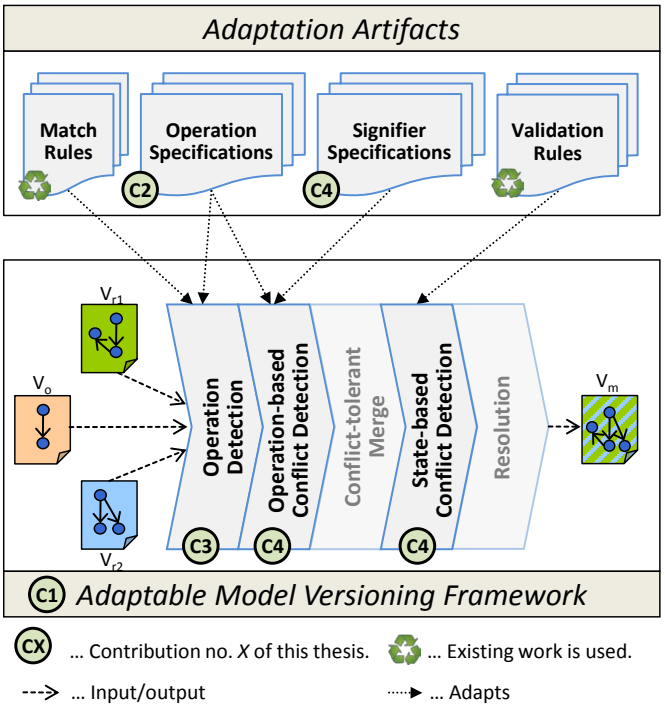


Figure 1.2: Contributions of this Thesis

based conflicts and which creates a merged version in every case. The resulting merged model is the input for the *state-based conflict detection*, which determines whether the merged model is well-formed and valid in terms of the modeling language's rules. In case, conflicts have been identified, the involved developers have to specify a *resolution* for the raised conflicts first in order to obtain a consolidated model, which is finally saved as the new version in the repository.

This general merge process has been conjointly elaborated by all project participants[7] in the course of the research project AMOR[8]. This thesis contributes solutions for creating the adaption artifacts and for realizing the steps in the merge process marked by C1 to C4 in Figure 1.2, whereas the numbers 1 to 4 denote the respective contribution number introduced below. The remaining adaptation artifacts, namely the approaches for specifying match rules and adaptation rules, are adapted and integrated from existing work; the other steps in the merge process, namely *conflict-tolerant merge* and *resolution*, are not the particular focus of this thesis. For more information on these two steps, we kindly refer to the Ph.D. theses by Brosch [Bro11] and Wieland [Wie11], which have also been elaborated in the course of the project AMOR. In the following, we outline the contributions of this thesis in more detail.

[7] In alphabetical order: Petra Brosch, Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, Martina Seidl, Konrad Wieland, and Manuel Wimmer

[8] AMOR (`http://www.modelversioning.org`), a research project funded by the Austrian Federal Ministry of Transport, Innovation, and Technology and the Austrian Research Promotion Agency under grant FIT-IT-819584

Contribution 1: Adaptable Model Versioning. According to the main principle of AMOR, the overall contribution of this thesis is to provide an adaptable versioning framework allowing for proper versioning support while ensuring generic applicability for various DSMLs. Therefore, the generic framework offers out-of-the-box support for all modeling languages conforming to a common meta-metamodel and, additionally, it enables users to improve the quality of the versioning capabilities by adapting the framework to specific modeling languages using certain well-defined adaptation points. Thereby, developers are empowered to balance flexibly between reasonable adaptation efforts and the required level for versioning support. The adaptation artifacts that can be created and plugged into the system in order to improve the versioning support for specific modeling languages are depicted in Figure 1.2.

Contribution 2: Composite Operation Specifications. Predefined composite operations are helpful for efficient modeling: in particular, for automatically executing recurrent refactorings, applying model completions, and introducing patterns to existing models. Moreover, as previously stated, the availability of explicit specifications of composite operations is the prerequisite for considering applications of such operations in the merge process. Composite operations are inherently specific to a certain modeling language. However, as it is infeasible to predefine all relevant operation specifications for all modeling languages being used, developers should be enabled to specify such operations on their own and adapt the versioning system to allow for these composite operation specifications (cf. Figure 1.1). Composite operations are, in more general terms, *endogenous model transformations* [MG06]; that is, model transformations that incrementally transform an existing model. Consequently, the source and the target metamodel of an endogenous model transformation are the same. However, the specification of new model transformations requires programming skills involving dedicated model transformation languages and, by implication, deep knowledge of the respective metamodel [SW08, Var06]. Usually, developers do not have such skills.

Therefore, in this thesis, we introduce a method for specifying endogenous model transformation within the user's modeling language and environment of choice enabling to create easily Operation Specifications (cf. Figure 1.2). The ease of creation is achieved by introducing *model transformation by demonstration*. Thereby, developers apply or "demonstrate" the transformation to an example model once and, from this demonstration as well as from the provided example model, the generic transformation (i.e., the Operation Specification) is semi-automatically derived including its explicit preconditions, operations to be applied, and postconditions. For model versioning purposes, endogenous transformations are of major importance, which is why we focus on specifying endogenous transformations in this thesis. However, we also show how this approach for endogenous transformation can be extended to also enable the specification of *exogenous transformations* [MG06]; that is, transformations generating a new target model from an existing source model, whereas source and target model may correspond to different metamodels. By using this extension, transformations can be specified by demonstration that translate models from one modeling language to another.

Contribution 3: Operation Detection. The first step of the merge process (cf. Figure 1.2) is to identify operations explicitly—including atomic operations as well as composite operations—which have been applied between two versions of a model (e.g., V_o and V_{r1} in Figure 1.2). As previously stated, operations applied between two versions of a model can be obtained either by recording the operations directly in the modeling editor or by applying model differencing. To avoid restricting the editor to be used, we refrain from recording the operations and apply model differencing using a two-phase process. First, a match is computed, which describes the correspondences between two versions of a model. In the second phase, differences are obtained by a fine-grained comparison of all corresponding model elements based on the beforehand computed match. Consequently, the quality of the obtained operations heavily depends on the quality of the computed match. To achieve a high-quality match, we assign *universally unique IDs* (UUIDs) to each model element and exploit these UUIDs for precisely matching model elements again after they have been modified. However, removed and re-added model elements (e.g., cut and paste) or similar model elements that have been added concurrently, have a different UUID, although they are equal. Hence, they cannot be matched because the content and characteristics of a model element are not considered in UUID-based matching. Aggravatingly, it is specific to the modeling language, to decide which characteristics of a model element should be used for determining whether they should be considered as a characteristic-based match. Therefore, we allow developers to specify language-specific *match rules*, which adapt the behaviour of the match algorithm for elements that could not be matched based on UUIDs. The language for expressing these match rules as well as the framework for evaluating those rules have been reused from existing work [Kol09].

Based on this improved match, *atomic operations* may be obtained precisely. However, as motivated above, not only atomic operations but also *composite operations* are a valuable source of information for versioning and allow for considering the actual developer's intention behind a set of atomic operations. Therefore, in this thesis, we contribute an a posteriori *composite operation detection* method by which occurrences of composite operations applied between two versions of a model can be identified. The specifications of composite operations, which are created by users (cf. Contribution 2), is used for automatically executing them in the modeling environment as well as for detecting applications of the executed composite operations. Hence, users may easily extend the set of detectable and executable composite operations by using the aforementioned *model transformation by demonstration* approach.

Contribution 4: Conflict Detection. Having obtained all atomic operations as well as all composite operations that have been applied concurrently by two developers, we then have to search for *conflicts*. We distinguish between two types of conflicts, in particular, operation-based conflicts and state-based conflicts.

Operation-based conflicts denote two concurrently applied operations that interfere with each other. Such conflicts occur, if, for instance, one developer deletes a model element and another developer modifies the same model element. Obviously, we may not apply both operations without omitting the effect of one of these operations. For detecting such conflicts, we introduce dedicated conflict detection patterns in this thesis. Besides operation-based conflicts between atomic operations, we also have to regard operation-based conflicts arising from the application

of composite operations. For instance, if a composite operation that has been applied by one developer cannot be applied anymore after a concurrent atomic operation performed by the other developer, because the atomic operation modifies the model in a way that the preconditions of the composite operation fail. Therefore, we present an algorithm to identify situations in which applications of composite operations are interfered by concurrent operations performed by the opposite developer. Using the contributed conflict detection algorithms, which respect atomic operations as well as composite operation specifications, we are able to detect a wide range of important conflicts. However, for certain modeling languages, developers might want to adapt the versioning system to raise additional warnings with respect to language-specific knowledge. For example, as already mentioned above, two developers concurrently modify the same operation in a UML class diagram. Developer 1 changes the operation name, while developer 2 concurrently modifies the same operation's return type. A generic model versioning system is not aware of the fact that an operation's return type, in combination with its name and its parameters convey the *superior meaning* of an operation. As a result, no warning will be raised for these parallel modifications, because they are indeed not spatially overlapping, but they concurrently modify the superior meaning of the same operation potentially leading to unrecognized contradictions. To address this deficiency, we introduce an adaptation point allowing users to specify so-called *signifiers* of model element types of their modeling languages (cf. Signifier Specifications in Figure 1.2). By signifier, we refer to a combination of specific features of a model element type, which convey the superior meaning of its instances (e.g., the name, the return type, and the parameters of a UML operation). For detecting such issues mentioned before, we present a dedicated detection algorithm, which analyses the concurrent modifications of a model element's signifier based on the language-specific signifier specifications provided by the user.

State-based conflicts denote violations of the *validation rules* of a modeling language in the merged model. Such violations are also referred to as inconsistencies in literature. Validation rules for checking the consistency are inherently specific to a modeling language and may, therefore, be plugged into the system. Once plugged in, the versioning system validates each merged model using the specified validation rules and raises additional conflicts in case a rule is violated. Well-formedness and validation rules are part of the modeling language definition. Therefore, we reuse those definitions and apply existing validation frameworks to reveal state-based conflicts.

Open Source Implementation. Besides introducing all elaborated approaches from a conceptual point of view, we provide a prototypical implementation of the approaches presented in this thesis. The contributed implementations are based on the Eclipse Modeling Framework[9] [SBPM08] and available under the terms of the Eclipse Public License[10] (EPL 1.0). For more information on the contributed implementations, we kindly refer to Appendix A.

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

[10] `http://www.eclipse.org/legal/epl-v10.html`

1.4 Thesis Outline

This thesis is structured according to the previously introduced merge process. Parts of this thesis have been published in peer-reviewed journals, conferences, and workshops. Some initial ideas originate from previous work published in my Master's thesis [Lan09] (in German) and have been extended in this Ph.D. thesis. In the following, we give a short overview of the remaining chapters of this thesis and refer to our publications that partially overlap with the content of the respective chapter.

Chapter 2: State of the Art In the next chapter, we introduce the fundamental concepts of the involved research domains and survey existing approaches in the area of versioning, software adaptation, and model transformation. This chapter contains contents also published in [BKL+11a, KLR+11].

Chapter 3: Adaptable Model Versioning. In this chapter, we present the big picture of the proposed adaptable model versioning system. In particular, we introduce some motivating examples posing challenges to be solved in this thesis and present the generic AMOR merge process, which has been conjointly elaborated by all project participants. Next, we show how this process is extended to be adaptable by users in order to incorporate language-specific knowledge. This chapter contains contents also published in [BKS+10].

Chapter 4: Model Transformation By Demonstration. The specification of composite operations is the prerequisite for respecting applications of composite operations in the merge process. Therefore, we introduce our editor- and language-independent approach for specifying *model transformation by demonstration* for endogenous transformations, as well as for exogenous transformations in Chapter 4. This chapter contains contents also published in [BLS+09, LWB10, LWK10].

Chapter 5: Operation Detection. In this chapter, we show how operations applied between two successive versions of a model are obtained by only analyzing their states. In particular, we provide insights into the applied match function for finding corresponding model elements across model versions, the identification of atomic operations applied between these model versions, and, finally, how applications of composite operations are detected a posteriori. This chapter contains contents also published in [LWB10, TELW10, TELW11].

Chapter 6: Conflict Detection. Having obtained all applied operations, this chapter presents the conflict patterns used to detect operation-based conflicts between atomic operations. Subsequently, we introduce our approach to detecting conflicts between composite operations. Moreover, we describe the specification as well as the detection of custom language-specific conflicts and, finally, we show how state-based conflicts are revealed. This chapter contains contents also published in [LWB10, TELW10, TELW11].

Chapter 7: Evaluation. In this chapter, we provide a detailed evaluation of each contribution presented in this thesis. This involves case studies, empirical user studies as well as precision/recall analysis and performance tests of the contributed implementations of the presented approaches. In addition to the evaluation of our own approach we also present comparisons with state-of-the-art approaches in the respective fields.

Chapter 8: Conclusion. Finally, the contributions of the thesis are summarized and critically discussed. In this chapter, we point out current limitations and interesting research directions to be addressed in future.

CHAPTER 2

State of the Art

In this chapter, we introduce the scientific foundations and survey the state of the art in the research areas that are related to the topics of this thesis. As the overall goal of this thesis is concerned with versioning of software models, we introduce the scientific background of *versioning* in software engineering being the predecessor of model versioning and survey existing model versioning systems in Section 2.1. Subsequently, we introduce the research area of *software adaptation* in Section 2.2 because the proposed model versioning system is designed to be adaptable to specific modeling languages. One major adaptation point of the model versioning system concerns composite operations applied to models. Composite operations are, in more general terms, *model transformations*. Thus, we survey existing model transformation approaches in Section 2.3.

2.1 Versioning

The history of versioning in software engineering goes back to the early 1970ies. Since then, software versioning was constantly an active research topic. As stated by Estublier et al. in [ELH+05], the goal of software versioning systems is twofold. First, such systems are concerned with maintaining a historical archive of a set of artifacts as they undergo a series of operations and form the fundamental building block for the entire field of Source Configuration Management (SCM), which deals with controlling change in large and complex software systems. Second, versioning systems aim at managing the evolution of software artifacts performed by a distributed team of developers.

In that long history of research on software versioning, diverse formalisms and technologies emerged. To categorize this variety of different approaches, Conradi and Westfechtel [CW98] proposed *version models* describing the diverse characteristics of existing versioning approaches. A version model specifies the objects to be versioned, version identification and organization, as well as operations for retrieving existing versions and constructing new versions. Conradi and Westfechtel distinguish between the *product space* and the *version space* within version models. The product space describes the structure of a software product and its artifacts without taking

versions into account. In contrast, the version space is agnostic of the artifacts' structure and copes with the artifacts' evolution by introducing versions and relationships between versions of an artifact, such as, for instance, their differences (deltas). Further, Conradi and Westfechtel distinguish between extensional and intentional versioning. *Extensional versioning* deals with the reconstruction of previously created versions and, therefore, concerns version identification, immutability, and efficient storage. All versions are explicit and have been checked in once before. *Intentional versioning* deals with flexible automatic construction of consistent versions from a version space. In other words, intentional versioning allows for annotating properties to specific versions and querying the version space for these properties in order to derive a new product consisting of a specific combination of different versions.

In this thesis, we only consider extensional versioning in terms of having explicit versions, because this kind of versioning is predominantly applied in practice nowadays. Furthermore, we focus on the *merge phase* in the optimistic versioning process (cf. Figure 1.1). In this section, we first outline the fundamental design dimensions of versioning systems. Subsequently, we present some representatives of versioning systems using different designs. Finally, we elaborate on the consequences of different design possibilities considering the quality of the merged version based on an example.

2.1.1 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 2.1). The first dimension concerns the product space, in particular, the *artifact representation*. This dimension denotes the representation of a software artifact, on which the merge approach operates. The used representation may either be *text-based* or *graph-based*. Some merge approaches operate on a tree-based representation. However, we consider a tree as a special kind of graph in this categorization. The second dimension is orthogonal to the first one and concerns how deltas are *identified*, *represented*, and *merged* in order to create a consolidated version. Existing merge approaches either operate on the *states*; that is, the versions of an artifact, or on identified operations that have been applied between a common origin model (cf. Version 0 in Figure 1.1) and the two successors (cf. Version 1 and 2 in Figure 1.1).

When merging two concurrently modified versions of a software artifact, conflicts might inevitably occur. The most basic types of conflicts are *update-update* and *delete-update* conflicts. Update-update conflicts occur if two elements have been updated in both versions whereas delete-update conflicts are raised if an element has been updated in one version and deleted in the other. A detailed discussion on more complex types of conflicts is given in Chapter 3. For more information on software merging in general, the interested reader is referred to [Men02].

Text-based merge approaches operate solely on the textual representation of a software artifact in terms of text files. Within a text file, the atomic unit of the versioned text file may either be a paragraph, a line, a word, or even an arbitrary set of characters. The major advantage of such approaches is their independence of the programming languages used in the versioned artifacts. Since a solely text- based approach does not require language-specific knowledge it may be adopted for all flat text files. This advantage is probably, besides simplicity and efficiency, the reason for the widespread adoption of pure text-based approaches in practice. However,

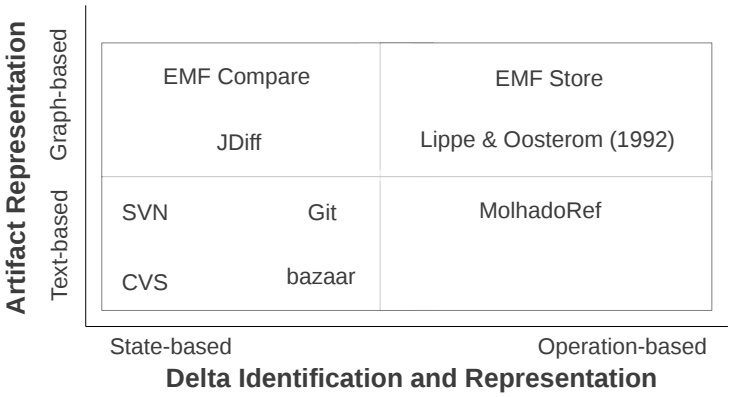


Figure 2.1: Categorization of Versioning Systems

when merging flat files—agnostic of the syntax and semantics of a programming language—both compile-time and run-time errors might be introduced during the merge. Therefore, graph-based approaches emerged, which take syntax and semantics into account.

Graph-based merge approaches operate on a graph-based representation of a software artifact for achieving more precise conflict detection and merging. Such approaches de-serialize or translate the versioned software artifact into a specific structure before merging. Mens [Men02] categorized these approaches in *syntactic* and *semantic merge approaches*. Syntactic merge approaches consider the syntax of a programming language by, for instance, translating the text file into the abstract syntax tree and, subsequently, performing the merge in a syntax-aware manner. Consequently, unimportant textual conflicts, which are, for instance, caused by reformatting the text file, may be avoided. Furthermore, such approaches may also avoid syntactically erroneous merge results. However, the textual formatting intended by the developers might be obfuscated by syntactic merging because only a graph-based representation of the syntax is merged and has to be translated back to text eventually. Westfechtel was among the first to propose a merging algorithm that operates on the abstract syntax tree of a software artifact [Wes91]. Semantic merge approaches go one step further and consider also the static and/or dynamic semantics of a programming language. Therefore, these approaches may also detect issues, such as undeclared variables or even infinite loops by using complex formalisms like program dependency graphs and program slicing. Naturally, these advantages over flat textual merging have the disadvantage of the inherent language dependence (cf. [Men02]) and their increased computational complexity. Furthermore, it is not always trivial to point the developers 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 2.1 is orthogonal to the first one and considers *how deltas are identified and merged* in order to create a consolidated version. This dimension is agnostic of the unit of versioning. Therefore, a versioned element might be a line in a flat text file, a node in a graph, or whatsoever constitutes the representation used for merging.

State-based merging compares the states, i.e., versions, of a software artifact to identify the differences (deltas) between these versions and merge all differences that are not contradicting with each other. Such approaches may either be applied to two states (Version 1 and Version 2 in Figure 1.1), called two-way merging, or to three states (including their common ancestor Version 0 in Figure 1.1), called three-way merging. Two-way merging cannot identify deletions since the common original state is unknown. A state-based comparison requires a match function which determines whether two elements of the compared artifact correspond to each other. The easiest way to match two elements is to search for completely equivalent elements. However, the quality of the match function is crucial for the overall quality of the merge approach. Therefore, especially graph-based merge approaches often use more sophisticated matching techniques based on identifiers and heuristics (cf. [KN06] for an overview of matching techniques). Model matching, or more generally the graph isomorphism problem is NP-hard (cf. [KR96]) and, therefore, very computation intensive. If the match function is capable of matching also partially different elements, a difference function is additionally required to determine the fine-grained differences between two corresponding elements. Having these two functions, two states of the same artifact may be merged with the algorithm shown in Algorithm 2.1. Note that this algorithm only serves to clarify conceptually basic state-based merging. This algorithm is applicable for both text-based and graph-based merging, whereas n_X denotes the atomic element n within the product space of Version X; that is, n_o for an element in the common origin version and n_1 or n_2 for an element in the two revised versions, respectively.

In line 1 of Algorithm 2.1, the merged version V_m is initialized by creating a copy of V_o. Then, it iterates through each element n_o in the common origin version V_o of a software artifact. In line 3 and 4, the elements matching with n_o are retrieved from the two modified versions V_{r1} and V_{r2}. However, there might be no match for n_o in V_{r1} or V_{r2} because n_o might have been removed. If n_o has a match in both versions V_{r1} and V_{r2} (cf. line 5), the algorithm checks whether n_o has been modified in the versions V_{r1} and V_{r2}. If the matching element, either n_1 or n_2, is different from the original element n_o (i.e., it has been modified) in one and only one of the two versions V_{r1} and V_{r2}, the modified element is used for creating the merged version (cf. line 7 or line 10). If, however, the matching element is different in *both versions*, an update-update conflict is raised by the algorithm (cf. line 13). If the matching element has not been modified at all, the original element n_o can be left as it is in the merged version (cf. line 16). Next, the algorithm checks if there is no match for n_o in one of the two modified versions (i.e., it has been removed). If so, the algorithm determines whether it has been concurrently modified and raises, in this case, a delete-update conflict (cf. line 20 and line 24). If the element has not been concurrently modified, it is removed from the merged version (cf. line 21 and line 25). The element n_o is also removed, if there is no match in both modified versions; that is, it has been deleted in both versions (cf. line 28). Finally, the algorithm adds all elements from V_{r1} and V_{r2} that have no match in the original version V_o and, consequently, have been added in V_{r1} or V_{r2} (cf. line 32 and line 35).

Operation-based merging does not operate on the states of an artifact. Instead, the operation sequences which have been concurrently applied to the original version are recorded and analyzed. Since the operations are directly recorded by the applied editor, operation-based approaches may support, besides recording atomic operations, also to record composite oper-

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

```
1  V_m ← V_o // Initialize V_m with the contents of V_o
2  foreach n_o ∈ V_o do
3      n_1 ← match(n_o in V_r1)
4      n_2 ← match(n_o in V_r2)
5      if hasMatch(n_o in V_r1) ∧ hasMatch(n_o in V_r2) then
6          if diff(n_o, n_1) ∧ ¬ diff(n_o, n_2) then
7              Replace n_o with n_1 in V_m
8          end
9          if ¬ diff(n_o, n_1) ∧ diff(n_o, n_2) then
10             Replace n_o with n_2 in V_m
11         end
12         if diff(n_o, n_1) ∧ diff(n_o, n_2) then
13             Raise update-update conflict
14         end
15         if ¬ diff(n_o, n_1) ∧ ¬ diff(n_o, n_2) then
16             Leave n_o as it is in V_m
17         end
18     end
19     if hasMatch(n_o in V_r1) ∧ ¬ hasMatch(n_o in V_r2) then
20         if diff(n_o, n_1) then Raise delete-update conflict
21         else Remove n_o in V_m
22     end
23     if ¬ hasMatch(n_o in V_r1) ∧ hasMatch(n_o in V_r2) then
24         if diff(n_o, n_2) then Raise delete-update conflict
25         else Remove n_o in V_m
26     end
27     if ¬ hasMatch(n_o in V_r1) ∧ ¬ hasMatch(n_o in V_r2) then
28         Remove n_o in V_m
29     end
30 end
31 foreach n_1 ∈ V_r1 do
32     if ¬ hasMatch(n_1 in V_o) then Add n_1 to V_m
33 end
34 foreach n_2 ∈ V_r2 do
35     if ¬ hasMatch(n_2 in V_o) then Add n_2 to V_m
36 end
```

Algorithm 2.1: State-based Merge Algorithm

ations, such as refactorings (e.g., [KHWH10]). The knowledge on applied refactorings may significantly increase the quality of the merge as stated by Dig et al. [DMJN08]. The downside of operation recording is the strong dependency on the 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. [LvO92]). Consequently, a decision procedure for commutativity is required. Such decision procedures are not necessarily trivial. In the simplest yet least efficient form, each pair of operations within the cross product of all atomic operations in both sequences are applied in both possible orders to the artifact and both results are checked for equality. If they are not equivalent, the operations are not commutative. After checking for commutativity, operation-based merge approaches apply all non-conflicting (commutative) operations of both sides to the common ancestor in order to obtain a merged model.

In comparison to state-based approaches, the recorded operation sequences are, in general, more precise and potentially allow for gathering more information (e.g., change order and refactorings), than state-based differencing. In particular, state-based approaches do not rely on a precise matching technique. Moreover, state-based comparison approaches are—due to complex comparison algorithms—very expensive regarding their run-time in contrast to operation-based change recording. However, these advantages come at the price of strong editor-dependence. Furthermore, one part of the computational complexity which was saved in contrast to state-based matching and differencing is lost again due to operation sequence cleansing and non-trivial checking for commutativity. Nevertheless, operation-based approaches scale for large models from a conceptual point of view because their computational effort mainly depends on the length of the operation sequences and—in contrast to state-based approaches—not on the size of the models [KHWH10].

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

Selected Representatives

In Figure 2.1, we show 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 2.1. For a more detailed description of existing model versioning approaches we kindly refer to Section 2.1.2.

The combination of *text-based and state-based merge approaches* are probably the most adopted ones in practice. For instance, traditional central version control systems, such as CVS[1] and SVN[2], use state-based three-way merging of flat text files. The smallest indivisible unit of merging in these systems is usually a *line* within a text file, as is the case for the Unix *diff* utility [HM76]. Lines are matched across different versions by searching for the Least Common Sub-sequence (LCS). For efficiency, usually only *completely equal* lines are matched and, therefore, no dedicated difference function for deriving the actual difference between two lines is required: A line is simply either matched and, therefore, equal or unmatched and, therefore, considered to be added or removed at a certain position in a text file. Consequently, parallel modifications to *different* lines can be merged without user intervention as long as they are at different positions. As soon as the same line is modified in both versions (Version 1 and Version 2) or modified and concurrently deleted, a conflict is annotated in the merged file. As stated earlier, due to their syntax and semantics unawareness, compile-time and run-time errors might be introduced by the merge. The same applies to the distributed version control systems (DVCS) git[3] and bazaar[4], since they are also state-based and line-based. The major difference to SVN and CVS is their distributed nature. DVCS disclaim a single central repository and take a peer-to-peer approach instead. Developers commit their operations 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. Operations might, therefore, be grouped into *atomic commits* and pushed to other peers more easily which is a step towards operation-based merging.

MolhadoRef [DMJN08], a representative for *text- and operation-based approaches*, aims at improving the merge result by 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 2.1, we cite two representatives for graph-based and state-based approaches—one for source code, namely *JDiff* [AOH07], and one for software models,

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

namely *EMF Compare*[5] [BP08]. JDiff is a graph-based differencing approach for Java source code. Corresponding classes, interfaces and methods are matched by their qualified name or signature. This matching also accounts for the possibility to interact with the user in order to improve the match of renamed but still corresponding elements due to the absence of unique identifiers. For matching and differencing the method bodies, the approach builds enhanced control-flow graphs representing the statements in the bodies and compares them. Thereby, JDiff can provide information that accurately reflects the effects of code operations on the program at the statement level. EMF Compare is a model comparison framework for EMF based models. It applies 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 operations directly and apply merging on a *graph-based representation*. The first paper, which introduced operation-based merging was published by Lippe and Oosterom [LvO92]. They propose to record all operations 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 operations to the common ancestor version. In general, a pair of operations is conflicting if they are not commutative. *EMFStore* [KHWH10] is an operation- and graph-based versioning system for software models. Since EMF Compare and EMFStore are representatives of model versioning systems, they are further elaborated on in Section 2.1.2.

Consequences of Design Decisions

To highlight the benefits and drawbacks of the four possible combinations of the versioning approaches based on Figure 2.1, we present a small versioning example depicted in Figure 2.2 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 2.2 and defined by the EBNF-like Xtext[6] grammar specified in the box labeled *Grammar*. The same language and the same examples are depicted in terms of graphs in the lower part of Figure 2.2. 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 operations in Version 1 and Version 2 are highlighted with bold fonts or edges in Figure 2.2. The first user changes the name of the class Human to Person, sets the lower bound of the property carNo to 1 (because every car must have exactly one number) and adds an explicit reference owns to Person. Concurrently, the second user renames the property carNo to regId and the class Vehicle to Car.

[5] http://www.eclipse.org/emft/projects/compare
[6] http://www.eclipse.org/Xtext

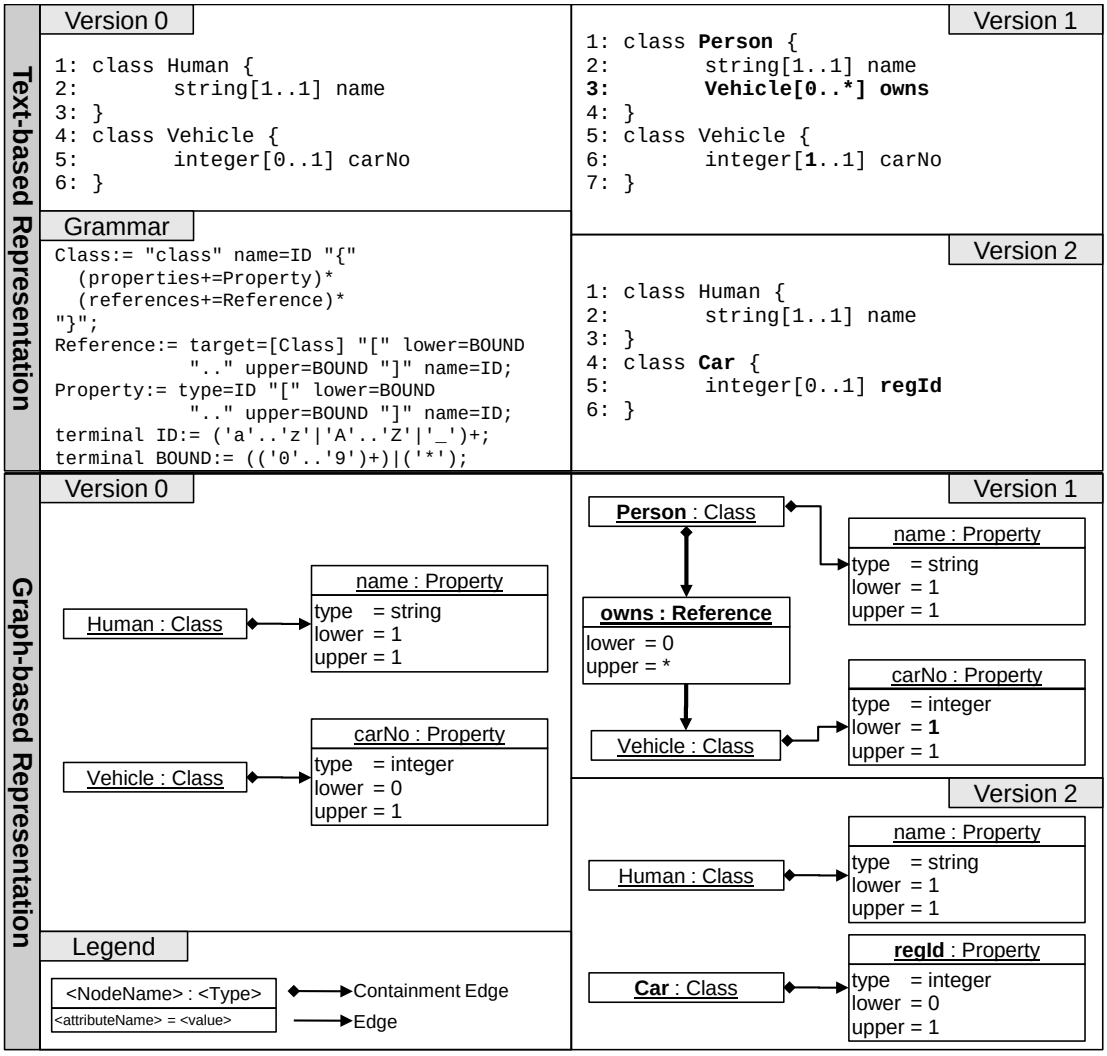
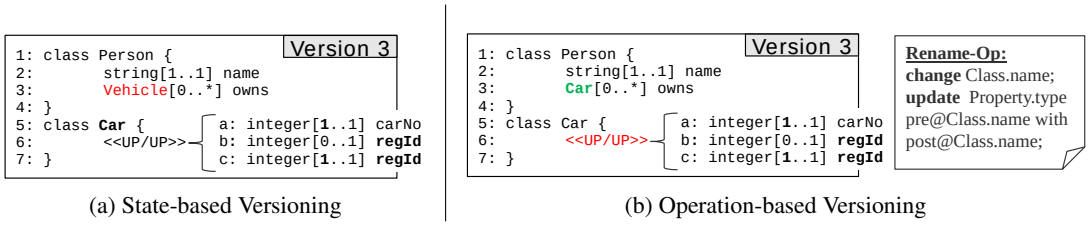


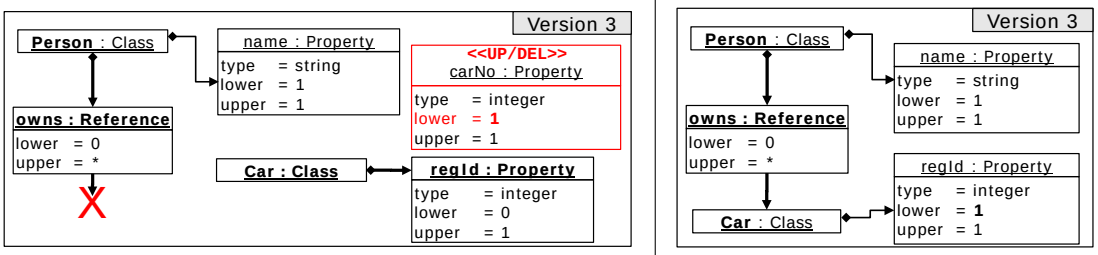
Figure 2.2: Versioning Example

Text-based versioning. When merging this example with *text- and state-based* approaches (cf. Figure 2.3a 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, and bazaar), the first line is correctly merged since it has only been modified in Version 1 and remained untouched in Version 2 (cf. Algorithm 2.1). The same is true for the added reference in line 3 of Version 1 and the renamed class Car in line 4 of Version 2. However, the property carNo shown in line 5 in Version 0 has been changed in both Versions 1 (line 6) and Version 2 (line 5). Although different features of this property have been modified (lower and name), these modifications result in a concurrent change of the same line and, hence, a conflict is raised. Furthermore, the reference added in Version 1 refers to class *Vehicle*, which does not exist in the merged version anymore since it has been renamed in Version 2. We may summarize that text- and state-based merging



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

Figure 2.3: Text-based Versioning Example



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

Figure 2.4: Graph-based Versioning Example

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 2.3b.

Graph-based versioning. Applying the merge on top of the *graph-based representation* depicted in Figure 2.2 may also significantly improve the merge result because the representation used for merging is a node in a graph which more precisely represents the versioned software artifact. However, as already mentioned, this advantage comes at the price of language dependence because merging operates either on the language specific graph-based representation or a translation of a language to a generic graph-based structure must be available. *Graph- and state-based approaches* additionally require a match function for finding corresponding nodes and a difference function for explicating the differences between matched nodes. The preciseness of

the match function significantly influences the quality of the overall merge. Assume matching is based on name and structure heuristics for the example in Figure 2.2. Given this assumption, the class Human may be matched since it contains an unchanged property name. Therefore, renaming the class Human to Person can be merged without user intervention. However, heuristically matching the class Vehicle might be more challenging because both the class and its contained property have been renamed. If the match does not identify the correspondence between Vehicle and Car, Vehicle and its contained property carNo is considered to be removed and Car is assumed to be added in Version 2. Consequently, a delete-update conflict is reported for the change of the lower bound of the property carNo in Version 1. Also the added reference owns refers to a removed class which might be reported as conflict. This type of conflict is referred to as *delete-use* or *delete-reference* in literature [TELW10, Wes10]. If, in contrast, the match relies on unique identifiers, the nodes can soundly be matched. Based on this precise match, the state-based merge component can resolve this issue and the added reference owns correctly refers to the renamed class Car in the merged version. However, the concurrent modification of the property carNo (name and lower) might still be a problem because purely state-based approaches usually take either the entire element from either the left or the right version to construct the merged version. Some state-based approaches solve this issue by conducting a more fine-grained difference function to identify the detailed differences between two elements. If these differences are not overlapping—as in our example—they can both be applied to the merged element. The result of a graph-based and state-based merge without taking identifiers into account is visualized in Figure 2.4a.

Purely *graph- and operation-based approaches* are capable of automatically merging the presented example (cf. Figure 2.4b). Between Version 0 and Version 1, three operations have been recorded, namely the rename of Human, the addition of the reference owns and the update concerning the lower bound of carNo. To get Version 2 from Version 0, class Vehicle and property carNo have been renamed. All these atomic operations do not interfere, i.e., they are commutative, and therefore, they all can be re-applied to Version 0 in order to obtain a correctly merged version.

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

2.1.2 State of the Art in Model Versioning

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

Features of Model Versioning Approaches

In the following, we surveying techniques applied for detecting operations applied between two versions of a model, as well as on the techniques used for detecting conflicts among those operations. Furthermore, we reveal whether these approaches are specifically tailored to a certain modeling language or whether they are generic, in the sense that they are applicable for all modeling languages that are defined in terms of a common meta-metamodel. If they are generic, we further investigate their adaptability to language-specific aspects. Particularly, we consider the following features.

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

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

Operation recording versus model differencing. As already introduced in Section 2.1.1, we may distinguish between approaches that obtain operations performed between two versions of a model by applying *operation recording* or by *model differencing*. If an approach applies model differencing, which is, in general, more flexible concerning the adopted modeling editors, it is substantial to consider the techniques conducted in the match function for identifying corresponding model elements because the quality of the match is crucial for an accurate subsequent operation detection. We may distinguish between match functions that rely on universally unique IDs (*UUIDs*), and those applying heuristics based on the model element's *content* (i.e., feature values and contained child elements). Relying on UUIDs, even intensively modified model elements can still be matched very efficiently. However, relying on UUIDs only, model elements that have been concurrently added by two developers will obviously not have a comparable UUID although they potentially should be identified as corresponding when considering their contents. The same applies to deleted and newly added model elements having the same content as the beforehand deleted model element. When completely neglecting the contents of model elements, which is the case when only UUIDs are used, important matches might

be missed. Thus, it would be beneficial to *combine* UUID-based and content-based matching. To summarize, we distinguish between operation recording, model differencing, and in case modeling differencing is applied, whether UUIDs, content-based heuristics or both are used for detecting corresponding model elements.

Composite operation detection. The knowledge on applied composite operations is the prerequisite for considering them in the merge process. Therefore, it is a distinguished feature whether an operation detection component is also capable of detecting applications of composite operations besides only identifying atomic operations. It is worth noting that, in case of model differencing, the state-based a posteriori detection of composite operation applications is highly challenging as stated in Section 6 of [DCMJ06].

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

Detection of conflicts between atomic operations. One key feature of model versioning systems is, of course, their ability to detect conflicts arising from contradictory operations applied by two developers in parallel. Consequently, we first investigate whether the approaches under consideration are capable of detecting conflicts between contradictory atomic operations. Such conflicts occur between two atomic operations, for instance, if one developer updates a feature value of a model element whereas the other developer concurrently deletes the same model element. This type of conflict is often referred to as delete-update conflict in literature [BKL+11a, TELW10, Wes10]. Also some other types of conflicts between atomic operations have been introduced in literature, such as update-update conflicts and delete-use conflicts. In this survey, we do not precisely examine *which* types of conflicts are supported. We rather investigate whether conflicts arising from contradictory atomic operations are considered at all.

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

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

Adaptability of the conflict detection. According to the evaluation concerning the adaptability of the operation detection approach in generic model versioning systems, we also review the adaptability to language-specific aspects of the conflict detection approach. This involves techniques to configure language-specific conflict types that can not be covered by a solely generic analysis of the obtained operations.

Evaluation Results

In this section, we introduce current state-of-the-art model versioning systems and evaluate them on the basis of the features discussed in the previous section. The considered systems and the findings of this survey are summarized in Table 2.1 and discussed in the following. Please note that the order in which we introduce the considered systems is alphabetically and has no further meaning.

ADAMS. The "Advanced Artifact Management System" (ADAMS) offers process management functionality, supports cooperation among multiple developers, and provides artifact versioning [DLFOT06]. ADAMS can be integrated via specific plug-ins into modeling environments to realize versioning support for models. In [DLFST09], De Lucia et al. present an ADAMS plug-in for ArgoEclipse[7] to enable version support for ArgoUML models. Because artifacts are stored in a proprietary ADAMS-specific format to be handled by the central repository, models have to be converted into that format before they are sent to the server and translated back to the original format, whenever the model is checked out again. ADAMS applies state-based model differencing based on UUIDs. Added model elements, which, as a consequence, have no comparable UUIDs, are matched using simple heuristics based on the element names to find corresponding elements concurrently added by another developer. The differences are computed at the client and sent to the ADAMS server, which finally performs the merge. The ADAMS plug-in for models is specific to a ArgoUML models. A specific translation has to be provided for each supported model type to allow ADAMS to process these models. Interestingly, ADAMS can be customized to a certain extent. For instance, it is possible to customize the unit of comparison; that is, the smallest unit, for which, if concurrently modified, a conflict is raised. In [DLFST09], it is also mention that the conflict detection algorithm may be customized for specific model types with user-defined *correlation rules*, which specify when two operations should be considered as conflicting. However, it remains unclear, how these rules are exactly specified and how these rules influence the conflict detection. The implementation promoted in this publication is not available to further review this interesting customization feature. Composite operations and state-based conflicts are not supported.

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

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

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

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

Approach by Alanen and Porres. One of the earliest works on versioning UML models was published by Alanen and Porres [AP03], who presented metamodel independent algorithms for difference calculation, model merging, as well as conflict resolution. They identified seven elementary operation types a developer may perform to modify a model. For calculating the differences between the original version and the modified version, first a match between model elements is computed based on UUIDs. Based on this match, created, deleted, and changed elements are identified. Alanen and Porres provide an algorithm to compute a union of two sets of operations whereas also merging values of ordered features are considered. The proposed algorithms are specific to UML models and do not allow for any customization or treatment of

composite operations. Still, their algorithms serve as a fundamental and influential work for many other researchers in the area of model versioning.

Approach by Cicchetti, Di Ruscio, and Pierantonio. Cicchetti et al. [CDRP08] present an approach to specify and detect language-specific conflicts arising from parallel modifications. Their work does not address the issue of obtaining differences, but proposes a model-based way of representing them. Howsoever the differences are computed, they are represented by instantiating an automatically generated language-specific difference metamodel. Conflicts are specified by manually created conflict patterns. These conflict patterns are represented in terms of a model of difference elements, which are reported as conflict whenever found in the combination of two difference models. To this end, a hand-crafted set of language-specific conflict patterns, represented as forbidden difference patterns, can be established to create a dedicated conflict detection system. Thereby, the realization of a customizable conflict detection component is possible. The authors also allow to specify reconciliation strategies to specific conflict patterns. Although the authors do not discuss how differences and applications of composite operations are obtained, their approach supports also conflicts caused by composite operations. It seems to be a great deal of work to establish a complete set of conflict patterns for a specific language; nevertheless, in the end, a highly customized conflict detection can be achieved.

CoObRA. The Concurrent Object Replication framework CoObRA developed by Schneider et al. [SZN04] realizes optimistic versioning for the UML case tool Fujaba[8]. CoObRA records the operations performed on the model elements and stores the recorded operations in a central repository. Whenever other developers update their local models, these operations are fetched from this repository and replayed locally. To identify equal model elements, unique identifiers are used. Conflicting operations are not applied (also the corresponding local change is undone) and finally presented to the user who has to resolve these conflicts manually. In [SZ07], the authors also shortly discuss state-based conflicts in terms of inconsistencies. CoObRA is capable of detecting a small subset of such conflicts when the underlying modeling framework rejects the execution of a certain operation. For example, a class cannot be instantiated anymore if the respective class has been concurrently deleted. However, for instance, concurrent additions of an equally named class is not reported as conflict. The authors also shortly mention composite operations in terms of a set of atomic operations grouped into commands. The operation recording component seems to be capable of grouping atomic operations into commands to allow for a more comprehensible undo mechanism. In particular, one command in the modeling editor might cause several atomic operations in the log; if the user aims to undo the last change, the complete command is undone and not only the latest atomic change. In their papers, however, no special treatment of these commands in the merge process is mentioned.

DSMDiff. In [LGJ07], the authors pointed out the urgent need for language-independent model differencing when domain-specific modeling languages are adopted. Therefore, a metamodel-independent differencing tool, named DSMDiff, is proposed, which makes no assumptions on

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

the editors used for modifying the models. To also allow for comparing models that are not subsequent versions, the proposed algorithm refrains from relying UUIDs. Instead, correspondences between model elements are obtained from signature and structural matching. Having obtained the corresponding model elements, the differences are computed by traversing through the model and comparing the corresponding elements with each other. DSMDiff supports only a two-way comparison and is, consequently, not directly designed to detect merge conflicts. DSMDiff is tailored to be completely generic: it's applied heuristics work for all domain-specific languages, but cannot be adapted with language-specific match rules. Furthermore, it does not support detecting applications of composite operations.

EMF Compare. The open-source model comparison framework EMF Compare [BP08], which is part of the Eclipse Modeling Framework Technology (EMFT) project[9], supports generic model comparison and model merging. EMF Compare provides two-way and three-way model comparison algorithms for EMF-based models. As for instance with DSMDiff, EMF Compare's model comparison algorithm consists of two phases, a matching phase and a differencing phase. The matching phase aims at establishing one-to-one correspondences between model elements in the original model and the revised models. For this, EMF Compare supports either UUID-based matching or content-based matching, which applies a combination of four heuristics: type, name, value, and relationship similarity. However, the combination of UUIDs and heuristics are not directly supported. Based on the established match, the differencing phase computes the differences between all corresponding model elements. The model element correspondences and differences are represented by a match model and a difference model, respectively. Additionally, EMF Compare provides a merge service, which is capable of applying difference elements in a difference model to allow for merging models. It also offers basic conflict detection capabilities and user interfaces for displaying match and difference models. All these features of EMF Compare are generic; consequently, they can be applied to any EMF-based model irrespectively of the modeling language these models conform to. However, EMF Compare can be extended programmatically for language-specific matching and differencing. Thus, it is not adaptable in the sense that it can be easily configured for a specific language, but it constitutes a programmatically extensible framework for all tasks related to model comparison.

EMFStore. The model repository EMFStore, presented by Koegel et al. [KHWH10], has been initially developed as part of the Unicase[10] project and provides a dedicated framework for model versioning of EMF models. After a copy of a model is checked out, all operations applied to this copy are tracked by the modeling environment. Once all modifications are done, the recorded operations are committed to a central repository. For recording the operations, a framework called *Operation Recorder* [HK10] is used. This framework exploits the the EContentAdapter and the *EMF Command Framework* for listening and saving all applied operations. Thereby, modifications performed in every EMF-based editor can be recorded. Also transactions (i.e., a series of dependent operations) can be tracked and grouped accordingly. Having two lists of the recorded operations, in particular, the list of uncommitted local operations and

[9] `http://www.eclipse.org/modeling/emft`
[10] `http://www.unicase.org`

the list of new operations on the server since the last update, relationships among those operations are established, in particular, the *requires relationship* and the *conflicts relationship*. The former relationship expresses dependencies between operations, the later indicates contradicting modifications. As the exact calculation of these relationships requires expensive computations, heuristics are applied to obtain an approximation for setting up those relationships. The conflict detection component classifies two operations as conflicting, if the same attribute or the same reference is modified. Furthermore, the authors introduce levels of severity to classify conflicts. They distinguish between hard conflicts and soft conflicts referring to the amount of user support necessary for their resolution. Whereas hard conflicts do not allow including both conflicting operations within the merged model, for soft conflicts this is possible (with the danger of obtaining an inconsistent model). Summarizing, EMFStore is completely operation-based; that is, the actual model states are never considered for detecting conflicts. This also entails that a removed and subsequently re-added model element is treated as a new model element so that all concurrent operations to the previously removed element are reported as conflict. Composite operations can be recorded and saved accordingly. In the conflict detection, however, composite operations are not specifically treated. If an atomic change within a composite operation conflicts with another change, the complete transaction is indeed marked as conflicting; the intentions behind composite operations, as well as potentially violated preconditions of composite operations are not specifically considered.

Approach by Gerth et al. Gerth et al. [GKLE10] propose a conflict detection approach specifically tailored to the business process modeling language (BPMN) [OMG09]. To identify the differences between two process models (cf. [KGFE08]), in a first step, a mapping between corresponding elements across two versions of a process model is computed based on UUIDs which are attached to each element. In the next step, for each element that has no corresponding counterpart in the opposite version, a operation is created representing the addition or deletion. The resulting operations are specific to the type of the added or deleted element (e.g., InsertAction or DeleteFragment). Finally, this list of operations is hierarchically structured according to the fragment hierarchy of the process model in order to group those atomic operations into so-called compound operations. Consequently, these compound changes group several atomic operations into composite additions or deletions. Having identified all differences in terms of operations between two process models, syntactic, as well as semantic conflicts among those concurrent operations can be identified using a term formalization of process models. According to their definitions, a syntactic conflict occurs if an operation is not applicable after another operation has been performed. A semantic conflict is at hand whenever two operations modify the same elements so that the process models are not "trace equivalent"; that is, all possible traces of a process model are not exactly equal. Obviously, rich knowledge on the operational semantics of process models has to be encoded in the conflict detection to be able to reveal semantic conflicts. Although the authors presented an efficient way of detecting such conflicts, no possibility to adapt the operation detection and conflict detection mechanisms to other languages is foreseen.

Approach by Mehra, Grundy, and Hosking. The publication by Mehra et al. [MGH05] mainly focuses on the graphical visualization of differences between versions of a diagram. Therefore, they provide a plug-in for the meta-CASE tool *Pounamu*, a tool for the specification and generation of multi-view design editors. The diagrams created with this tool are serialized in XMI and are converted into an object graph for comparison. In their proposed comparison algorithm, the differences are obtained by applying a state-based model differencing algorithm, which uses UUIDs to map corresponding model elements. The obtained differences are translated to Pounamu editing events, which are events corresponding to the actions performed by users within the modeling environment. Differences cover not only modifications performed on the model, but also modifications performed on the graphical visualization. The differences between various versions are visualized in the concrete syntax so that developers may directly accept or reject modifications on top of the graphical representation developers are familiar with. In their works, also conflict detection facilities are shortly mentioned. However, this aspect seems not to be the primary focus of the approach and, consequently, is not elaborated in more detail. Composite operations are not considered at all.

Approach by Oda and Saeki. Oda and Saeki [OS05] propose to also generate versioning features along with the modeling editor generated from a specified metamodel as known from metamodeling tools. The generated *versioning-aware* modeling editors are capable of recording all operations applied by the users. In particular, the generated tool records operations to the logical model (i.e., the abstract syntax tree of a model), as well as the diagram's layout information (i.e., the concrete syntax). Besides recording, the generated modeling tool includes check in, check out, and update operations to interact with a central model repository. It is worth noting that only the change sequences are sent to the repository and not the complete model state. In case a model has been concurrently modified and, therefore, needs to be merged, conflicts are identified by re-applying all recorded operations to the common ancestor version. Before each change is performed in the course of merging, its precondition is checked. In particular, the precondition of each change is that the modified model element must exist. Thereby, delete-update conflicts can be identified. Update-update conflicts, however, remain unrevealed and, consequently, the values in the resulting merged model might depend on the order in which the recorded updates are applied because one update might overwrite another previous update. Composite operations and their specific preconditions are not particularly regarded while merging. The approach also does not enable to specify additional language-specific conflicts. Although metamodel violations can, in general, be checked in their tool, they are not particularly considered in the merge process. Because the versioning tool is generated from a specific metamodel, the generated tool is language dependent; the approach in general, however, is independent from the modeling language. However, the approach obviously forces users to use the generated modeling editor to be able to use their versioning system.

Odyssey-VCS 2. The version control system Odyssey-VCS by Oliveira et al. [OMW05] is dedicated to versioning UML models. Operations between two versions of a model are identified by applying state-based model differencing relying on UUIDs for finding corresponding model elements. Language-specific heuristics for the match functions may not be used. Also

language-specific composite operations are neglected. Interestingly, however, for each project, so-called behavior descriptors may be specified, which define how each model element should be treated during the versioning process. Consequently, the conflict detection component of Odyssey-VCS is adaptable, in particular, it may be specified which model elements should be considered to be atomic. If an atomic element is changed in two different ways at the same time, a conflict is raised. These behavior descriptors (i.e., adaptations) are expressed in XML configuration files. Thus, Odyssey-VCS is customizable for different projects concerning the unit of comparison, as well as whether to apply pessimistic or optimistic versioning. Conflicts coming from language-specific operations, as well as additional language-specific conflicts, however, may not be configured. Odyssey-VCS may be used either with a standalone client or with arbitrary modeling tools. More recently, Odyssey-VCS 2 [MCPW08] has been published, which is capable of processing any EMF-based models and not only UML models. A validation of the resulting merged model is not considered.

Approach by Ohst, Welle, and Kelter. Within the proposed merge algorithm, also Ohst et al. [OWK03] put special emphasis on the visualization of the differences. Therefore, differences between the model as well as the layout of the diagram are computed by applying a state-based model differencing algorithm relying on UUIDs. Conflict detection, however, is not discussed in detail; only update-update and delete-update conflicts are shortly considered. After obtaining the differences, a preview is provided to the user, which visualizes all modifications, even if they are conflicting. The preview diagram can also be modified and, therefore, allows users to resolve easily conflicts using the concrete syntax of a diagram. For indicating the modifications, the different model versions are shown in a unified document containing the common parts, the automatically merged parts, as well as the conflicts. For distinguishing the different model versions, coloring techniques are used. In the case of delete-update conflicts, the deleted model element is crossed out and decorated with a warning symbol to indicate the modification.

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

SMOVER. The semantically-enhanced model versioning system by Reiter et al. [RAB+07], called SMOVER, aims at reducing the number of falsely detected conflicts resulting from syntactic variations of semantically equal modeling concepts. Furthermore, additional conflicts shall be identified by incorporating knowledge on the modeling language's semantics. This knowledge is encoded by the means of model transformations which rewrite a given model to so-called

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

semantic views. These semantic views provide a canonical representations of the model, which makes certain aspects of the modeling language more explicit. Consequently, also potential semantic conflicts might be identified when the semantic view representations of two concurrently evolved versions are compared. It is worth noting that the system itself is independent from the modeling language and language-specific semantic views can be configured to adapt the system to a specific modeling language. The differences are identified using a state-based model differencing algorithm based on UUIDs. Therefore, the system is independent of the used modeling editor. However, this differencing can not be adapted to specific modeling languages and only works in a generic manner. SMOVER also only addresses detecting conflicts regarding the semantics of a model and does not cover syntactic operation-based conflicts.

Approach by Westfechtel. Recently, Westfechtel [Wes10] presented a formal approach for merging EMF models. Although no implementation of his work is available, it provides well-defined conflict rules based on set-theoretical conflict definitions. In this paper, Westfechtel does not address the issue of identifying differences between model versions and rather focuses on conflict detection only and assumes the presence of change-based differences that can be obtained by, for instance, EMF Compare. Westfechtel's approach is directly tailored to EMF models and defines *context-free merge rules* and *context-sensitive merge rules*. Context-free merge rules determine "the set of objects that should be included into the merged versions and consider each feature of each object without taking the context [i.e., relationships to other objects] into account" [Wes10]. The presented algorithm also supports merging of ordered features and specifies when to raise update-update conflicts. In contrast to context-free merging, context-sensitive merge rules also consider containment conflicts, delete conflicts, and reference conflicts. Containment conflicts occur, in particular, if an object in the merged model has no unique container, or if the merged model comprises cyclic containment structures, or if a dangling object (i.e., an object having no parent except for the root object) exists. Delete conflicts occur if an object has been deleted and concurrently modified, or if an object has been deleted and concurrently added as a reference value in another object, or if an object has been deleted and concurrently moved. Finally, reference conflicts concern inconsistent operations to bi-directional references. Besides these conflicts, Westfechtel also addresses state-based conflicts arising from the well-formedness rules of EMF. However, no techniques that enable further language-specific constraints are discussed. Moreover, he only addresses conflicts among atomic operations and is not adaptable to language-specific knowledge.

Summary

After surveying existing model versioning approaches, we may conclude that the predominant strategy is to apply state-based model differencing and generic model versioning. The majority of model differencing approaches rely on UUIDs for matching. However, only ADAMS combines UUIDs and (very simple) content-based heuristics. The detection of applications of composite operations is only supported by approaches applying operation recording. The only approach that is capable of detecting composite operations by using a state-based model comparison approach is Gerth et al.; however, their approach is specifically tailored to process models and the supported composite operations are limited to compound additions and deletions.

Consequently, none of the surveyed generic approaches is capable of detecting applications of more complex composite operations having well-defined pre- and postconditions without directly recording their application in the editor. Furthermore, none of the approaches are adaptable in terms of additional match rules or composite operation specifications. EMF Compare and EMFStore foresee at least an interface to be implemented in order to extend the set of detectable applications of composite operations. In EMF Compare, however, the detection algorithm has to be provided by an own implementation. In EMFStore, additional commands may be plugged into the modeling editor programmatically for enabling EMFStore to record them.

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

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

State-based conflicts have not gained much attention in the model versioning community yet. CoObRA is capable of detecting at least a subset of all potentially occurring violations of the modeling language's rules. Westfechtel only addresses the basic well-formedness rules coming from EMF, such as spanning containment tree. Only Gerth et al., Oda and Saeki, and the RSA perform a full validation after merging.

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

Consequently, SMOVER only aims to detect a very specific subset of conflicts and can be seen as orthogonal to existing model versioning systems.

2.2 Software Adaptation

This thesis proposes an *adaptable* model versioning system that provides extension points to be used for adapting the system's behaviour to specific modeling languages, as well as to recurrently applied composite operations being considered in the merge process. Therefore, we consider existing work in the domain of software adaptation. Software adaptation, however, is a large research domain on its own; thus, we provide only a brief overview on the terminology and basic concepts in this section.

Although extensive research in the domain of requirements engineering has lead to well-defined systematic processes to determining efficiently and precisely the needs of potential users, it is impossible to anticipate fully the requirements of *all different future users* and to foresee *every potential change* in the environment in which the software operates. As a consequence, approaches are needed to *adapt* the behaviour of software systems as efficiently as possible.

The term *adaptation* is defined by the Merriam-Webster dictionary[12] as the adjustment to environmental conditions or a modification of an organism that improves its fitness under the conditions of its environment. Correspondingly, in the domain of software systems, adaptation refers to the modification of a system to satisfy new requirements and changing circumstances [TMD09]. We also refer to [And05] for a detailed discussion of the meanings of the terms "adaptability", "adaptation", and "flexibility". The reasons why a system has to be adopted are manifold. By the adaption of a software one may realize corrective changes, such as, for instance, bug fixes, a modification to the functional requirements such as, adding new or changing existing features, changes to the non-functional properties of a system, or improvements concerning changed operating environments [TMD09]. According to Oppermann et al. [Opp05], we may distinguish between *adaptive* and *adaptable* systems, which are complementary to each other. Adaptivity refers to the ability of an *adaptive system* to itself adapt automatically and autonomously to changing conditions, which is also referred to as *self-adaptation* [CdLG+09]. In contrast, adaptability refers to the ability of an *adaptable system* to be actively changed by its stakeholders in order to improve its functioning for specific use cases or environments. Oppermann et al. [OR97] describes the whole spectrum from adaptive to adaptable systems as depicted in Figure 2.5. This spectrum ranges from adaptive systems, in which the stakeholder has no control over performed adaptations, via systems, in which stakeholders may choose from a set of suggested adaptations through to a adaptable system, in which a stakeholder has to initiate actively the adaptation on her own.

2.2.1 Adaptive Systems

Adaptive systems are capable of adjusting their behaviour in response to their perception of the environment and the system itself [CdLG+09]. The concept of adaptivity has been applied in many research domains, such as adaptive user interfaces, autonomic computing, embed-

[12] `http://www.merriam-webster.com/dictionary/adaptation`

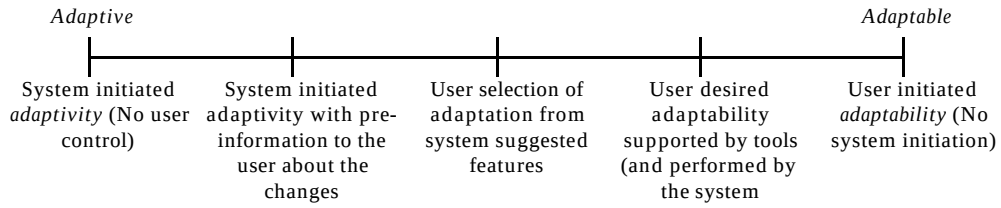


Figure 2.5: Spectrum of Adaptation by Oppermann [OR97]

ded systems, autonomous robots, and service-oriented architectures. As stated by Cheng et al. in [CdLG+09], there is a lack of consensus among researchers and practitioners on the points of variation among adaptive systems. Therefore, Cheng et al. identified four variation points referred to as modeling dimensions, which are shortly described in the following.

Adaptive systems may vary, firstly, in terms of *goals* that they aim to achieve. Goals can either refer to self-adaptability aspects, or to the middleware, or the infrastructure that supports the adaptive system. Secondly, systems may vary regarding their *cause* of adaptation. These causes may, for instance, be the actors interacting with the system, the environment in which the system operates, or properties of the system itself. Thirdly, adaptive systems may differ in the *mechanism* used to react; that is, the adaptation process itself. Finally, Cheng et al. also introduces the variation point regarding the *effects* or impact of the adaptation upon the system.

In the context of this thesis, especially *model-based self-adaption* is an interesting research field. The term *models at runtime* refer to software models that are used to reason about the operating environment and runtime behaviour of a software system [BBFJ11, Nie11]. These models aim to represent abstractions of runtime phenomena, such as resource efficiency, context dependency, as well as personalization of systems. By taking advantage of these abstractions of runtime information, runtime decisions can be facilitated and better automated. Thus, runtime models may play an integral role in the management of self-adaptive systems. For more information on this research topic, we kindly refer to the yearly workshop *Models@run.time*[13].

Self-adaptation and adaptivity being model-based or not, however, is beyond the scope of this thesis. We rather aim at providing a set of extension points that can be utilized by stakeholders to adapt the behaviour of the model versioning system according to their needs. Therefore, the approaches proposed in this thesis may rather be ascribed to adaptable systems according to the classification by [OR97] depicted in Figure 2.5.

2.2.2 Adaptable Systems

Adaptable systems enable their stakeholders to modify actively the system's behaviour. Thus, after recognizing the need for a modification of an existing system, its stakeholders initiate the adaptation to improve or extend the system for addressing the stakeholder's specific requirements. Adaptable systems may vary in terms of *adaptation time*, *adaptation transparency*, and *adaptation technique*.

[13] `http://www.comp.lancs.ac.uk/~bencomo/WorkshopMRT.html`

Adaptation time. Taylor et al. [TMD09] distinguish between offline and online adaptations. The former concerns systems that are taken offline before they can be changed and which are, eventually, restarted or re-installed again after they have been adapted. Obviously, there are many scenarios in which offline adaptation is infeasible; for instance, non-stop systems, such as web services that have to run 24/7 or systems that, when restarted, loose (mental) context that cannot be saved and recreated during maintenance. Another scenario in which offline adaptations are infeasible concerns systems that are difficult to reinstall, such as software in automobiles. Therefore, these systems have to be adaptable at run time, or during they are online. Gschwind [Gsc02] further distinguishes between design-time, compile-time, and run-time adaptations.

Adaptation transparency. The transparency of an adaptation classifies adaptable systems according to how much an adaptation has to "know" about the system being adapted. In [Gsc02], Gschwind distinguishes between *black-box*, *gray-box*, and *white-box* adaptations. Black-box adaptations are not aware of the actual implementation of the system being adapted. Hence, it only interacts with interfaces or abstract definitions of the system. In the case of gray-box adaptations, the user, who performing the adaptation, does not have to understand the implementation of the system. However, the toolkits and compilers that actually perform the adaptations must be able to access the implementation of the system being adapted in order to directly modify the implementation or to use knowledge on the implementation for further optimizations. White-box adaptations refer to cases in which the user undertaking the adaptation has to know the implementation in order to be able to adapt it.

Adaptation techniques. Besides the adaptation time and transparency, we may also categorize adaptable systems according to the applied adaptation technique; that is, how an adaptation is specified and deployed. These techniques differ regarding the level of abstraction and the degree of automation. In the following, we discuss some techniques for adapting software systems. Please note that this list is not intended to be complete. We rather aim to provide a brief overview on different adaptation approaches.

Object-oriented design [Boo90, Mey88] follows the principle of *design for change* [Dij82, Par79], which enables developers to structure their software in a way to minimize the impact of future changes. Object-oriented design offers, among others, three fundamental concepts to ease future adaptations: *information hiding*, *inheritance*, and *composition*. Information hiding protects values that are intended to be only used by one class and, in combination with an interface concept, decouples two dependent implementations. As a result, changing one class has mostly no impact on classes using the class that is changed. Inheritance is a way of reusing, extending, or, in combination with polymorphism, altering the implementation invoked by other classes. Finally, composition allows to compose more complex objects from several other objects and, thereby, compose their behaviour. In this context, *design patterns* [GHJV95], such as the *Abstract Factory Pattern* or the *Strategy Pattern*, provide a solution to more flexibly change the behaviour of a system. Although object-oriented design allows to structure a software to be more easily changed in future, the implementation must be accessible and known to the

developer aiming to adapt the system. Thus, object-oriented design per se enables white-box adaptation at design time.

Architectural styles, which are the more coarse-grained counterpart to design patterns, may also foster the ability of a system's behaviour to be adapted. Taylor et al. [TMO09], proposed a conceptual framework called BASE for evaluating, comparing, and combining techniques for run-time adaption of a software system based on architectural styles. This framework differentiates techniques based upon the following four aspects of adaption: (i) the behaviour aspect, which specifies how the behavioural specification is changed, (ii) the asynchrony aspect, which indicates whether a system can continue to run while the behaviour is changed, (iii) the state aspect, which specifies how the current state of a system is changed, and finally, (iv) the execution context aspect, which concerns the influence of the adaptation on the execution runtime (e.g., the virtual machine). Popular architectural styles that have been described by the aforementioned framework, are among others, pipes and filters [SG96], the event notification architecture [Rei90], and the service-oriented architecture [Pap03] (SOA). Most of these architectural styles allow for run-time adaptation (e.g., SOA) and offer white-box adaptation; that is, they do not force users to know the internal implementation in order to adapt the system.

Frameworks [FS97] offer generic functionality in the context of a specific application that can be selectively extended by client code. Frameworks usually provide an application programming interface (API) with which client code communicates. In this regard, frameworks are also one very common way of realizing an adaptable software. Unlike software libraries, frameworks should dictate the control flow and call client code [Rie00] and not vice versa. This paradigm is referred to as *Inversion of Control* [FS97]. Clients may choose which functionality they want to extend by instantiating the respective part of the framework. Many frameworks offer a default behaviour for parts that have not been overwritten by client code. Usually, frameworks offer a white-box adaptation because only the API has to be known to the users instantiating the framework. However, frameworks traditionally allow for design-time adaptations only.

Component-based Software Engineering [HC01,KB98] (CBSE) enables software adaptation on a more coarse-grained level than with object-oriented design. Thereby, the goal of CBSE is to glue prefabricated components together to construct a new software system. Every component has well-defined interfaces through which components interact with each other. The actual component's implementation is completely hidden from other components. Consequently, components may easily be exchanged by other components having compatible interfaces as the component to be removed. This enables the adaptation of a software system by exchanging components. One popular framework realizing the component-based architecture is the *Open Services Gateway Initiative Framework* [OSG03] (OSGi). In OSGi, a component, called bundle, can register its services in a central service registry. Bundles may be deployed and exchanged at run time to allow for online adaptation. As bundles only interact with the interfaces of other bundles, CBSE can be considered as black-box adaptation technique.

Aspect-Oriented Software Development (AOSD) is a fairly young but rapidly advancing research field and adopts the idea of separating concerns, which has been originally raised by [Par72]. More precisely, AOSD aims at separating *crosscutting* concerns from traditional units of decomposition, such as class hierarchies. Crosscutting concerns are concerns that are distributed over several parts of an application. In particular, AOSD represents the convergence

of several approaches, such as adaptive programming [Lie96], composition filters [ABV92], subject-oriented programming [HO93], multi-dimensional separation of concerns [TOHS99], and aspect-oriented programming [KLM+97]. Although the primary goal of all of them is to allow for separating crosscutting concerns, all these approaches may easily be used to adapt existing software systems. For instance, if a specific behaviour of an existing system shall be adapted, developers may configure a pointcut, which specifies a join point at the place at which the behaviour to be adapted is located. Whenever the system execution reaches the join point, the additional code, called advice, specified in the pointcut is executed. In many AOSD frameworks, a specific compiler is needed that weaves the advices into the join points. Hence, only design-time adaptation is possible. However, using other techniques, such as proxy objects, also run-time adaptation can be realized. Anyway, adapting software systems using aspects, the original code of the software system being adapted must be available and known to developer; consequently, AOSD only enables white-box adaptation.

Configuration of a software system is heavily used in practice to influence a system's behaviour. Configuration, however, is a very broad term ranging from specifying simple initial settings through to extending an existing software with custom behaviour that is specified using sophisticated scripting languages [Ous98]. Depending on the used configuration language, configuration can be very powerful whereas still no deep knowledge on the implementation of the system being adapted is required (i.e., black-box adaptation). However, developers specifying the adaptation must be aware of the configuration language. Such languages often use a generic textual syntax such as XML [W3C08] or YAML [BKnIN09] and usually are specific to the system being adapted; only the system to be adapted is able to interpret the adaptations. Thus, these languages are comparable to domain-specific modeling languages (DSML) to a certain extent, especially, if the language is defined by a dedicated schema language, such as XML Schema [W3C09], which would correspond, in terms of DSMLs, to a metamodel.

2.3 Model Transformation

The approach proposed in this thesis aims at respecting the importance of composite operations in model versioning by considering their applications during the merge process. Composite operations are, in more general terms, model transformations. Therefore, we discuss the state of the art of model transformation in this section and present existing model transformation languages. One very promising approach for easing the specification of model transformations is *model transformation by example* (MTBE). In this thesis, we introduce a novel technique called *model transformation by demonstration*, which can be seen as a special kind of MTBE. Therefore, we also review existing work in MTBE in this section.

2.3.1 Basics of Model Transformations

In general, a model transformation takes a model as input and generates a model as output[14]. Mens et al. [MG06] distinguish between two kinds of model transformations. First, there are

[14] Also multiple input models and output models may be possible, but in the scope of this thesis, such settings are not considered.

exogenous transformations, which are also referred to as model-to-model transformations or out-place transformations. In such transformations, the source and target metamodels are *distinct*, as for instance, a transformation from UML class diagrams to ER models [Che76]. Second, there are *endogenous transformations*, which are also referred to as in-place transformations, deal with scenarios, in which the source and target metamodels are the *same* as, for instance, a refactoring of a UML class diagram. In the following, we elaborate on these two kinds in more detail.

Exogenous transformations. Exogenous transformations are used both to exploit the constructive nature of models in terms of *vertical transformations*, thereby changing the level of abstraction and building the bases for code generation, and for *horizontal transformation* of models that are at the same level of abstraction [MG06]. Horizontal transformations are of specific interest to realize different integration scenarios as, for instance, translating a UML class model into an Entity Relationship (ER) model [Che76]. In vertical and horizontal exogenous transformations, the complete output model has to be built from scratch.

Endogenous transformations. In contrast to exogenous transformations, endogenous transformation only rewrite the input model to produce the output model. The first step in such transformations is the identification of model elements to rewrite, and, in the second step, these elements are updated, added, and deleted. Endogenous transformations are applied for different tasks, such as model refactoring, optimization, evolution, and simulation, to name just a few.

Model transformation languages. Various model transformation approaches have been proposed in the past decade following different paradigms (cf. [CH06] for a survey). However, mostly they are based on either a mixture of declarative and imperative concepts, such as ATL [JABK08], ETL [KPP08], and RubyTL [CMT06], or on graph transformations, such as AGG [Tae03] and Fujaba [NNZ00], or on relations, such as MTF[15] and TGG [AKRS06]. Moreover, the Object Management Group (OMG) has published the model transformation standard QVT [OMG05a]. All approaches describe model transformations by transformation rules using metamodel elements, whereas the rules are executed on the model layer for transforming a source model into a target model. Rules comprise *in-patterns* and *out-patterns*. The in-pattern defines when a rule is actually applicable and retrieves the necessary model elements for computing the result of a rule by querying the input model. The out-pattern describes what the effect of a rule is, such as which elements are created, updated, and deleted. All mentioned approaches are based on the abstract syntax of modeling languages only, and the concrete syntax (i.e., the notation) of the modeling language is completely neglected.

2.3.2 Model Transformation By Example

Specifying model transformations with existing model transformation languages requires users to know the respective transformation language. Moreover, users also have to be familiar with the metamodel (i.e., the abstract syntax) of the involved modeling languages. This is because

[15] `http://www.alphaworks.ibm.com/tech/mtf`

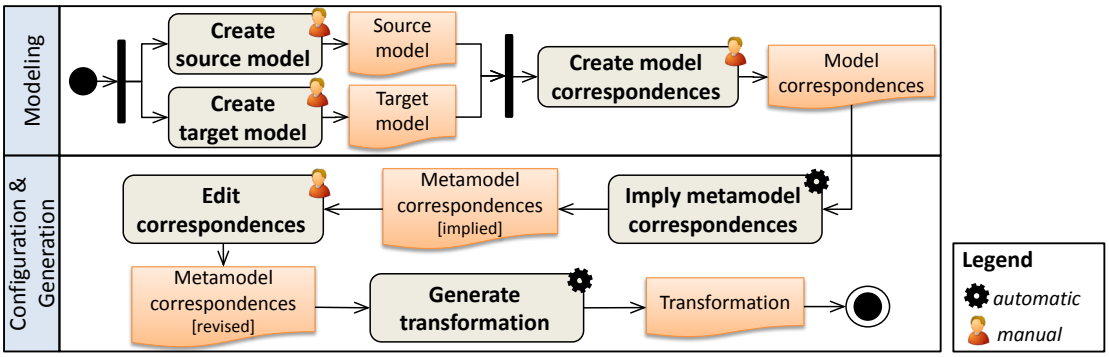


Figure 2.6: Process of Model Transformation by Example

current model transformation languages are specified using the abstract syntax of a model and not using the concrete syntax, which is, however, the only representation users are more familiar with. Thus, creating such transformations based on the abstract syntax is often complicated and hard to accomplish [BW07, dLV02, SW08, Var06, WMA+07].

To address this problem, *model transformation by example* (MTBE) approaches have been proposed, which follow the same fundamental idea as *query by example* developed for querying database systems by giving examples of query results [Zlo75] and *programming by example* for demonstrating actions, which are recorded as replayable macros [Lie01]. Thus, instead of specifying the transformation in terms of rules operating on metamodel concepts, MTBE allows to define transformations using examples represented in the model's concrete syntax. Consequently, the user's knowledge on the concrete syntax (i.e., the notation) of a modeling language is sufficient for developing model transformations.

During the last five years, various MTBE approaches [BV09, DHN09, GMnGSFF09, KSB08, Var06, WSKK07], have been proposed. In the following, we discuss the general process of specifying model transformations by example, and subsequently, we present an instantiation of this process for transforming UML Class Diagrams to ER Diagrams [Che76]. Finally, we conclude this section by elaborating on the peculiarities of current MTBE approaches.

MTBE Process

The main idea of MTBE is the semi-automatic generation of transformations from so-called correspondences between source and target model pairs. The underlying process for deriving model transformations from model pairs is depicted in Figure 2.6. This process, which is largely the same for all existing approaches, consists of five steps grouped in two phases.

Phase 1: Modeling. In the first step, the user specifies semantically equivalent model pairs. Each pair consists of a source model and a corresponding target model. The user may decide whether to specify a single model pair covering all important concepts of the modeling languages, or several model pairs whereby each pair focuses on one particular aspect. The requirement on the model pairs are twofold. First, certainly they must conform to their metamodels,

and second, all available modeling concepts of the source modeling language should be covered by the examples—at least for the intersection of both modeling languages. In the second step, the user has to align the source model and the target model by defining correspondences between source model elements and corresponding target model elements. For defining these correspondences, a correspondence language has to be available. One important requirement is that the correspondences may be established using the concrete syntax of the modeling languages. Hence, the modeling environment must be capable of visualizing the source and target models and the correspondences in one diagram or at least in one dedicated view.

Phase 2: Configuration & Generation. After finishing the mapping task, a dedicated reasoning algorithm is applied to derive automatically *metamodel correspondences* from the model correspondences. How the reasoning is actually performed is explained in more detail by the example discussed below. The automatically derived metamodel correspondences might not always reflect the intended mappings. Thus, the user may revise some metamodel correspondences or add further constraints and value computations. Note that this step is not foreseen in all MTBE approaches, because it may be argued that this is contradicting with the general by-example idea of abstracting from the metamodels. Nevertheless, it seems to be more user-friendly to allow the modification of the metamodel correspondences in contrast to modifying the generated model transformation code at the end of the generation process. Finally, a code generator takes the metamodel correspondences as input and generates executable model transformation code.

MTBE Example

For exemplifying the presented MTBE process, we apply it to specify the transformation of the core concepts of UML class diagrams into ER diagrams. As modeling domain, a simple university information system is used. The user starts with creating the source model comprising the UML classes Professor, and Student, as well as a one-to-many association between them as depicted in the upper left area of Figure 2.7. Subsequently, the corresponding ER diagram, depicted in the upper right area of Figure 2.7, is created. In this figure, both models are represented in the concrete syntax, as well as in the abstract syntax in terms of UML object diagrams. After both models are established, the correspondence model is created, which consists of simple one-to-one mappings. These mappings are depicted as dashed lines in Figure 2.7a and Figure 2.7b between the source and target model elements.

In the next step, a reasoning algorithm analyzes the model elements and its properties (i.e., attribute and reference values) in the source and target models, as well as the correspondences between them in order to derive metamodel correspondences. In the following, we discuss inferring metamodel correspondences between classes, attributes, and references.

Class correspondences. For detecting class correspondences, the reasoning algorithm first checks whether a certain object type in the source model is always mapped to the same object type in the target model. If this is the case, a *full equivalence mapping* between the respective classes in the source and target metamodel is generated. In our example, a full equivalence

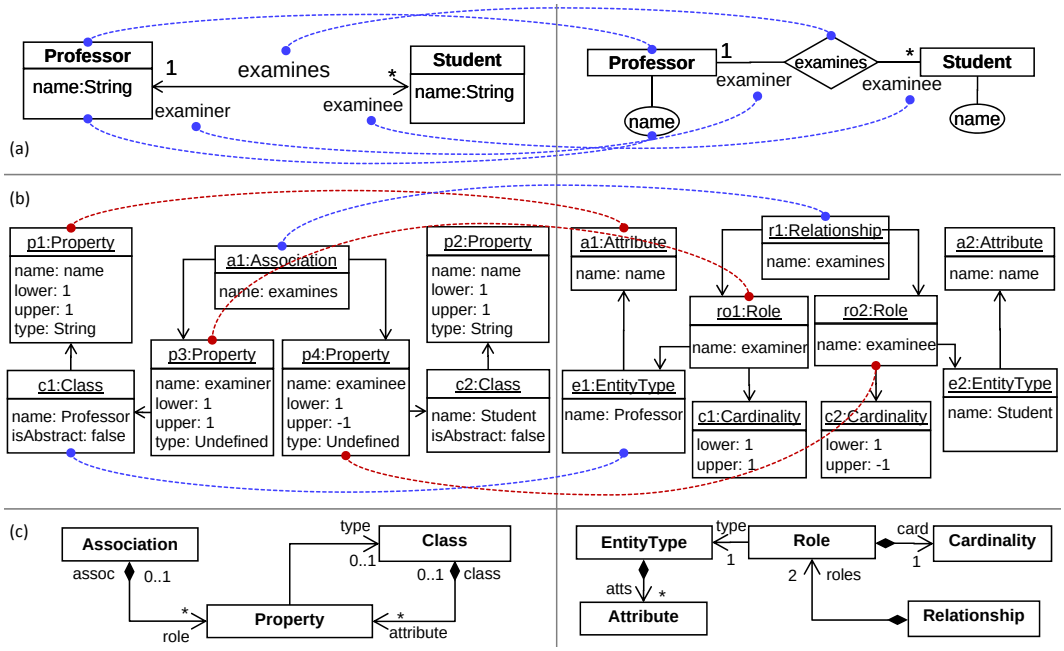


Figure 2.7: Example for Exogenous Transformations: (a) Correspondences in Concrete Syntax, (b) Correspondences in Abstract Syntax, and (c) Metamodels

mapping between objects of type Class and objects of type EntityType is inferred. However, Properties in the source model are mapped to different object types, namely Attributes and Roles, depending on their attribute values and links. For such cases, an additional mapping kind is used, namely *conditional equivalence mapping*. The conditions of such a mapping are derived by analyzing the links and values of the involved objects to find a discriminator for splitting the source objects into distinct sets having an unambiguous mapping to target objects. One appropriate heuristic for finding such a discriminator is to examine the container links of these objects. Therefrom, the algorithm may deduce the constraints `property.class != null` to find an unambiguous mapping to Attributes and the condition `property.assoc != null` for Roles. Finally, also unmapped objects, such as the Cardinality objects have to be considered. In our example, these objects have to be generated along with their container objects of type Role. Thus, the mapping for Roles has to be extended to a *one-to-two conditional equivalence mapping*. By this, a Role object and a properly linked Cardinality object is created for each Property in the source model.

Attribute correspondences. We may distinguish between *ontological attributes* and *linguistic attributes* [KKK+06] in metamodels. Ontological attributes represent semantics of the real-world domain. Values have to be explicitly given by the user. Examples for ontological attributes are Class.name or Attribute.name. In order to find correspondences between ontological attributes, heuristics have to be used which compare the attribute values, for instance, based on edit distance metrics. In our example, we may conclude that Class.name should be

mapped to EntityType.name because the values of the name attributes are equivalent for each Class/EntityType object pair. In contrast, linguistic attributes are used for the reification of modeling concepts, such as Class.isAbstract. Usually these attributes have predefined, restricted value ranges in the language definition. When dealing with linguistic attributes in the context of MTBE, similar heuristics based on string matching as for ontological attributes may be used. However, the probability for accidentally matching wrong pairs and for ambiguities is much higher. Consider for instance the mapping between the property p3 and the role ro1 without taking into account other mappings. Then, we cannot decide if the attribute Property.lower is mapped to Role.cardinality.lower or to Role.cardinality.upper by solely looking at the example. Here, the problem is that we do not have unique values which help us finding the metamodel correspondences. This may be improved by using matching techniques on the metamodel level for finding similarities between attribute names. An alternative solution used in this example is to define an additional mapping between the property p4 and the role ro2 where we have unique values for the lower and upper attributes.

Reference correspondences. For deriving reference correspondences, the previously calculated class correspondences are of paramount importance since they serve as anchors for reasoning about corresponding links. For example, consider the reference atts in the ER metamodel between EntityType and Attribute. For finding the corresponding reference in the UML metamodel, we have to reason about the previously derived class correspondences. First, the Attribute class in the ER metamodel is mapped to the Property class of the UML metamodel. Furthermore, when looking at the example models, each Attribute is contained by an EntityType and each Property is contained by a Class. Luckily, the EntityType class is accordingly mapped to the Class class on the metamodel level, so that we can conclude that whenever transforming a Property into an ER Attribute, a link between the created Attribute and the EntityType previously generated for the Class containing the aforementioned Property is generated. Thus, there should be a correspondence between the reference atts in the ER metamodel and the reference attribute in the UML metamodel.

After the metamodel correspondences have been derived automatically, MTBE approaches usually allow the user to verify and adapt the generated correspondences. For our running example, however, this is not required. The next task is to translate automatically the correspondences into executable transformation code. Listing 2.1 depicts the transformation required for our running example in imperative OCL [Cab07]. For each metamodel correspondence, a transformation rule is generated which queries the source model and generates the corresponding target model elements. Inside each rule, the attribute and reference correspondences are translated to assignments. Please note that current transformation engines are able to schedule rules automatically and to build an implicit trace model between the source and target model. Based on this trace model, assignments, such as `e.atts = c.attribute` (cf. line 2 in Listing 2.1) are automatically resolved. In particular, not the UML attributes (`c.attribute`) are assigned to the EntityType, but the ER attributes generated *from* these UML attributes are resolved by applying the trace model. These features of transformation languages and their encompassing engines drastically ease the transformation code generation from correspondences.

Listing 2.1: Generated Transformation Code

```
1  rule 1: Class.allInstances() -> foreach (c |
2      create Entity e (e.name = c.name, e.atts = c.attribute);
3  rule 2: Property.allInstances()
4      -> select(p | p.class <> OclUndefined) -> foreach( p |
5      create Attribute a (a.name = p.name));
6  rule 3: Property.allInstances() -> select(p |
7      p.assoc <> OclUndefined) -> foreach( p |
8      create Role r (r.name = p.name, r.cardinality = c),
9      create Cardinality c (c.upper = p.upper, c.lower = p.lower,
10     r.type = p.type));
11 rule 4: Association.allInstances() -> foreach(a |
12     create Relationship r (r.name = a.name, r.roles = a.role));
```

Existing Approaches

We compare existing approaches by highlighting their commonalities and differences. Mostly all approaches define the input for deriving exogenous transformations as a triple comprising an input model, a semantically equivalent output model, as well as correspondences between these two models. These models have to be built by the user, preferably using the concrete syntax as is, for instance, supported by [WSKK07], but most approaches do not provide dedicated support for defining the correspondences in graphical modeling editors.

Subsequently, reasoning techniques, such as specific rules again implemented as model transformations [GMnGSFF09, Var06, WSKK07], inductive logic [BV09], and relational concept analysis [DHN09] are used to derive model transformation code. Current approaches support the generation of graph transformation rules [BV09, Var06] or ATL code [GMnGSFF09, WSKK07].

All approaches aim for *semi-automated* transformation generation meaning that the generated transformations are intended to be further refined by the user. This is especially required for transformations involving global model queries and attribute calculations, such as aggregation functions, which have to be manually added. Furthermore, it is recommended to develop iteratively the transformations, i.e., after generating the transformations from initial examples, the examples must be adjusted or the transformation rules must be adapted in case the actual generated output model is not fully equivalent to the expected output model. However, in many cases it is not obvious whether to adapt *the aligned examples* or the *generated transformations*. Furthermore, adjusting the examples might be a tedious process requiring a large number of transformation examples to assure the quality of the inferred rules. In this context, self-tuning transformations have been introduced [KWSK09, KSB08]. Self-tuning transformations exploit the examples as training instances in an iterative process for further improving the quality of the transformation. The goal is to minimize the differences between the actual output model produced by the transformation and the expected output model given by the user by using the differences to adapt the transformation over several iterations. Of course, adapting the transformation is a computation intensive problem leading to very large search spaces. Whereas in [KWSK09] domain-specific search space pruning tailored to EMF-based models is used, a generic meta-heuristic–based approach is used in [KSB08] to avoid an exhaustive search.

2.3.3 Summary

Model transformations gained enormous attention from the MDE community in research and practice. As a result, several matured dedicated model transformation languages emerged. In this domain, MTBE—as an approach to ease the challenging task of manually specifying model transformations in terms of metamodel-based transformation rules—seems to be a very promising research direction. A variety of research papers on MTBE have been published within the last years. However, all of these papers mentioned above focus on deriving *exogenous* model transformations from user-specified correspondences between a source and target model. Interestingly, MTBE dedicated for endogenous model transformations has not gained much attention yet.

At the time when we started to work in this field, we were the first to propose an MTBE approach dedicated to endogenous model transformations in [Lan09, BLSW09, BLS+09], which takes advantage of the *demonstration* of an endogenous model transformation performed by a user instead of exploiting user-specified model correspondences as is the case for existing MTBE approaches. Nevertheless, at the same time a very similar approach by Sun et al. [SWG09] emerged. Sun et al. introduced the notably suitable term *model transformation by demonstration* for this approach—in the remainder of this thesis, we will adopt this term for our approach. As their approach has been published at the same time as we published ours, we refrain from discussing their approach as *state of the art*; we rather present a detailed comparison between their approach and the one presented in this thesis in Chapter 4.

CHAPTER 3

Adaptable Model Versioning

In this chapter, we present the big picture of the proposed adaptable model versioning system AMOR [BKS+10]. This system is the result of the equally named research project[1], which has been carried out from 2009 to 2011. Please note that the basic idea behind AMOR (i.e., building an adaptable model versioning system) and the AMOR merge process have been elaborated conjointly by all project participants[2].

In the following, we present some motivating examples posing the challenges that are solved in this thesis. Next, we introduce a categorization of conflicts that might occur when merging the parallel work of two developers on the same model. The goal of this categorization is to set up the terminology of conflicts used in the remainder of this thesis. Subsequently, we discuss the basic design principles of AMOR and disclose our design rationale. We also provide a brief introduction of AMOR's technical infrastructure in this chapter. Finally, we present an overview of the generic merge process first and subsequently, we show how this generic process is extended in order to be adaptable with respect to language-specific knowledge.

3.1 Motivating Examples

In this section, we introduce small model versioning scenarios in which two developers, referred to as developer 1 and developer 2 in the following, concurrently modify a common original model denoted with V_o. The issues occurring in these scenarios go beyond simple spatial overlapping and, hence, conflicting operations. Instead, these scenarios illustrate merge issues for which language-specific knowledge is necessary to handle them correctly. Hence, current generic approaches would largely fail to either report the correct conflict or to produce an optimally merged version.

[1] AMOR (`http://www.modelversioning.org`), a research project funded by the Austrian Federal Ministry of Transport, Innovation, and Technology and the Austrian Research Promotion Agency under grant FIT-IT-819584.

[2] In alphabetical order: Petra Brosch, Gerti Kappel, Philip Langer, Werner Retschitzegger, Wieland Schwinger, Martina Seidl, Konrad Wieland, and Manuel Wimmer.

3.1.1 Additions of Equal Model Elements

Although entirely equal operations might be treated correctly to a certain extent by generic model versioning systems, in several scenarios additional language-specific knowledge is necessary to enable the correct identification of operations having an equal effect—especially if additions of model elements are involved. Otherwise, the *equality of model elements* may be hard to determine.

An example of such a scenario is depicted in Figure 3.1a. In this scenario, we illustrate the need for language-specific knowledge by means of UML class diagrams [OMG03]. The original model V_o contains two classes, which are shown in the concrete syntax on the left side, as well as in the abstract syntax in terms of an object diagram on the right side. Please note that we omitted some details in the abstract syntax for the sake of readability.

The original model is now concurrently modified by two developers leading to the revised versions V_{r1} and V_{r2}. Both developers concurrently add a generalization relationship between the classes Employer and Person specifying Employer to be a subclass of Person. Although not directly visible in the concrete syntax, generalizations are realized in UML using a dedicated object of type Generalization. Thus, corresponding Generalization objects, g1 and g2, exist in the object diagrams of V_{r1} and V_{r2}, respectively.

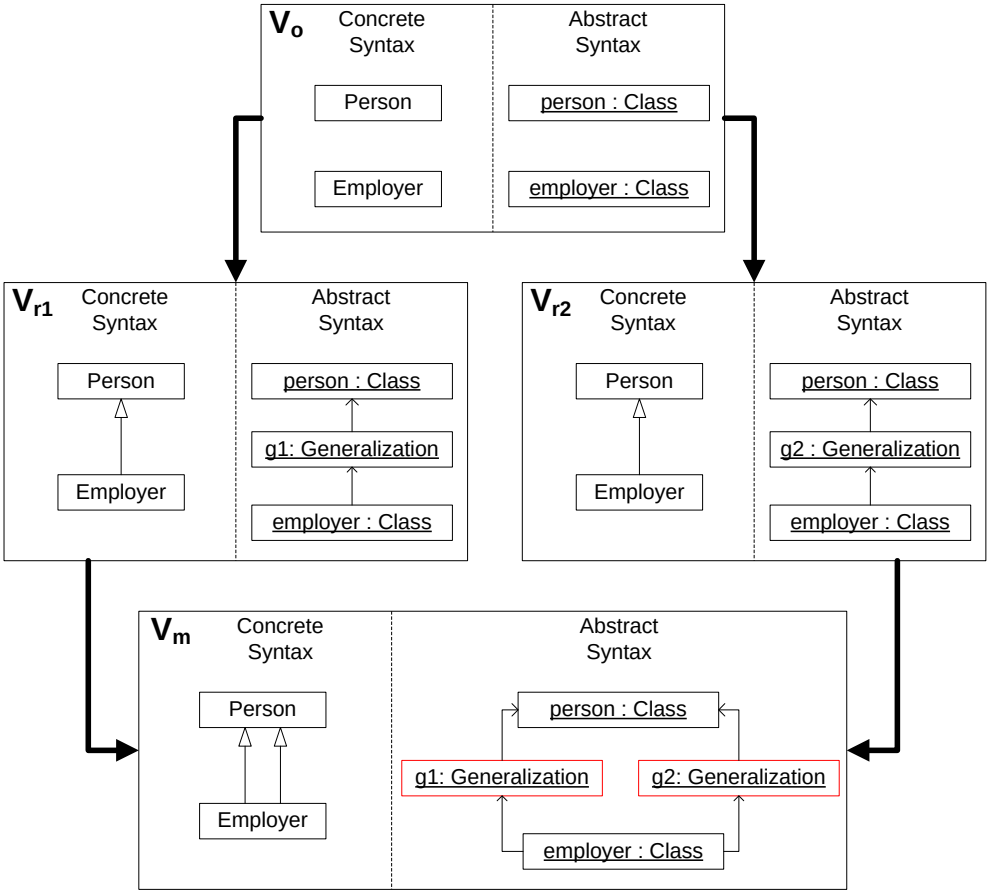
Using a generic merge approach, all modifications applied by both developers may be merged without any conflicts because no spatially overlapping concurrent operations have been performed. However, when naively merging the modifications of both developers, we end up having two Generalization objects expressing exactly the equivalent semantics in the merged model V_m, because two distinct objects g1 and g2 have been added; thus, they are included in the merged version. This redundancy in the merged model is obviously unfavourable and might even, in the worst case, cause the editor to fail when trying to open the merged model. A completely generic merge approach is not aware of the fact that these two objects, g1 and g2, are entirely redundant and as a consequence, is not able to detect and report such a scenario.

An ideal model versioning system would recognize that g1 and g2 are indeed distinct objects that, however, express the equivalent semantics. Being aware of this information, the ideal model versioning system would be able to omit either the addition by developer 1 or the one by developer 2 in order to obtain a finally merged model as depicted in Figure 3.1b.

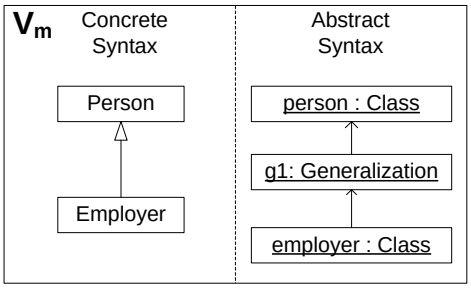
3.1.2 Additions of Similar Model Elements

A comparable yet different scenario is presented using Ecore models [SBPM08] in Figure 3.2a. The common original model V_o contains two Ecore classes, Shop and Product. This model is now concurrently modified. In particular, developer 1 adds the reference sells to Shop, which refers to Product. This reference's cardinality has a lower bound of 1 and an unbounded upper bound (cf. V_{r1} in Figure 3.2a). Concurrently, developer 2 adds a reference also named sell to Shop, which refers to Product. However, the lower bound is specified to be 0 (cf. V_{r2} in Figure 3.2a). Thus, the added model elements are largely similar, but not completely equivalent because of the different lower bounds 0 and 1.

Applying a generic merge to this scenario, we obtain the model V_m depicted in Figure 3.3b. As expected, this merged model contains both the reference sells added by developer 1 as well

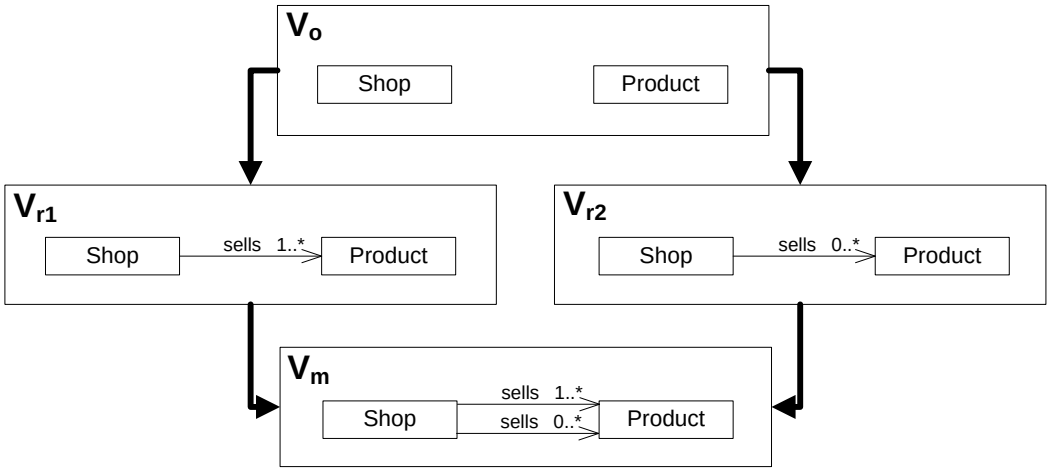


(a) Scenario with Generic Merge

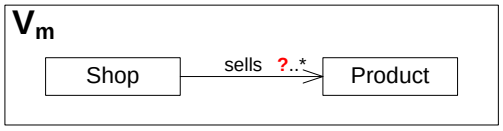


(b) Optimal Merge

Figure 3.1: Addition of an Equal Model Element



(a) Scenario with Generic Merge



(b) Optimal Merge

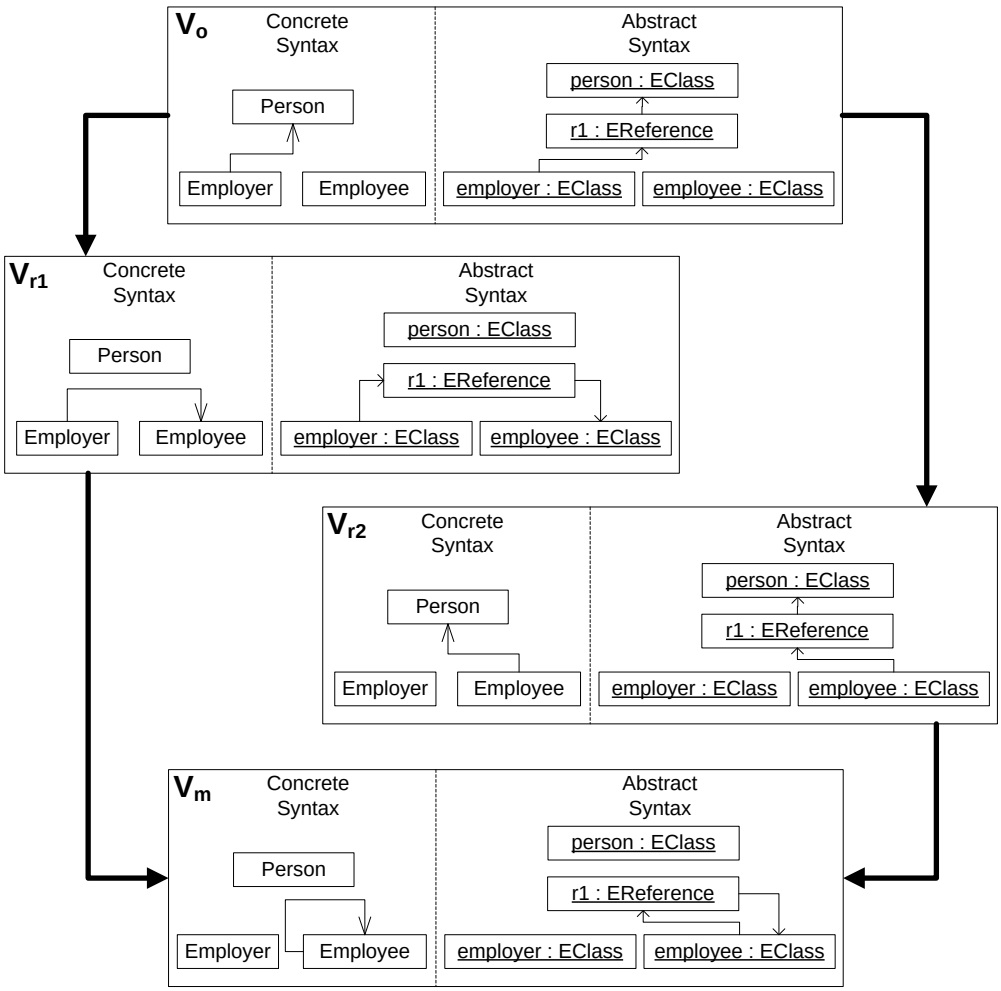
Figure 3.2: Addition of a Similar Model Element

as the reference sells added by developer 2. Ultimately, we end up having two redundant equally named references in the merged model. Moreover, the two redundant associations are not completely equal. Hence, a decision has to be made among the developers to specify which lower bound finally should be applied in the merged model. However, a generic merge approach, which is unaware of the fact that an reference's name as well as its target class are the meaning-carrying properties or the *signifier* of a reference, would neither detect the redundancy nor indicate the need for such a decision. The term *signifier* is discussed in Section 3.2 in more detail.

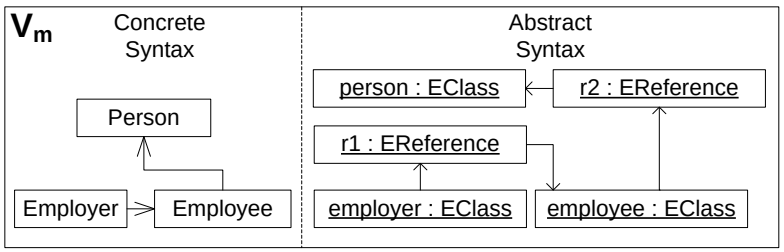
Ideally, a model versioning system that involves language-specific knowledge would be able to detect the correspondence between both added references, because it would compare the references' names as well as their target class. Being aware of the correspondence, an ideal system would only incorporate one of both additions performed by the developers so that only one reference is included in the merged model. Additionally, the system should detect that the added objects, however, are not entirely equal and, therefore, indicate the need for a decision on how to resolve this contradiction.

3.1.3 Concurrent Change of a Model Element's Signifier

Another type of conflict, which may be hard to detect by solely generic approaches, occurs if two concurrent, yet not spatially overlapping operations, both modify the same model element's properties. Ultimately, the resulting model may obfuscate the intentions of both developers.



(a) Scenario with Generic Merge



(b) Optimal Merge

Figure 3.3: Concurrent Change of a Model Element's Signifier

Consider the scenario for Ecore models depicted in Figure 3.3a. The model versions are shown in the concrete syntax on the left side and in the abstract syntax in terms of object diagrams on the right side. The original model V_o contains three classes; namely, Person, Employer, and Employee. Additionally, there is a reference between Employer and Person.

Now, developer 1 modifies the target of the reference from Person to Employee (cf. V_{r1} in Figure 3.3a). As a result, the object r1 representing the reference is retained, but its link is changed from Person to Employee. Concurrently, developer 2 modifies the source of the reference from Employer to Employee (cf. V_{r2} in Figure 3.3a). In the concrete syntax, this change is realized by moving the reference r1 from its original container Employer to Employee. Developer 1 intended the reference to go from Employer to Employee and developer 2 wanted the reference to go from Employee to Person.

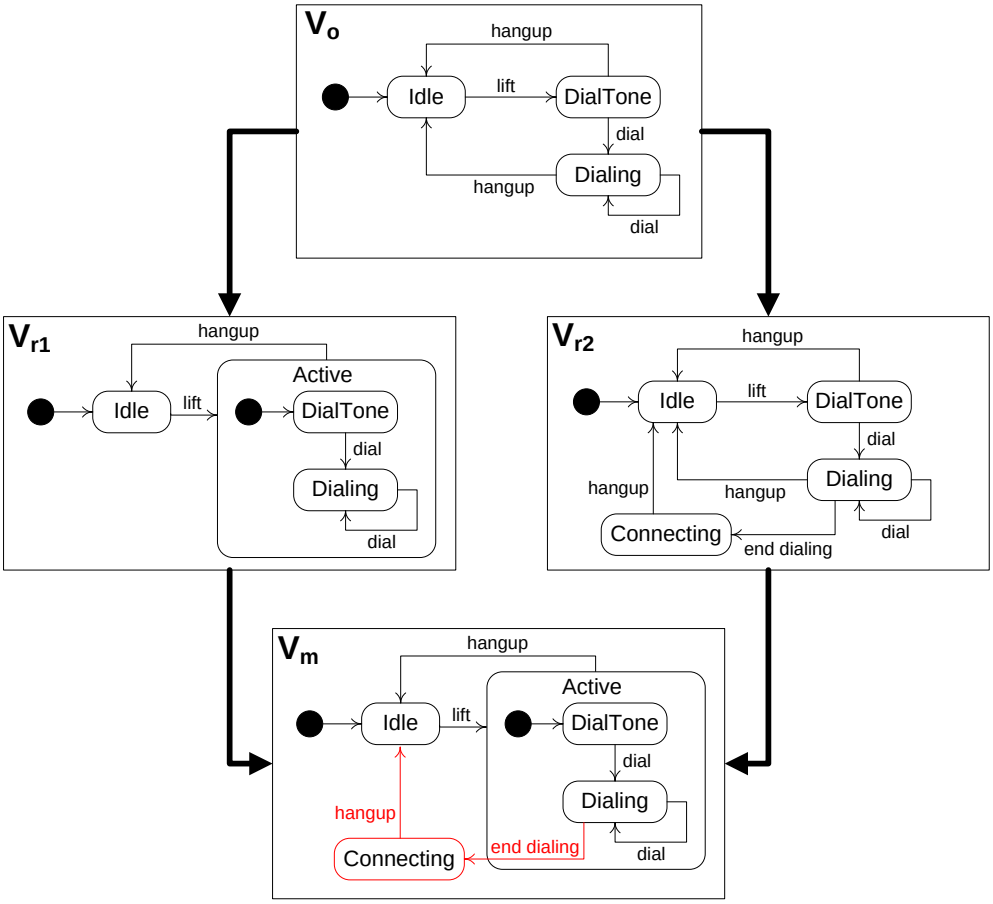
As in the previous scenario, a generic model versioning system would not report a conflict because no spatially overlapping operations have been applied to the original model. Therefore, all modifications are merged to obtain an integrated model: the target of the reference r1 is changed from Person to Employee (as performed by developer 1) and the reference is moved from the original container Employer to Employee. However, merging both operations leads to a model that, in the end, contradicts the intention of *both* developers: in the merged model V_m (cf. Figure 3.3a), the reference is contained by Employee and also refers to Employee. In other words, the reference has been changed accidentally into a reflexive reference, although none of the developers intended it to be that way.

One potential merged model that better reflects the intentions of both developers is depicted in Figure 3.3b. In this merged model V_m, the original reference r1 has been duplicated: one reference reflects the operations of developer 1 and the other one reflects the operations of developer 2. Admittedly, this is only *one possible* way of resolving this issue. The developers should be confronted with a warning so that they are aware of their indirectly contradicting operations regarding the meaningful properties or *signifier* of a model element. Unfortunately, generic approaches will not be able to identify the concurrent change of a model element's signifier (cf. Section 3.2 for a more detailed discussion of the term *signifier*).

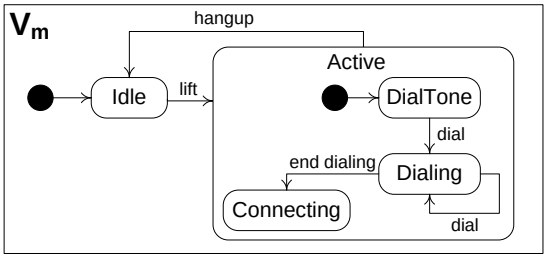
3.1.4 Intentions Behind Composite Operations

As already stressed in Section 2.1.2, the importance of considering composite operations in the conflict detection and during the merge is neglected by current model versioning systems. Of course, the intention behind composite operations can not be regarded by solely generic approaches. In the following, we present a scenario that illustrates the drawbacks of neglecting composite operations in detail by discussing the model versioning scenario depicted in Figure 3.4a.

Consider the common original UML state machine [Har87] V_o in Figure 3.4a representing the states of a phone: starting in the state Idle, the phone changes its state to DialTone when the event lift (the handset) is issued. Being in this state, users may hangup or dial, which causes the state of the phone to change to the state Dialing. In this state, a user may keep on dialing until she hangs up the handset again, which causes the phone to switch to the state Idle. Please note that for the sake of readability this state machine does not cover all possible states of a real-world phone.



(a) Scenario with Generic Merge



(b) Optimal Merge

Figure 3.4: Intention Behind a Composite Operation

Again, this original model is changed by two developers in parallel. Developer 1 identifies the need for applying the refactoring *Introduce Composite State* [SPLTJ01] to this state machine. Therefore, a new composite state called Active is introduced. Next, the states DialTone and Dialing are moved to the newly created composite state. Then, the target of the transition called lift, which was originally DialTone, is changed to the new composite state. To preserve the semantics of the state machine, a new initial state with a transition to DialTone has to be created in the composite state. Finally, both transitions named hangup, which are outgoing from state DialTone and Dialing back to Idle can be folded: one of these transitions is deleted and the other one is moved to the composite state. The refactored state machine is depicted in V_{r1} in Figure 3.4a.

In parallel, developer 2 works towards completing this state machine and adds a new state named Connecting. This state has one incoming transition, namely end dialing and one outgoing transition named hangup referring back to the state Idle (cf. V_{r2} in Figure 3.4a).

When merging these two revised state machines using a generic merge algorithm, all atomic operations that have been performed by both developers can be merged without any issues. The resulting merged state machine V_m is depicted in Figure 3.4a. This state machine contains the composite state as well as the state Connecting, which resides outside of the composite state having the outgoing transition hangup. However, recall that the original intention behind the refactoring applied by developer 1 is to collect all states sharing the common transition hangup and put them together into the composite state Active. This is obviously not the case in the naively merged state machine V_m in Figure 3.4a.

A merged state machine, which better reflects the intentions of both developers, is illustrated in Figure 3.4b. In this state machine, the new state Connecting resides *within* the composite state Active just as developer 1 intended it to be. Of course, the transition hangup, which originally was outgoing from Connecting, is removed because this transition is already present in the containing composite state Active.

3.1.5 Violated Preconditions of Composite Operations

Another scenario for merging state machines is depicted in Figure 3.5. The original model V_o is equal to the original model in Section 3.1.4. Again, developer 1 performs the refactoring *Introduce Composite State* in order to collect all states having an outgoing state named hangup. However, in contrast to the previous scenario in Section 3.1.4, developer 2 now does not introduce an additional state but renames the transition connecting DialTone and Idle from hangup to abort (cf. V_{r2} in Figure 3.5).

Again, the modifications of both developers can be merged by fine-grained generic model versioning systems without raising any conflicts. In the resulting model V_m in Figure 3.5, the transition going from the composite state Active to Idle is, according to the change of developer 2, now named abort. As a result, the merge inadvertently changed the semantics of the model because the state Dialing, which originally had an outgoing transition named hangup, now implicitly has an outgoing transition abort from its containing composite state Active.

The reason for this unintended change of the semantics is that the preconditions responsible for ensuring the semantic preservation of the state machine refactoring have not been considered during the merge. Ideally, a model versioning system would check whether the preconditions

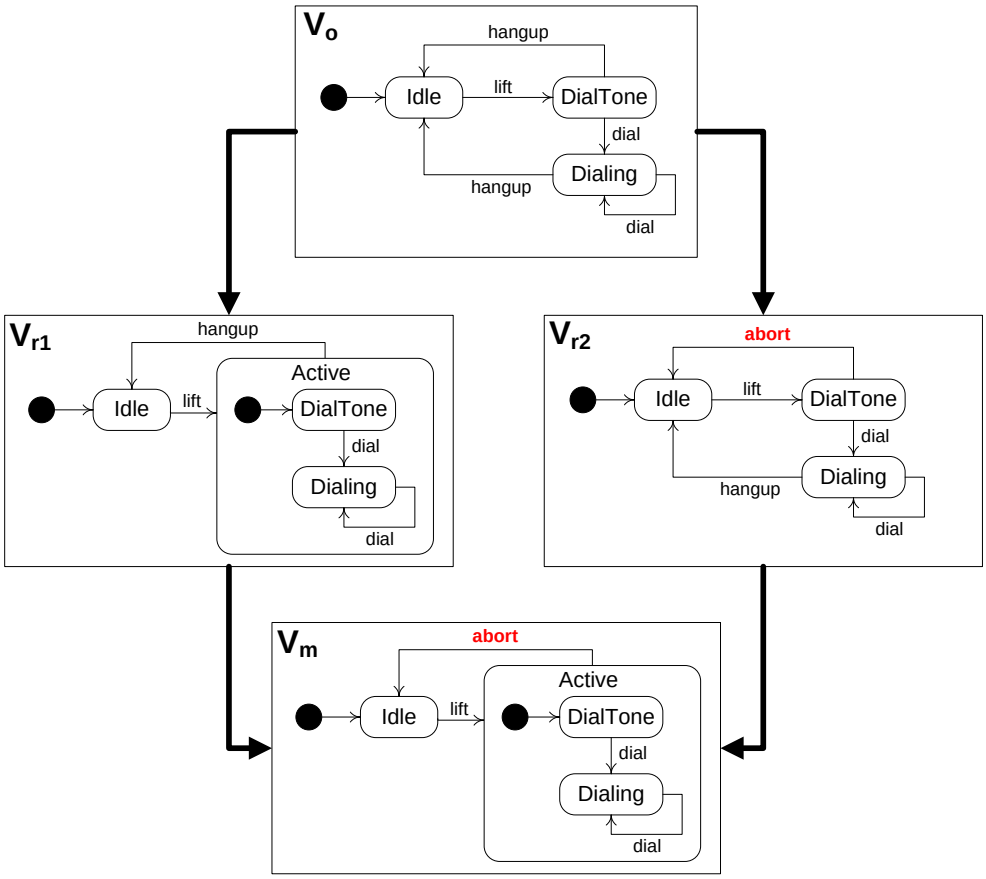


Figure 3.5: Violated Precondition of a Composite Operation

of the applied composite operations still hold after the modifications of the opposite developer have been performed. In the scenario at hand, the condition of the *Introduce Composite State* refactoring restricting all folded transitions to be named equally is violated in V_{r2} depicted in Figure 3.5 (DialTone.abort $\neq$ Dialing.hangup). Consequently, a corresponding conflict indicating this violation of the composite operation's precondition should be raised prior to constructing the merged model.

3.1.6 Inconsistent Merge Results

The model versioning scenario depicted in Figure 3.6 deals with operations that lead to an inconsistent merge result in terms of language-specific validation rules. In the original model V_o in Figure 3.6, a UML model comprising a class diagram and a dependent sequence diagram is illustrated. More specifically, the class diagram contains two classes, namely Client and Logger. The class Logger can be instantiated using the public constructor Logger() and may receive messages to the operation print(String). The interaction between these two classes is specified in

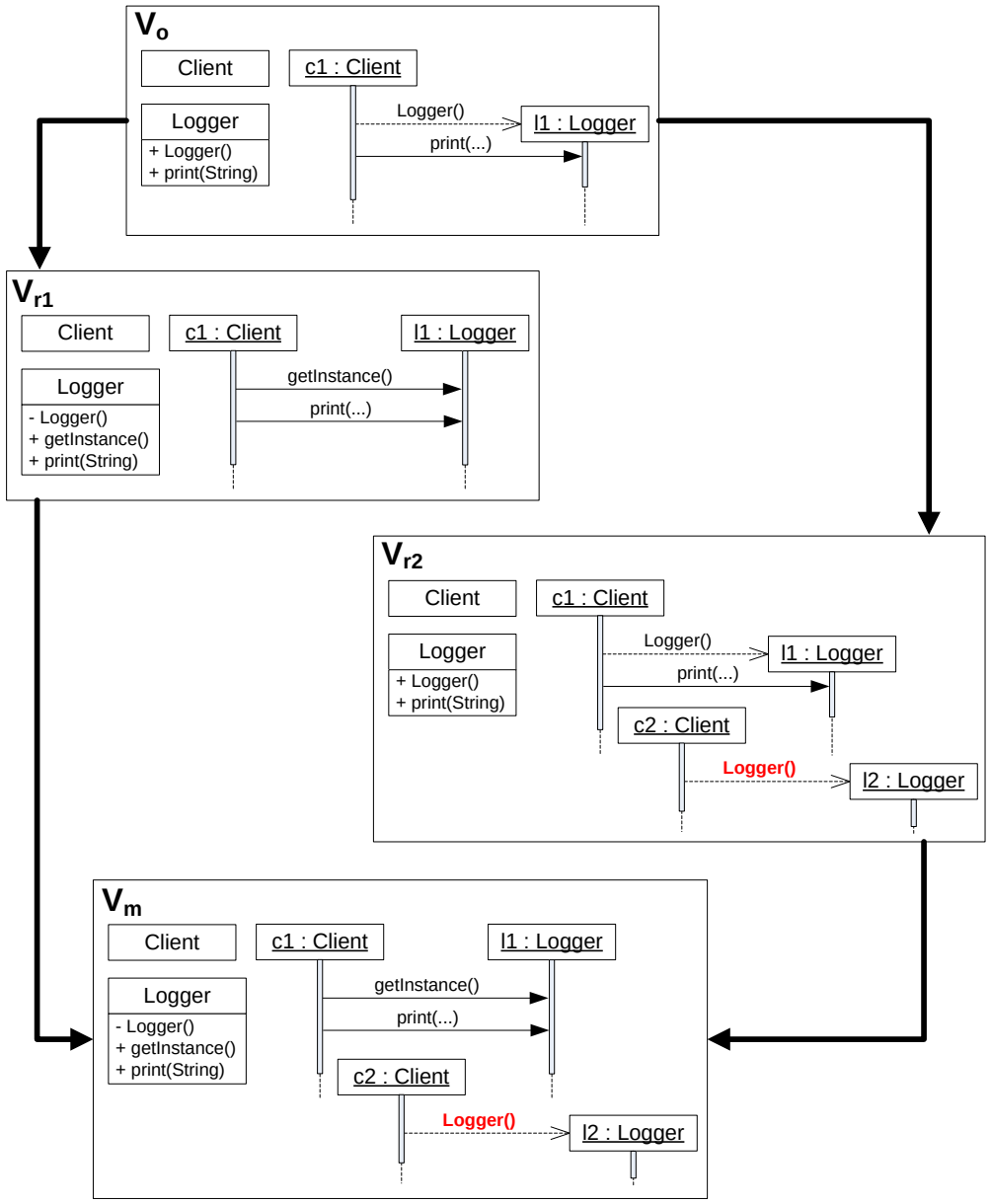


Figure 3.6: Inconsistent Merge Results

the sequence diagram next to the class diagram in Figure 3.6. In particular, the client instantiates the logger using the public constructor in order to be able to call the operation print().

This UML model is now concurrently modified by two developers. Developer 1 decides to turn the class Logger into a *Singleton* [GHJV95]. More precisely, developer 1 modifies the visibility of the constructor to *private* and introduces a new operation named getInstance for obtaining the single instance of the class. Accordingly, developer 1 also adapts the sequence

diagram: instead of creating the instance of Logger by calling its constructor, the new operation getInstance is used (cf. V_{r1} in Figure 3.6).

In parallel, developer 2 introduces new instances of the classes Client and Logger in the sequence diagram. Unaware of the operations performed by developer 1, developer 2 adds a call to the constructor for instantiating the class Logger (cf. V_{r2} in Figure 3.6).

When the modifications of both developers are merged generically, no conflict is raised. Instead, we obtain the merged model V_m depicted in Figure 3.6. In this merged model, the class Logger is, according to the operations by developer 1, a singleton containing the constructor, which is now private, as well as the operation getInstance. Also, the part of the sequence diagram that already existed in the original model V_o has been adapted accordingly because the operations of developer 1 are incorporated into the merged model. However, this model comprises an inconsistent call of the private constructor of the class Logger in the part of the sequence diagram that has been introduced by developer 2 who was not aware of the modifications performed by developer 1.

3.2 Categorization of Conflicts

Having presented some exemplary conflict scenarios, we now present a more systematic view on conflicts by grouping conflict types into categories. For this purpose, we first discuss the meaning of the term *conflict* in related research areas and survey existing categorizations of conflict types. We derive the terminology of conflict types used in the remainder of this thesis.

3.2.1 Existing Conflict Categorizations

The term *conflict* has been used in the area of versioning to refer to interfering operations in the parallel evolution of software artifacts. However, the term conflict is heavily overloaded and differently co-notated. Besides using the term conflict, also the terms *interference* and *inconsistency* have been applied synonymously in the literature as, for instance, in [Fea89, TP05] and [Men02], respectively. The term conflict usually refers to directly contradicting operations; that is, two operations, which do not commute [LvO92]. Nevertheless, there is a multitude of further problems that might occur, especially when taking syntax and semantics of the versioned artifact's language into account. Therefore, in order to better understand the notion of conflict, different categories have been created to group specific merge issues as surveyed in the following.

In the field of software merging, Mens [Men02] introduces *textual*, *syntactic*, *semantic*, and *structural* conflicts. Whereas *textual* conflicts concern contradicting operations applied to text lines as detected by a line-based comparison of a program's source code, *syntactic* conflicts denote issues concerning the contradicting modification of the parse tree or the abstract syntax graph; thus, syntactic merging takes the programming language's syntax into account and may also report operations that cause parse errors when merged (cf. line-based versus graph-based versioning in Section 2.1). *Semantic* merging goes one step further and also considers the semantic annotation of the parse tree, as done in the semantic analysis phase of a compiler. In this context, static semantic conflicts denote issues in the merged artifact such as undeclared vari-

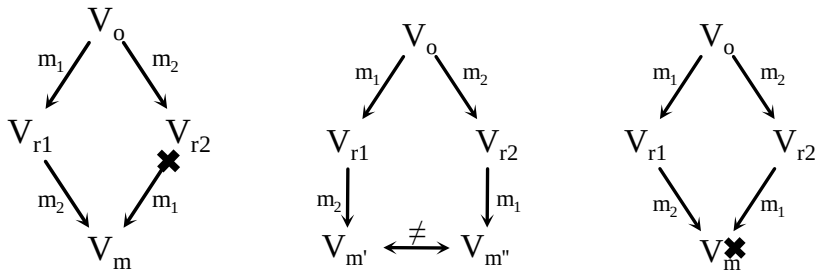
ables or incompatible types. Besides static semantic conflicts, Mens also introduced the notion of *behavioural* conflicts, which denote unexpected behavior in the merged result. Such conflicts can only be detected by applying even more sophisticated semantic merge techniques that rely on the runtime semantics [Men02]. Finally, Mens also introduces the notion of *structural* conflicts, which arise when one of the applied operations to be merged is a "restructuring" (i.e., a refactoring) and the merge algorithm cannot uniqually decide in which way the merged result should be restructured [Men02]. Mens stresses that detecting structural conflicts is a challenging future research topic [Men02]; thus, it is worth noting that detecting structural conflicts among composite modeling operations is a key topic of this thesis (cf. Chapter 6).

Also the notion of conflict in the domain of graph transformation theory serves as a valuable source of knowledge in this matter. As defined by Heckel et al. [HKT02], two direct graph transformations are in conflict if they are *not parallel independent*. Two direct graph transformations are parallel independent if they preserve all elements that are in the match of the other transformation; otherwise we encounter a *delete-use* conflict. Another manifestation of such a case is a *delete-delete* conflict. Although both transformations delete the same element anyway, this is still considered a conflict because one transformation deletes an element that is indeed in the match of the other transformation. If the graph transformations additionally comprise negative application conditions, they also must not create elements that are prohibited by negative application conditions of the other transformation; otherwise an *add-forbid* conflict occurs. To summarize, two direct graph transformations are in conflict, if one of both disables the other. Furthermore, as shown in [Ehr79], based on the local Church-Rosser theorem [CR36], we may further conclude that two parallel independent direct transformations can be executed in any order with the same final result.

In the domain of model versioning, no dedicated, widely accepted categorization of different merge conflict types has been established yet. Nevertheless, Westfechtel establishes a detailed definition of conflicts between two atomic operations in [Wes10]. More precisely, he distinguishes between *context-free conflicts* and *context-sensitive conflicts*. Context-free conflicts denote contradicting changes to the same feature value in the same model element (also known as update-update conflict); thus, the context of the model element is not taken into account. In contrast, context-sensitive conflicts concern also the context of a concurrently modified model element such as the container and referenced model elements. Context-sensitive conflicts are again classified into *(i) containment conflicts*, which occur, for instance, if both developers move the same model element to different containers so that no unique container can be chosen automatically, *(ii) delete conflicts*, which denote delete-update, delete-use, and delete-move conflicts, and finally, *(iii) reference conflicts*, which concern contradicting changes to bi-directional references. This categorization is tailored to EMF models and are defined clearly using set-theoretical rules. However, Westfechtel considers only *generic* conflicts among *atomic* operations.

3.2.2 Conflict Categorization Applied in this Thesis

Having surveyed existing conflict categorizations and terminologies, we now introduce the categories of conflicts and the terminology used in the remainder of this thesis. The following categorization is partly based on the conflict categorization we presented in [BLS+10a, BKL+11a]. Nevertheless, we now adapt and refine certain parts in order to better integrate it into the con-



(a) Parallel Dependence (b) Non-commutativity (c) Inconsistent Result

Figure 3.7: Properties of Two Operations

text of this thesis. Furthermore, we introduce the notion of merge *warnings* representing merge issues, which do not directly interfere with the merge process or destroy the consistency of the model, but which should still be brought to the attention of the involved developers. An overview of the terminology of merge issues is depicted in Figure 3.8.

Two concurrent operations m_1 and m_2 applied to the same version of an artifact V_o may have three different properties that indicate a conflict as depicted in Figure 3.7. First, similar to the concept of parallel independence from graph transformation theory, two operations m_1 and m_2 may be *parallel dependent* (cf. Figure 3.7a). That is, the operation m_2 cannot be applied after the operation m_1 has been applied. In other words, the preconditions of m_1 are not fulfilled anymore after m_2 has been applied. Second, according to Lippe and Oosterom [LvO92], we may encounter the case that the operations m_1 and m_2 do not commute (cf. Figure 3.7b) such that $m_1(m_2(V_o)) \neq m_2(m_1(V_o))$. Thus, no unique merged version can be found. Third, if the operations m_1 and m_2 are parallel independent and commutative, the result V_m may be inconsistent with a specification of the artifact's language, as described by Mens [Men02] (cf. Figure 3.7c).

Overlapping Operations

We use the term *overlapping operations* or *operation-based conflict* to denote two operations that are either parallel dependent or not commutative. Thus, both operations cannot be applied together without nullifying one operation; in other words, overlapping operations interfere with the merge unless at least one of the overlapping operations is omitted. In such a conflict, *atomic* operations as well as *composite* operations may be involved. In the following, we discuss conflicts arising from parallel dependence and non-commutativity in more detail. An overview is depicted in Figure 3.8.

Parallel Dependence. As defined in graph transformation theory, for determining parallel independence the preconditions of operations are crucial. The precondition of atomic operations is that the affected model element still exists. For instance, updating an attribute value of a model element requires that the model element is not deleted by a concurrent operation; otherwise, we

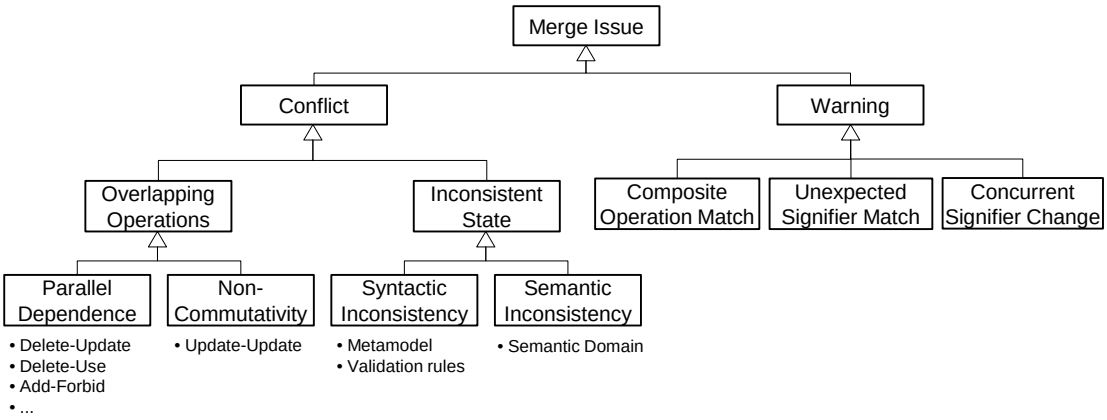


Figure 3.8: Terminology of Merge Issues

encounter parallel-dependent operations, or more precisely, a *delete-update conflict*. Additionally, atomic operations such as adding a link to model elements obviously require both the source model element and the target model elements to exist. In case the target has been deleted concurrently, we use the terminology of graph transformations and denote such a scenario as *delete-use conflict*. For composite operations, the preconditions may be more complicated as they may also check for non-existence in terms of negative application conditions or require certain attribute or reference values in a model. Consequently, with composite operations we additionally may face *add-forbid* or *update-forbid* conflicts. An example of an update-forbid conflict is illustrated in Section 3.1.5. According to graph theory, there are also delete-delete conflicts among direct graph transformations because both transformations require the same element to exist in order to be able to delete the element. However, such a conflict is not important in the context of model versioning because both developers intended to delete the element anyway; hence, we may delete it in the merged model and by this, reflect the intention of both developers.

Non-commutativity. Besides parallel dependence, operations may also overlap if they do not commute. For example, if two operations update the same attribute of a model element to *different* values, the order in which the operations are applied to the common origin model affect the respective attribute value in the merged model; thus, such operations do not commute and are referred to as *update-update* conflicts. Please note that whenever two operations update the same attribute of a model element to the *same value*, the property of commutativity does hold so that no conflict is at hand.

Inconsistent State

Even if concurrent operations are parallel-independent and commutative (i.e., they are not overlapping), they may still cause an inconsistent state if they are both applied to a merged model. This inconsistent state has been caused by operations, but the inconsistency itself concerns the state and may only be detected when analysing the resulting state in contrast to analysing the operations. Hence, we also refer to them as *state-based conflicts*. According to the categorization

of Mens [Men02], we may further distinguish between syntactic inconsistencies and semantic inconsistencies (cf. Figure 3.8). This differentiation is made upon the *specification type* with which the state is inconsistent.

Syntactic Inconsistency. The merged model may be inconsistent with the abstract syntax specification of a modeling language. In our context, the abstract syntax is specified by the metamodel and additional validation rules (e.g., OCL invariants). The metamodel may be seen as the context-free syntax specification and the validation rules as additional context constraints. In the UML specification [OMG03] as well as in other literature, the term *static semantics* is used to refer to such context conditions. However, as stressed by Harel and Rumpe in [HR04], context conditions (also if they are sometimes called static semantics) are not the specification of a language's semantics; context constraints simply further restrict the abstract syntax. Thus, violations of the static semantics are still syntactic conflicts. An example for a syntactic inconsistency is illustrated in Section 3.1.6.

Semantic Inconsistency. The merged model may also be inconsistent with a specification of the semantics of a modeling language. A language's semantics must specify the *meaning* of all concepts using a well-defined and well-understood semantic domain (e.g., denotational semantics [Win93]). At the moment, however, there is "no simple and obvious way to define this complex semantic domain precisely, clearly, and readably." [HR04]. Consequently, the semantics of modeling languages is often specified only in an informal way. Nevertheless, in this thesis, we do not consider semantic inconsistencies, but list them here for the sake of completeness.

Merge Warnings

Conflicts have to be eventually resolved in order to obtain a consolidated and consistent model. However, in many merge scenarios, the involved developers should for now be only informed that there are merge issues, which indeed do not directly interfere the merge process or destroy the consistency of the model, but which should be still carefully reviewed by the developers. Therefore, picking up on the idea of Koegel et al. [KHWH10], we introduce merge *warnings* and discuss the specific types of warnings, which we considered in this thesis, in the following (cf. Figure 3.8).

Composite Operation Match. A composite operation is more than its set of contained atomic operations. The atomic operations are applied to fulfill a common goal reflecting the intention of the developer who applied it. The intention of the developer is fulfilled when the composite operation has been applied successfully to all selected and matching model elements. However, if another developer concurrently changes or adds model elements, the effect of the composite operation might be mitigated because the concurrent operations have not been considered in the original application of the composite operation. As already mentioned, composite operation specifications comprise detailed preconditions and the application of a composite operation affects model elements that fulfill or *match* the preconditions. If concurrent operations applied by another developer modify the model so that this match is influenced, we may either face an

operation-based conflict, or we may encounter valid preconditions and an increase of the match size. Thus, the composite operation application is still valid, however, more model elements match the preconditions after the concurrent operations have been applied than before. Therefore, developers should be notified in terms of a *warning* that these additionally matching model elements might be also incorporated in the composite operation application. An example that illustrates such a scenario is presented in Section 3.1.4.

Signifiers. Adopting the notion of signs and signifiers in linguistics [DS16], we introduce the term *signifier* to refer to one or more intrinsic or extrinsic properties of a model element that convey the superior meaning of the respective model element. For instance, the meaning of a UML operation is mainly conveyed by its name, its return type, and the types of its contained parameters; thus, the signifier of a UML operation is a combination of its name and return type, as well as the types of its contained parameters. These properties, constituting the *signifier* of a model element, may overlap with the *natural identifier* of the model element such as its name. However, a natural identifier is usually only *one intrinsic* property. A signifier, on the contrary, may additionally incorporate *multiple* properties, which may also come from its context such as its child model elements, its container, or cross-referenced model elements. As these properties are particularly important for the meaning of a model element, we argue that they should be treated specifically in the merge process. Therefore, we introduce two types of warnings related to signifiers in the following.

Unexpected Signifier Match. An unexpected signifier match indicates scenarios in which two model elements, which have either been added or modified, eventually have the same signifier; that is, they share the same meaningful properties. If these two model elements are completely equal as in the scenario in Section 3.1.1, we may safely remove one of those added or modified model elements to avoid redundancies in the merged model. If, however, the model elements indeed have the same signifier, but are not entirely equal, a decision of the developer is needed to verify if both model elements should be retained or how they should be joined (cf. Section 3.1.2 for an example). Such scenarios are referred to as *unexpected signifier match*.

Concurrent Signifier Change. Besides having new signifier matches, we may also face the opposite case. One model element is modified concurrently so that the signifier is affected contradictorily in both revised models; that is, after the concurrent modifications, the corresponding model elements have different signifiers. In such scenarios, which are referred to as *concurrent signifier change*, it is likely that the model elements meaning is obfuscated and, therefore, developers should be warned and review the merged model. An example for such a scenario is given in Section 3.1.3.

3.3 Design Principles of AMOR

In this section, we discuss the basic design principles of the adaptable model versioning system AMOR. Subsequently, we discuss several fundamental techniques with regard to these principles and document the reasons behind the design decisions made when developing AMOR.

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

Therefore, AMOR is designed to provide generic versioning support irrespective of the used modeling languages and modeling editors. Generic model versioning can be achieved by using one of two alternatives. The first alternative is having an internal representation of models, which are put under version control. This internal representation must be capable of expressing every piece of information that is also available in the original model. The implementations of the versioning system may then be designed to work with models conforming to the internal representation and are consequently independent of the original modeling language. However, this requires the existence of a bi-directional transformation between models conforming to a specific modeling language and models conforming to the internal representation. Specifying these transformations might be a tedious task. Therefore, we use an alternative way of realizing a language-independent system, which is actually used by several other generic model versioning systems. Instead of translating every model into an internal representation, we use the reflective interfaces of the Eclipse Modeling Framework [SBPM08] (EMF). Thereby, all modeling languages can be handled immediately that are supported by the chosen metamodeling framework (i.e., a metamodel is specified in terms of the metamodeling framework's meta-metamodeling language). By choosing a popular metamodeling framework, a plethora of modeling languages can be handled at one stroke. Of course, this only allows to deal with modeling languages for which a metamodel (conforming to the supported meta-metamodel) is available. Nevertheless, it is always possible to develop a transformation from the models defined in the "foreign" metamodeling framework into a corresponding new or existing metamodel conforming to the supported metamodeling framework.

A model versioning system that is also independent of the used modeling editor must not make any assumptions on how a model is manipulated by users and must not rely on specific features on the editor side. Therefore, we may not apply editor-specific operation recording to obtain the applied operations. Instead, AMOR works only with the states of a model before and after it has been changed and derives the applied operations using state-based model differencing.

Easy adaptation by users. Generic versioning systems are very flexible, but they lack in precision in comparison to language-specific versioning systems because no language-specific knowledge is considered (cf. Section 3.1 for examples in which language-specific knowledge is required). Therefore, a generic versioning system should be adaptable with language-specific knowledge whenever this is needed. Some existing model versioning approaches are adaptable in terms of programming interfaces. Hence, it is possible to implement specific behavior to adapt the system according to their needs (i.e., white-box adaptation as discussed in Section 2.2). Especially with domain-specific modeling languages, a plethora of different modeling

languages exists, which often are not even publicly available. Bearing that in mind, it is hardly possible for versioning system vendors to pre-specify the required adaptations to incorporate language-specific knowledge for all existing modeling languages. Thus, users of the versioning system should be enabled to create and maintain those adaptation artifacts by themselves. This, however, entails that these adaptation artifacts do not require deep knowledge on the implementation of the versioning system and programming skills. In other words, *black-box adaptations* should be preferred over white-box adaptations and the adaptation artifacts should be created in a *descriptive language* that is easy to use (cf. Section 2.2).

Therefore, AMOR is designed to be adapted by providing descriptive adaptation artifacts and uses, as far as possible, well-known languages to specify the required language-specific knowledge. No programming effort is necessary to enhance AMOR's versioning capabilities with respect to language-specific aspects. Besides aiming at the highest possible adaptability, the *ease of adaptation* is one major goal of AMOR. Thus, for one of the most complicated adaptation points (i.e., the specification of composite operations), we introduce a novel technology named *model transformation by demonstration* (cf. Chapter 4) to achieve this goal.

Don't Repeat Yourself. Adapting a software system to one's specific needs is often a great deal of work. Besides gathering the requirements and identifying the right adaptation points for realizing those requirements, also specifying the correct adaptation artifact might be a time-consuming task. This effort can be counteracted by easing the approach and providing appropriate tool support to create the adaptation artifact, but also by not forcing developers to specify repeatedly the same piece of knowledge over and over again. This principle is also known as *Don't Repeat Yourself* (DRY) and has been introduced by Hunt and Thomas [HT00]. In particular, they state that "every piece of knowledge must have a single, unambiguous, authoritative representation within a system".

In AMOR, we adopt this principle in order to aim at reducing the adaptation effort. More precisely, we designed AMOR to exploit user-specified match rules for improving the model matching, but also for enabling the system to detect *unexpected signifier matches* and *concurrent signifier changes* (cf. Section 3.2). Furthermore, a user-specified composite operation specification allows for automatic execution of the composite operation, but also for the a posteriori detection of its applications and for detecting *composite operation conflicts* and *composite operation match warnings* (cf. Section 3.2). Finally, we reuse the constraints of a modeling language's abstract syntax specification to reveal inconsistent states after merging.

3.4 Technical Infrastructure of AMOR

In this section, we give a brief overview of the technical infrastructure of this thesis. The concepts presented in this thesis can be ported to any platform and metamodeling framework. However, the concepts have been implemented in terms of Eclipse plug-ins [BG03, CR04, Hol04] and are elaborated in the context of EMF [SBPM08]. When describing models in the following, we refer, in particular, to EMF-based models. Furthermore, for realizing the contributions presented in this thesis, we reuse and extend the model comparison framework EMF Compare [BP08]

and integrate the Epsilon Comparison Language (ECL) [Kol09]. Therefore, we provide a short overview of these technologies in the following.

3.4.1 Eclipse and Eclipse Plug-ins

Eclipse[3] [BG03, CR04, Hol04] is an open-source[4], Java-based software development environment with the goal of providing a generic platform for bundling integrated development environments (IDE). The most popular bundle is the Eclipse IDE for Java. Besides this bundle, there are numerous other bundles for several programming languages or other application domains. These bundles range from environments for report development to IDEs for an extensive set of diverse programming languages such as C++, Ruby, PHP, and many others.

The development of Eclipse is organized by an independent consortium consisting of many companies and organizations. Its implementation is performed by thousands of professional and independent developers spread all over the world. This diversity of stakeholders and developers has led to a very powerful, flexible, and extensible platform and a variety of features. A project of such a size and complexity may hardly be organized by one single organization unit. Therefore, the development of Eclipse is divided into three main projects having distinct responsibilities and a specific focus. These three projects are *(i)* the Eclipse Project, *(ii)* the Tools Project, and *(iii)* the Technology Project. Each of these main projects consists of a range of subprojects such as the Java Development Tools Project (JDT), the C/C++ Development Tools Project (CDT), and many others (cf. [SBPM08]).

It is worth noting that the main goal of Eclipse is not to provide an IDE for a specific set of programming languages. It rather aims at offering a platform that enables to develop every kind of IDE for every kind of language. For achieving this goal, the core of Eclipse is designed to be a runtime system that manages and loads plug-ins. This runtime system is a component-based system called *Equinox*, which is an implementation of the *Open Services Gateway initiative* (OSGi) [OSG03]. Every plug-in contributes a set of features by providing its own implementations or bundling, and composing features of other plug-ins. A plug-in consists of all artifacts required to realize the set of features. This comprises the compiled source code, interface definitions, image resources, dependencies to other plug-ins, etc. The central declaration of a plug-in is the so-called *plugin.xml*, which wraps the following information (as stated in [SBPM08]).

- **Requires**: Dependencies to external libraries and libraries provided by other plug-ins.
- **Exports**: Visibility of its own public classes, which can be called by other plug-ins.
- **Extension Points**: Public declaration of interfaces that can be used by other plug-ins to extend the behaviour of its own plug-in.
- **Extensions**: Public declaration of the implementations that are contributed by this plug-in to other plug-ins (i.e., extensions extending foreign extension points).

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

[4] Eclipse Public License (EPL): `http://www.eclipse.org/legal/epl-v10.html`

3.4.2 (Meta-)Modeling with EMF and Ecore

When describing models in this thesis, we refer to models that are based on EMF. EMF is a matured Eclipse-based framework providing powerful metamodeling support within the Eclipse ecosystem. EMF has found significant recognition among researchers and practitioners, which is also why we chose EMF as the underlying modeling technology. EMF offers, besides the meta-metamodeling language Ecore (introduced below), facilities for code generation, generation of modeling editors, reflective APIs to access and manipulate models generically. Based on EMF, many very powerful technologies have been built, which allow, for instance, to persist models in relational databases, to transform models, and much more. In the following, however, we focus on introducing the metamodeling language Ecore and discuss its relationship to the well-known metamodeling stack [Küh06].

The heart of EMF is its metamodeling language *Ecore*, a Java-based implementation of the Essential Meta Object Facility (EMOF) [OMG04] standardized by the Object Management Group (OMG). Using Ecore, developers may specify a metamodel to define the abstract syntax of a new modeling language. This metamodel may then be used to generate modeling editors for creating models, that is, instances of the developed metamodel. The relationship among meta-metamodels, metamodels, and models may best described in terms of the metamodeling stack [Küh06]. The metamodeling stack consists of three layers called *M3*, *M2*, and *M1* whereas a model in *M2* conforms to a model in *M3* and a model in *M1* conforms to a model in *M2*.

M3: Meta-metamodel. In the most upper layer in the metamodeling stack, namely *M3*, the meta-metamodeling language is located (cf. Figure 3.9). In the context of EMF, this meta-metamodeling language is Ecore. The core language elements of Ecore are depicted in the upper area of Figure 3.9 in terms of a UML class diagram. Please note that we do not present all language elements and features in this figure. Instead, we concentrate on those classes and features that are of paramount importance in the current context. Ecore allows to model EClasses, which may contain an arbitrary number of structural features. For structural features, upper and lower multiplicities have to be defined. Additionally, structural features having an upper multiplicity greater than 1, may be defined as ordered. Structural features are divided into two distinct subsets, namely EReferences and EAttributes. Attributes as well as references must have a type. For attributes, primitive data types such as String, Boolean, and Integer are allowed. References refer to classes for defining their types and may additionally be defined as containments. This means that referenced elements are nested inside the container element and, therefore, the deletion of a container element results in cascaded deletions of all directly and indirectly contained elements. It is worth noting that Ecore is recursively specified by Ecore. This means that, for example, EReference is indeed an instance of EClass having the name "EReference". This class contains, for instance, the structural feature "containment", which is an instance of EAttribute and more.

M2: Metamodel. The meta-metamodeling language may now be used to create metamodels. A metamodel specifies the abstract syntax of a modeling language and is an instance of Ecore, which resides in *M3*—therefore, a metamodel resides on *M2*. In Figure 3.9, we provide a small example of such a metamodel in terms of an object diagram. In particular, this metamodel is

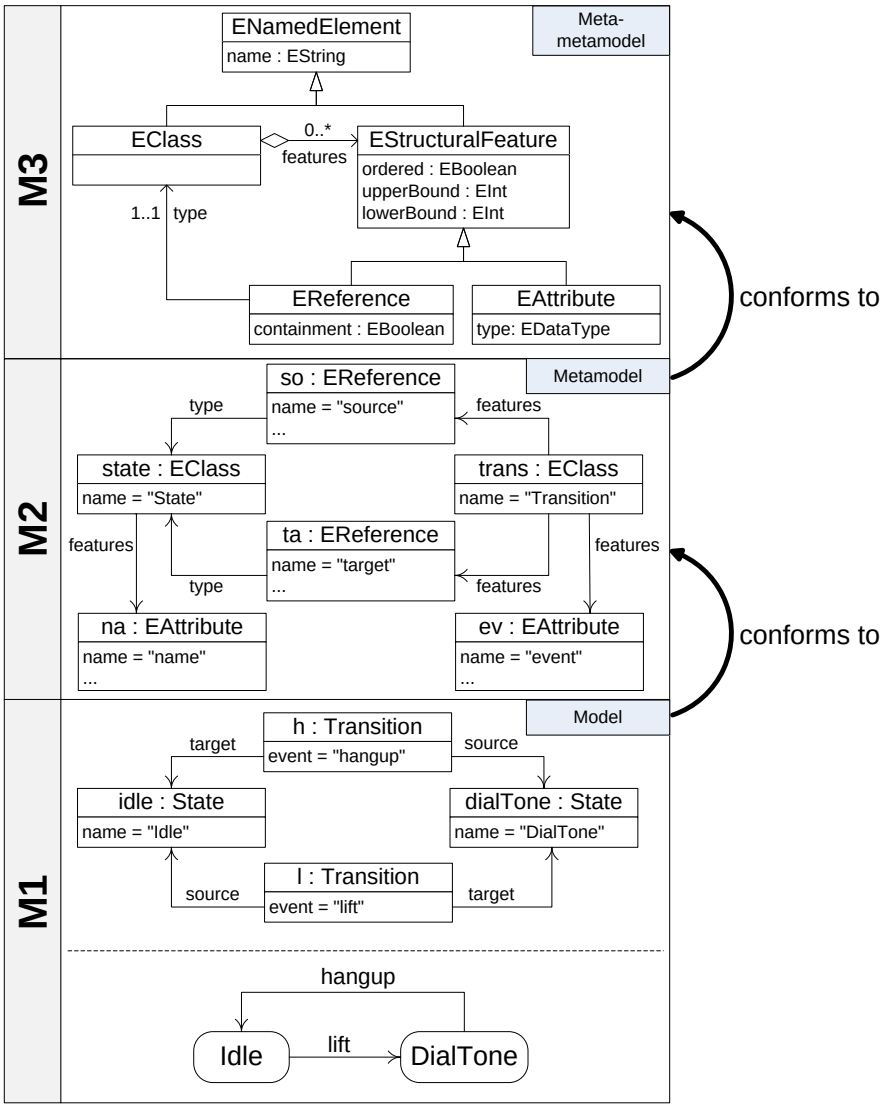


Figure 3.9: Metamodeling with Ecore

a simplified excerpt of the state machine metamodel. A state machine consists of States and Transitions. Therefore, we have two instances of Ecore's EClass, one for states and one for transitions. Both classes contain an attribute (i.e., an instance of Ecore's EAttribute): a state has a name and a transition has an event. Transitions further refer to the source state and the target state. Therefore, the metamodel for state machines contains two instances of EReferences, namely source and target.

M1: Model. The metamodel in *M2* may now be instantiated to specify arbitrarily many state machines on *M1*. In Figure 3.9, we illustrate a small state machine comprising two states and two

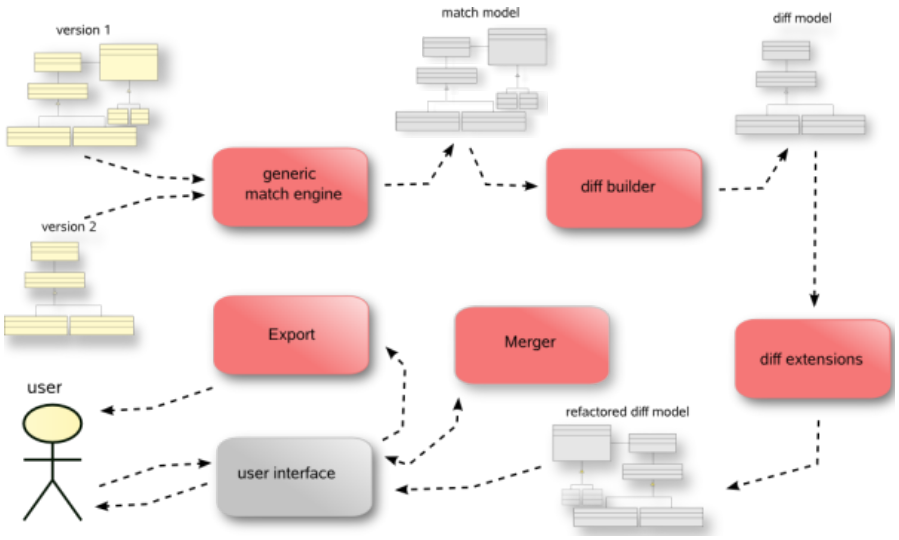


Figure 3.10: EMF Compare Architecture [EMC]

transitions between those states. More precisely, the states are instances of the corresponding class State in the metamodel residing in *M2*. In the upper area of *M1* in Figure 3.9, the small state machine model is depicted in terms of an object diagram and in the lower area of *M1*, the same model is illustrated by the commonly used concrete syntax of state machines for the sake of readability.

3.4.3 EMF Compare

EMF Compare[5] [BP08] is a subproject of the Eclipse Modeling Framework Technology project (EMFT) and provides an extensible tool and framework for model comparison and merging. Therefore, we also considered EMF Compare in the discussion of existing work related to the topics of this thesis in Section 2.1.2.

EMF Compare supports two-way and three-way model comparison. The model comparison process is divided in a two-phased process: the match phase and the differencing phase (cf. Figure 3.10). In the match phase, the so-called *generic match engine* aims to identify corresponding model elements among two or three versions of a model by either a UUID-based or heuristics-based match. Having obtained the correspondences, they are saved into a *match model*. Based on this match model, the so-called *diff builder* compares each set of corresponding elements and computes the fine-grained differences at the feature level. The computed differences are saved into a *diff model*. The resulting diff model may be optionally "refactored" by user-specified implementations of the *diff extension* interface. The goal of these diff extensions is to allow for improving the structure of the abstract diff model according to some language-specific rewriting rules. In some modeling languages, one change from the user perspective results in several differences from a generic perspective, as for instance one element in the concrete syntax is

[5] `http://www.eclipse.org/emf/compare`

represented by several elements in the abstract syntax. The diff extension allows to improve the comprehensibility of the diff model by using language-specific diff extensions, which search for specific difference elements and group them into one difference element accordingly. Additionally, EMF Compare offers user interfaces for visualizing match and difference models, provides extension points for *export* difference models into reports, and also allows to *merge* models by applying difference elements from a diff model to the input models.

To summarize, EMF Compare is a very flexible and extensible framework that can be used for any tasks related to model comparison. Hence, AMOR heavily makes use of the extensions offered by EMF Compare. In particular, AMOR replaces the match engine provided by EMF Compare with our own implementation and uses only EMF Compare's *diff builder*. For incorporating applications of composite operations in the diff model, AMOR exploits the *diff extension* interface (cf. Chapter 5). Furthermore, EMF Compare's *merger* is the basis for AMOR's model transformation engine that is used to execute composite operations (cf. Chapter 4) and for merging models in AMOR.

3.4.4 Epsilon Comparison Language

The Epsilon Comparison Language[6] (ECL) [Kol09, KRP11] is a domain-specific language for developing model comparison rules. ECL is part of the Epsilon project, which is a family of interoperable task-specific languages for working with EMF models. In particular, the Epsilon project provides languages for code generation, model-to-model transformation, model validation, comparison, migration, merging, and refactoring. The aim of ECL is to enable the specification of language-specific comparison algorithms in a rule-based manner. Thereby, ECL can be used to identify pairs of matching elements between two models conforming to the same or even different metamodels. ECL supports inheritance among match rules, recursive calls of rules using the function matches, rule guards, which can be used to restrict the execution of a rule in certain scenarios, as well as lazy rules, which are only manually invoked. Furthermore, ECL allows to specify custom operations, which can be called from several rules. Another very distinguished feature for a domain-specific language is that existing external libraries may be called from ECL rules. Thereby fuzzy string matching frameworks or dictionaries such as WordNet[7] can be integrated easily in an ECL rule system.

The concrete syntax specification for ECL match rules is provided in Listing 3.1 and an example for a match rule, which uses an external library for string matching, is given in Listing 3.2. In this example, the rule FuzzyTree2Tree matches two instances of the metaclass Tree if their label is similar to a certain degree in terms of the Levenshtein [Lev66] distance, as specified in the operation fuzzyMatch, and if their parents match. To verify whether the parents match, the generic function matches can be called recursively from any rule.

In the course of this thesis, we show how ECL is integrated in our versioning framework to allow users to plug in language-specific match rules to improve the generic UUID-based matching (cf. Chapter 5). Furthermore, ECL rules are involved in the conflict detection approach presented in this thesis to reveal merge issues concerning similar model elements as shown in

[6] http://www.eclipse.org/gmt/epsilon/doc/ecl
[7] http://wordnet.princeton.edu

Listing 3.1: Concrete Syntax of a Match Rule [KRP11]

```
1  (@lazy)?
2  (@greedy)?
3  (@abstract)?
4  rule <name>
5    match <leftParameterName>:<leftParameterType>
6    with <rightParameterName>:<rightParameterType>
7    (extends (<ruleName>,)*<ruleName>)? {
8    (guard (:expression)|({statementBlock}))?
9    compare (:expression)|({statementBlock})
10   (do {statementBlock})?
11 }
```

Listing 3.2: Example of a Match Rule using Fuzzy String Matching [KRP11]

```
1  pre {
2    var simmetrics =
3      new Native("org.epsilon.ecl.tools.
4        textcomparison.simmetrics.SimMetricsTool");
5  }
6
7  rule FuzzyTree2Tree
8    match l : T1!Tree
9    with r : T2!Tree {
10   compare : l.label.fuzzyMatch(r.label) and
11     l.parent.matches(r.parent) and
12     l.children.matches(r.children)
13 }
14
15 operation String fuzzyMatch(other : String) : Boolean {
16   return simmetrics.similarity(self,other,"Levenshtein") > 0.5;
17 }
```

the motivating scenario in Section 3.1.2, as well as concurrent operations that contradictorily modify the signifier of model elements as shown in the motivating scenario in Section 3.1.3. For more information on how ECL is integrated to improve conflict detection, we kindly refer to Chapter 6.

3.5 Adaptable Merge Process of AMOR

In this section, we first introduce the generic merge process of AMOR and, subsequently, we show how this process is extended in this thesis so that it may incorporate language-specific knowledge. In particular, we discuss the adaptable components in this extended merge process and its adaptation points, which may be used for enhancing the quality of the operation and conflict detection.

3.5.1 Generic Merging in AMOR

With the generic merge process, AMOR offers generic versioning support for every EMF-based model without requiring users to perform any kind of adaptation (i.e., out of the box). All components in this generic process are designed to be model metamodel agnostic and operate only on the reflective API provided by EMF.

The generic merge process is depicted in Figure 3.11. This figure presents a more fine-grained view on the same merge process that was introduced in Figure 1.2. Furthermore, we now illustrate explicitly the artifacts that are exchanged between the steps of this process. The input of this merge process are three models: the common original model V_o and two concurrently changed models, V_{r1} and V_{r2}. Thus, V_{r1} is the result of the first modification m_1 performed by developer 1 and V_{r2} is the result of the second modification m_2 performed by developer 2.

UUID-based Matching. The first step of the merge process in Figure 3.11 is the UUID-based matching. The goal of this step is to identify the corresponding model elements between V_o and V_{r1}, as well as between V_o and V_{r2}. As the merge process aims to be generic, no language-specific correspondence rules are used. Instead, this match algorithm assumes that there are immutable UUIDs attached to each model element, which are used to map unambiguously each model element in V_o to its respective counterpart in V_{r1} and V_{r2}. The obtained correspondences are saved into two distinct *match models*. The first match model $M_{V_o,V_{r1}}$ represents the correspondences between V_o and V_{r1}, and the second match model contains the mappings between V_o and V_{r2}.

Atomic Operation Detection. The goal of the next step is to identify the atomic operations that have been applied to the common original model V_o in order to obtain the revised models, V_{r1} and V_{r2}. Therefore, in this step, each pair of corresponding model elements in the match model is compared to each other. In particular, each feature value of both corresponding model elements is checked whether they are equal or not. If they are not equal, a corresponding operation is derived and saved into a so-called *diff model*. Additionally, for each model element in

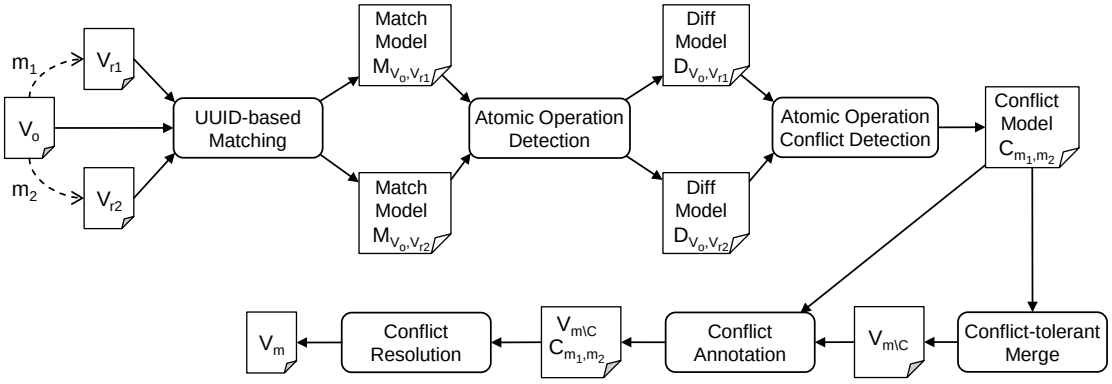


Figure 3.11: Generic Merge Process of AMOR

the revised model that has no corresponding model element in the original model, an operation element representing the addition is saved to the diff model. Accordingly, an operation element representing a deletion is saved for the opposite case. This step is performed for both match models, $M_{V_o,V_{r1}}$ and $M_{V_o,V_{r2}}$, in order to create the two diff models, $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$. Ultimately, these diff models, $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$, contain all operations that have been performed in the course of the modification m_1 and m_2, respectively.

This concludes the two-phased operation detection for atomic operations in the merge process. These two phases are elaborated in more detail in Chapter 5. Having identified all atomic operations that have been performed concurrently, we may now proceed with identifying conflicts among these operations.

Atomic Operation Conflict Detection. The input of the atomic operation conflict detection are two diff models, one for each revised model. These two diff models are now analysed to detect overlapping atomic operations (cf. Section 3.2). To reveal such cases, for each operation contained by one diff model, it is checked whether an operation exists in the opposite diff model that is parallel dependent or non-commutative. Finally, each detected conflict is saved to a *conflict model* (cf. C_{m_1,m_2} in Figure 3.11). A detailed discussion of the atomic operation conflict detection is provided in Chapter 6.1.

Conflict-tolerant Merge. As argued in [BLS+10b] and further elaborated in [Wie11], resolving conflicts directly in a preliminarily merged model is easier and more natural than resolving conflicts by choosing one of the conflicting operations that should be applied from a list of conflicting operations. Therefore, the conflict-tolerant merge produces a model, called $V_{m\setminus C}$, to which all operations of both developers are applied that are not in conflict with another operation.

Conflict Annotation. In the next step, the preliminarily merged model $V_{m\setminus C}$ is annotated with all conflicts in C_{m_1,m_2} that need to be resolved. For annotating models independently of their metamodel, we introduced a novel mechanism called EMF Profiles in [LWWC11], which ports the light-weight language extension mechanism known from UML Profiles [FFVM04] to domain-specific models in EMF. For annotating conflicts, we developed a dedicated conflict profile [BKL+11b], which is used to indicate merge conflicts directly in the merged model $V_{m\setminus C}$. The annotated merged model, referred to as $V_{m\setminus C}\ C_{m_1,m_2}$ in Figure 3.11, is handed over to the next step.

Conflict Resolution. Having annotated all previously detected conflicts in the merged model, the user may resolve the conflicts directly in the model. In [BKS+10] and [BSW+09], we presented more automated or supportive ways for users to resolve conflicts by introducing conflict resolution recommendations and collaborative, synchronous modeling tools for conflict resolution, respectively. The topic of conflict resolution is also further elaborated in [Bro11]. After all conflicts have been resolved, the new merged model V_m is saved in the common repository.

The presented merge process provides reasonable versioning support that is comparable to the quality of state of the art such as [KHWH10]. However, the generic process is not able to handle correctly the model versioning scenarios presented in Section 3.1.

3.5.2 Adaptation Points of the Merge Process

Having introduced the generic merge process, we show how this process is extended to allow for its adaptation with respect to language-specific knowledge. The extended adaptable merge process depicted in Figure 3.12 aims at correctly handling the challenging model versioning examples presented in Section 3.1. In the following, we discuss the reasons behind the new steps in the merge process and provide a brief overview of the functionality of the introduced adaptable steps.

Accuracy of the atomic operation detection. The accuracy of the atomic operation detection is crucial for all succeeding tasks in the merge process. In this context, the accuracy can be specified in terms of precision and recall as defined by Olson and Delen in [OD08]. These terms, precision and recall, originally stem from the area of information retrieval and denote the completeness of pattern recognition algorithms. If the operation detection lacks in precision, a succeeding conflict detection phase might raise incorrect conflicts. The main reason for a lack of precision in the operation detection when using state-based model differencing lies in a lack of precision of the model matching phase. Consider for instance, developer 1 modifies the name of a model element and developer 2 adds a new containment to the same model element. If the model matching component is not capable of matching the model element in the original model with the corresponding model element in the revised model of developer 1 because of the different name, a deletion of that model element is reported as well as an addition of another (actually the same) model element having the new name. Consequently, a delete-update conflict is reported because developer 2 added a new containment to the model element that has been incorrectly considered as removed. If the operation detection provides a low recall (i.e., some applied operations have not been detected), the succeeding conflict detection might also miss detecting some important conflicts. To summarize, high precision and recall of the model matching and the operation detection is an essential prerequisite for high-quality conflict detection.

Perhaps the most accurate way of obtaining the applied operations among model versions with model differencing algorithms is to use UUIDs. UUID-based matching, however, completely neglects the contents (i.e., its properties, references, and containments) of a model element. However, in some scenarios, the content is an important source of information for obtaining the precise operations. For instance, if a model element has been deleted and a new model element having similar properties as the deleted one has been added again (e.g., cut and pasted elements), UUID-based approaches are not able to establish correct correspondences. The same is true for equal or at least similar model elements that have been added concurrently by different developers as is the case in the model versioning scenarios presented in Section 3.1.1 and in Section 3.1.2. Therefore, we introduce a new matching step after the UUID-based matching, named rule-based matching, with the goal of improving the match models ($M_{V_o,V_{r1}}$ [UUID] and $M_{V_o,V_{r2}}$ [UUID]) obtained from UUID-based matching. This improvement is achieved by

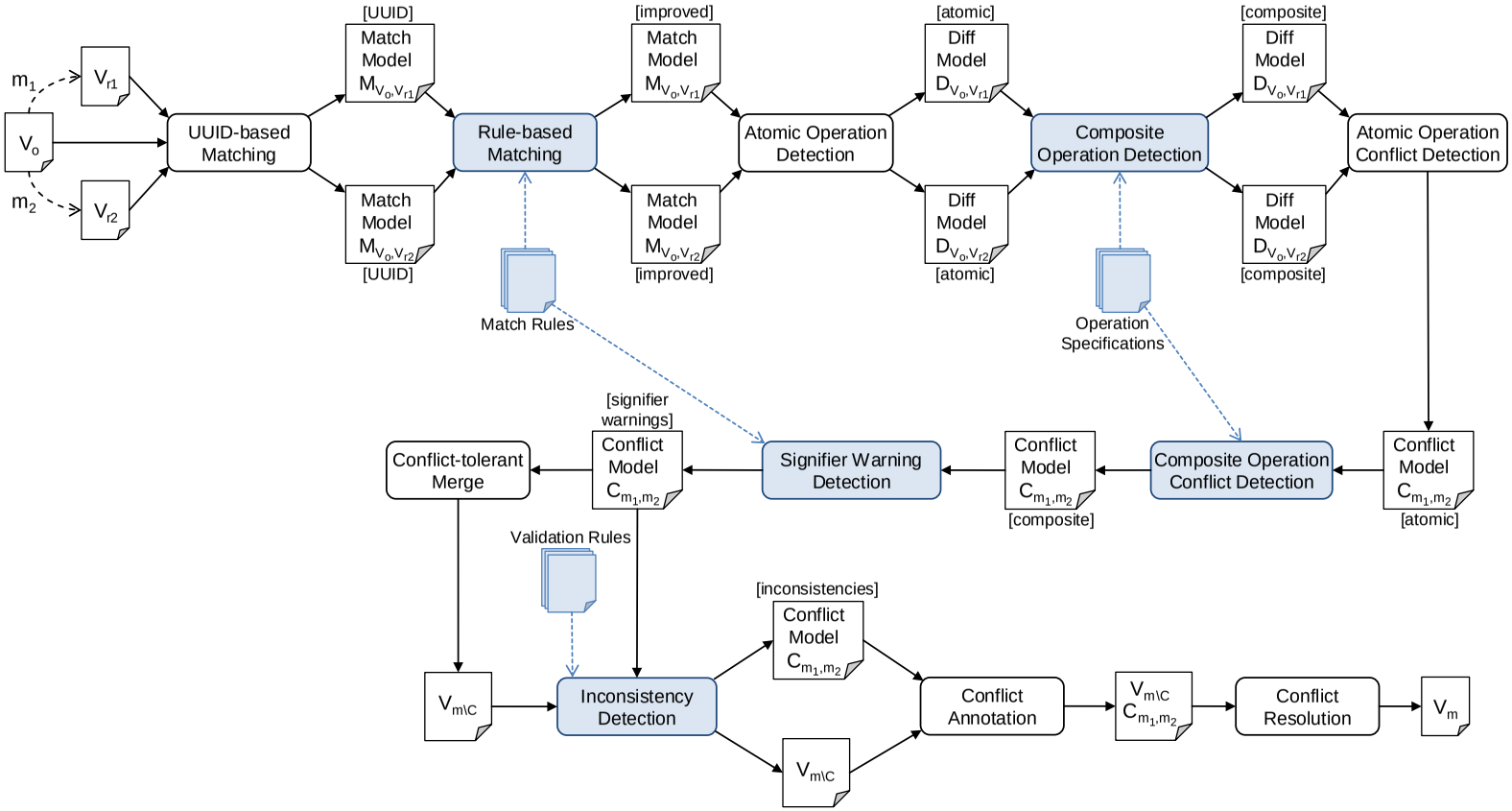


Figure 3.12: Adaptable Merge Process of AMOR

using content-based heuristics to find corresponding model elements that *could not be matched using UUIDs*. Consequently, we aim at combining the advantages of UUID-based and content-based matching methods. The specific properties of a model element that should be used for matching two model elements, however, are specific to the modeling language. Therefore, we introduce an adaptation point that allows users to specify match rules for a certain modeling language, which are interpreted by the rule-based matching component to improve the match model. The improved matches are incorporated in the match models ($M_{V_o,V_{r1}}$ [improved] and $M_{V_o,V_{r2}}$ [improved] in Figure 3.12) and are handed over to the next step. For more information on the adaptable rule-based matching, we kindly refer to Chapter 5.

Composite operations. The next extension to the generic merge process concerns composite operations. As illustrated in the versioning scenarios in Section 3.1.4 and 3.1.5, the knowledge on applications of composite operations between two versions of a model significantly helps in many scenarios to better respect the original intention of a developer, as well as to reveal additional issues when merging two concurrent modifications.

The prerequisites for considering applications of composite operations is to detect them. When using state-based model differencing algorithms, this is a challenging task because only two succeeding versions of a model are available. To address this challenge, we introduce the new step, composite operation detection, immediately after the step for detecting atomic operations in the extended merge process, as depicted in Figure 3.12. This step takes two diff models, $D_{V_o,V_{r1}}$ [atomic] and $D_{V_o,V_{r2}}$ [atomic], which contain a set of applied atomic operations, as input. These two diff models are analyzed to find occurrences of specific *diff patterns* within them. Having found such a diff pattern, the pre- and postconditions of the respective composite operation are evaluated; if these conditions hold for a certain pattern, an application of the respective composite operation is detected and saved to the input diff models. The detection of user-specified composite operations among atomic operations in a diff model is presented in Section 5.3. The diff models, enriched with the obtained information on applied composite operations ($D_{V_o,V_{r1}}$ [composite] and $D_{V_o,V_{r2}}$ [composite] in Figure 3.12) are handed over to the next step.

For detecting conflicts caused by violated preconditions or issues concerning the original intention behind the composite operation, we installed the new step called composite operation conflict detection in the merge process after the step for detecting atomic operation conflicts. This step is based on the previously detected applications of composite operations and checks for each application whether concurrent operations affect the validity of the preconditions and whether more model elements match the preconditions after the concurrent operations than before. The former is done to detect composite operation conflicts and the latter allows to detect composite operation match warnings (cf. Section 3.2). If such a merge issue is detected, a corresponding conflict or warning description is added to the input conflict model C_{m_1,m_2} [atomic].

Composite operations are inherently specific to a certain modeling language. Therefore, the composite operation detection and the composite operation conflict detection are designed to be adaptable for new modeling languages by allowing users to add new operation specifications. For creating such operation specifications, we introduce a novel approach called *model transformation by demonstration* in Chapter 4. According to our principle, *don't repeat your-*

self, such specifications contain the information necessary to detect their applications, which is presented in Chapter 5, as well as the information that is needed to detect composite operation conflicts and composite operation match warnings. The detection of such conflicts and warnings is presented in Chapter 6.2.

Signifiers. The importance of considering signifiers of model elements is illustrated in the versioning scenarios presented in Section 3.1.2 and 3.1.3. For addressing such issues, we introduce the step called signifier warning detection in the merge process depicted in Figure 3.12. This component searches for added or changed model elements in both modifications, m_1 and m_2, that unexpectedly have matching signifiers, as well as for concurrent operations that both change the signifier of the same model element in a contradicting manner; in other words, this component aims to detect *unexpected signifier matches* and *concurrent signifier changes* (cf. Section 3.2). If such issues are detected, the input conflict model C_{m_1,m_2} [composite] is extended by additional warning descriptions. The resulting conflict model C_{m_1,m_2} [signifier warning] is handed over to the next step. Thereby, scenarios like those presented in Section 3.1.2 and 3.1.3 can be detected to avoid unfavorable redundancies and unintended obfuscations of existing model elements. Which properties of a model element's metaclass have to be combined in order to obtain the signifier cannot be derived generically from a metamodel. Therefore, this component is adaptable to allow users to specify the signifier specifications according to their own needs. For this specification, we reuse the technology used for adapting the model matching phase; that is, language-specific match rules. For more information on detecting merge issues in the context of signifiers, we kindly refer to Chapter 6.3.

Inconsistencies. Finally, we introduce a new step addressing the consistency of the resulting merged model in the adaptable merge process depicted in Figure 3.12. This step, called inconsistency detection, is situated after the conflict-tolerant merge and validates the preliminary merged version $V_{m\setminus C}$ against language-specific validation rules to reveal inconsistencies that are inadvertently introduced by the merge. An example for such an inconsistency is shown in the versioning scenario in Section 3.1.6. Such consistency rules are specific to the modeling language and are usually specified along side the metamodel by the language designer. Thus, we reuse the consistency rules coming from the language definition and apply the *EMF Validation framework*[8] for validating the merged model. This framework supports validation rules specified in the Object Constraint Language (OCL) [OMG10], as well as rules programmed in Java. If inconsistencies in the merged model $V_{m\setminus C}$ are found, these are added to the input conflict model C_{m_1,m_2} and passed on to the next step in the process. For more information on detecting model inconsistencies, we kindly refer to Chapter 6.4.

The remaining steps of the adaptable merge process are the same as in the generic merge process. In these remaining steps, the preliminarily merged model $V_{m\setminus C}$ is annotated with all detected conflicts in C_{m_1,m_2}. Finally, the annotated model is passed to the user in order to resolve the raised conflicts and review the annotated warnings. Eventually, the resulting model V_m is finally saved to the repository.

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

CHAPTER 4

Model Transformation By Demonstration

Predefined composite operations are helpful for efficient modeling, in particular, for automatically executing recurring refactorings, applying model completions, and introducing patterns to existing models. Moreover, the availability of explicit specifications of composite operations (comprising pre- and postconditions as well as the atomic operations to be applied) is the prerequisite for adequately considering applications of such operations in the merge process.

Composite operations are tailored specifically for a certain modeling language. As domain-specific modeling is becoming more important, a plethora of different modeling languages exist. Consequently, it is infeasible to predefine all relevant composite operations for all modeling languages being used in practice. Therefore, users of a certain modeling language themselves should be enabled to specify such composite operations on their own so that these specifications can be used for automatically executing the specified composite operations, but also for adapting a model versioning system as outlined in Chapter 3. Composite operations are, in more general terms, *endogenous model transformations* [MG06] (cf. Section 2.3). Thus, an approach is needed that allows users to develop easily such endogenous model transformations to represent composite operations.

For specifying model transformations, several dedicated languages (cf. [CH06] for an overview) have been developed in the last decade. Most of them are based on the *abstract syntax* as defined in the metamodel makes it difficult for common users of modeling languages to specify model transformations, because they are usually unfamiliar with the abstract syntax as they mainly work with the *concrete syntax* of the modeling languages (i.e., their notation) and not with its metamodel [SW08, Var06]. This is aggravated by the fact that metamodels may become very large. For instance, the UML 2 metamodel [OMG03] has about 260 metamodel classes [MSZJ04]. Moreover, some language concepts, which have a particular representation in the concrete syntax, are not even explicitly represented in the metamodel. Instead, these concepts are hidden in the metamodel and may only be derived by using specific combinations of attribute values and links among model elements [KKK+07].

To address this problem, we introduce a novel approach for specifying endogenous model transformation more easily using the concrete syntax. The increased ease of use is achieved by applying an approach called *model transformation by demonstration* (MTBD). In MTBD, users apply or "demonstrate" the transformation to an example model once and, from this demonstration as well as from the provided example model, the generic model transformation is semi-automatically derived. Please note that at the time when we published our approach for specifying composite operations by demonstration in [Lan09, BLSW09, BLS+09], a very similar approach by Sun et al. [SWG09] emerged[1]. Thereby, Sun et al. introduced the notably suitable term *model transformation by demonstration* for such demonstration-based specification approaches. Thus, in the remainder of this thesis, adopt this term.

For model versioning purposes, endogenous model transformations are of major importance. Therefore, we focus on specifying endogenous model transformations in Section 4.1. However, in Section 4.2, we also show how the idea behind MTBD can be extended to also enable the specification of *exogenous model transformations*. Finally, in Section 4.3, we discuss current limitations of our MTBD approach for endogenous as well as for exogenous model transformations and highlight some potential research directions to be addressed in the future.

4.1 Endogenous Model Transformation By Demonstration

Our MTBD approach for specifying endogenous model transformation, called *Eclipse Modeling Operations*[2] (EMO), is designed according to the principles of AMOR (cf. Section 3.3). More precisely, EMO aims at enabling users who are not trained in model transformation languages and who are unfamiliar with the modeling language's metamodel, to specify endogenous model transformations, called composite operations hereafter, without posing any restrictions regarding the modeling language and modeling editor.

In the following, we first introduce an exemplary composite operation in Section 4.1.1 serving as a running example for the remainder of this section. Subsequently, we give an overview of the basic idea behind EMO in Section 4.1.2 and present the specification process in more detail in Section 4.1.3 by means of solving the running example. In Section 4.1.4, we examine the concept of *templates* and their bindings to model elements and in Section 4.1.5, we show how developed composite operations are executed to arbitrary models. In Section 4.1.6, we introduce advanced features of our approach for also addressing more complex composite operations. Finally, we discuss the related work in the area of MTBD in Section 4.1.7 and point to some possible directions for future work for endogenous as well as for exogenous model transformations by demonstration in Section 4.3.

Please note that we present an evaluation of our MTBD approach for endogenous model transformations in order to assess its usefulness and ease of use by conducting an empirical case study with 57 users in Section 7.1.

[1] We provide a detailed comparison of our approach and the approach by Sun et al. in Section 4.1.7.

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

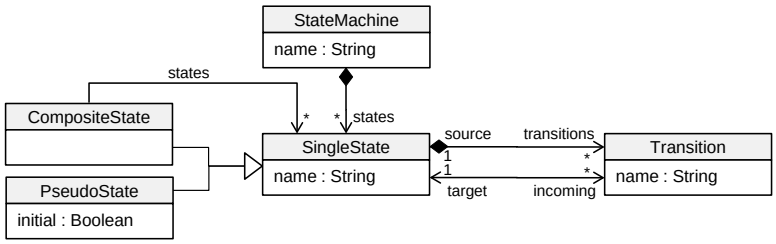
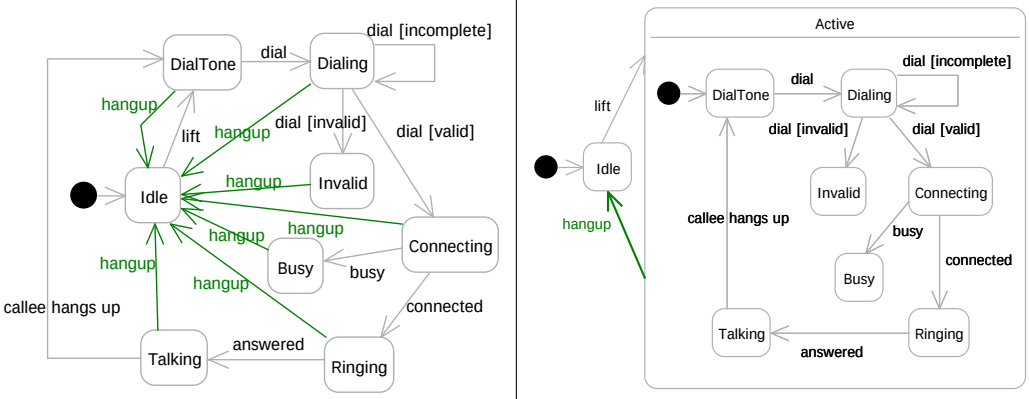


Figure 4.1: Metamodel for State Machine



(a) Initial Phone State Machine

(b) Refactored Phone State Machine

Figure 4.2: Refactoring *Introduce Composite State* [SPLTJ01]

4.1.1 Running Example

For illustrating the functionality of EMO, we make use of a refactoring for UML state machines. Therefore, we first introduce the metamodel of the simplified state machine modeling language in Figure 4.1. This metamodel contains the class StateMachine acting as a container for arbitrarily many instances of SingleState through the containment reference states. Such instances of SingleState may further contain instances of Transition through the reference transitions. A transition refers to its connected states through the references source, which is actually the opposite reference of transitions and target. The reference target also has an opposite reference, which is named incoming. Thereby, states "know" their incoming transitions. Besides usual states, the metamodel also contains the class PseudoState, for expressing initial and end states, as well as the class CompositeState grouping arbitrarily many other states.

The refactoring serving as a running example is called *Introduce Composite State*. We illustrate this refactoring by applying it to a concrete example that represents the states and transitions of a phone. The initial state machine and the refactored state machine are depicted in Figure 4.2. Please note that this refactoring as well as the example is taken from Sunyé et al. [SPLTJ01].

The initial phone state machine shown in Figure 4.2a contains several states such as Idle, DialTone, and Dialing. Please note that whenever a hangup event occurs, the phone switches

back to state Idle. The multitude of similar transitions, which are pointing to the state Idle and which are triggered by the same event, suggests the application of the refactoring *Introduce Composite State*. This refactoring introduces a composite state and folds all hangup transitions into one single transition as depicted in Figure 4.2b. More precisely, the refactoring consists of the following atomic operations:

1. A composite state named Active is created.
2. All states having the outgoing transition hangup are moved into the new composite state Active.
3. The outgoing hangup transitions of these states are folded into one single transition, which is outgoing from the composite state Active.
4. The target of the transition lift is changed to the state Active.
5. A new initial state having a transition to DialTone is created in Active.

Although the specification of such a refactoring is possible by using general-purpose programming languages, this task would require programming skills and deep knowledge of the underlying modeling framework and the modeling language's metamodel. When developing the *Introduce Composite State* refactoring in Java, the solution comprises nearly 100 lines of code for implementing only the pure refactoring logic, not counting an implementation of the refactoring's preconditions and the code necessary for realizing a user interface for applying it.

Another alternative to specify such composite operations is to use dedicated model transformation languages. This enables the development of composite operations, for instance, by developing declarative transformation rules, which is more concise in comparison to an implementation using general-purpose programming languages. However, as already stressed, besides requiring experiences in using such model transformation technologies, current approaches force users to specify the transformation rules using the abstract syntax of the modeling language, which might quickly become challenging and complex for untrained users. Furthermore, model transformation approaches are rarely integrated in current modeling environments. Thus, tool adapters are required to enable calling the transformation from within the modeling environment, which again requires dedicated knowledge for implementing such adapters.

Modelers, as the potential users of the our approach, are familiar with the notation, semantics, and pragmatics of the modeling languages they use in daily activities. They are, however, not experts in programming languages, transformation techniques, or APIs. Therefore, a novel approach is required to enable the specification of composite operations without posing these prerequisites.

4.1.2 EMO at a Glance

Composite operations may be described by a set of atomic operations, namely, *create*, *update*, *delete*, and *move*, which are applied to a model that adheres to certain preconditions [ZLG05]. A straightforward way to obtain these atomic operations from a user demonstration is to record each user interaction within the modeling environment as proposed for programming languages in [RL08]. However, this would demand an intervention in the modeling environment, and due to the multitude of modeling environments, we refrain from this possibility according to the

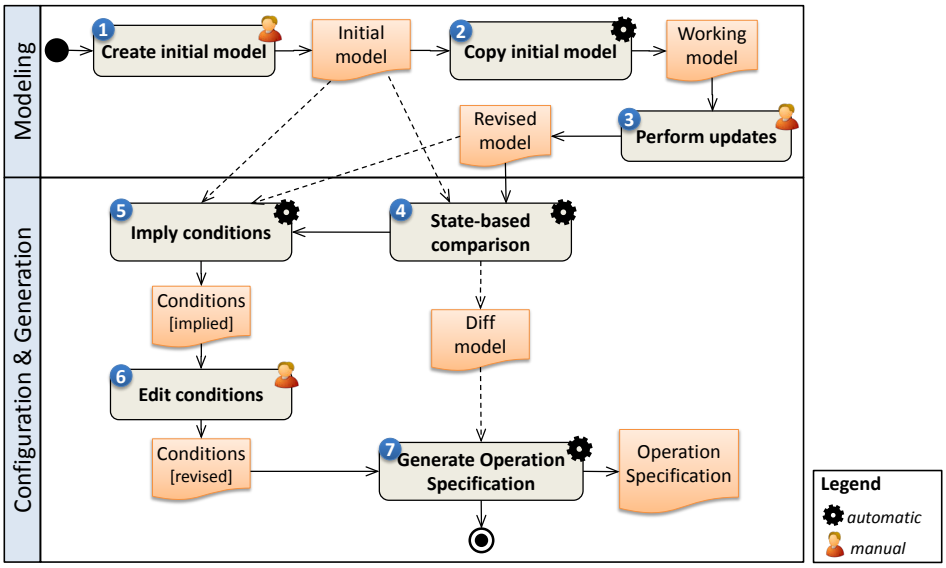


Figure 4.3: Process of Endogenous Model Transformation By Demonstration

design principles of AMOR (cf. Section 3.3). Instead, we apply a state-based model comparison to determine the demonstrated atomic operations. This allows the use of any editor without depending on editor-specific operation recording. To overcome the imprecision of heuristic state-based model comparison approaches, a unique ID is automatically assigned to each model element before the user demonstrates the atomic operations. Moreover, EMO is designed in such a way to be independent from any specific modeling language, as long as it is based on Ecore or the metamodel may be mapped to a corresponding metamodel expressed in Ecore. Therefore, we propose a two-phase specification process as shown in Figure 4.3. In the following, we discuss this two-phase specification process step by step.

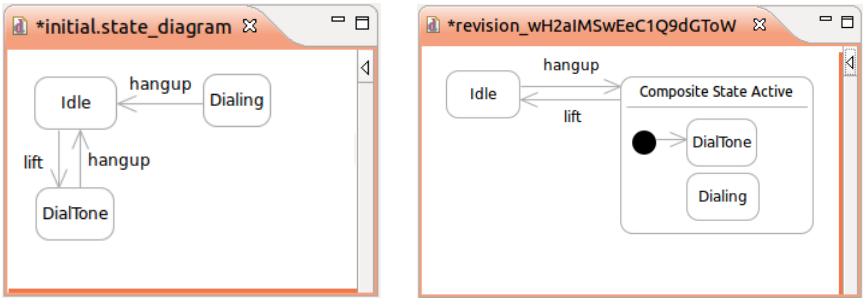
Phase 1: Modeling. In a first step, the user creates the initial model in a familiar modeling environment. This initial model contains all model elements that are required in order to apply the composite operation. In a second step, each element of the initial model is annotated automatically with an ID, and a so-called working model (i.e., a copy of the initial model for demonstrating the composite operation by applying its atomic operations) is created. In the third step, the user performs the complete composite operation on the working model, again in a familiar modeling environment by applying all necessary atomic operations. The output of this step is the revised model, which is together with the initial model the input for the second phase of the operation specification process.

Phase 2: Configuration & Generation. Due to the unique IDs, which preserve the relationship among model elements in the initial model and their corresponding model elements in the revised model, the atomic operations of the composite operation may be obtained precisely in step 4 by using a state-based model comparison. The results are saved in the diff model. Sub-

sequently, an initial version of *pre-* and *postconditions* of the composite operation is inferred in step 5 by analyzing the initial model and the revised model, respectively. The automatically generated conditions from the example might not always entirely express the intended pre- and postconditions of the composite operation. They only act as a basis for accelerating the operation specification process and may be refined by the user in step 6. In particular, parts of the conditions may be activated, deactivated, or modified within a dedicated environment. If needed, additional conditions may be added. After the configuration of the conditions, the operation specification is generated in step 7, which is a model-based representation of the composite operation consisting of the diff model and the revised pre- and postconditions, as well as the initial and revised example model. Thus, this model contains all necessary information for its further usage such as applying the operation to arbitrary models (cf. Section 4.1.5).

4.1.3 EMO in Action

In the previous section, we illustrated the operation specification process from a generic point of view. In the following, we show how the refactoring *Introduce Composite State* from Section 4.1.1 is specified using EMO from the users' point of view. Users are supported during the specification process by EMO's user interface of which some extracts are depicted in Figure 4.5. To view the complete user interface for developing composite operations, we kindly refer to the EMO project website[3] containing several screencasts and further information regarding the implementation.

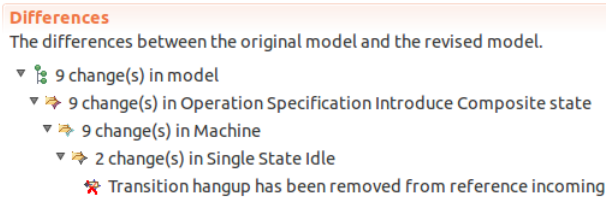


(a) Initial Model (b) Revised Model

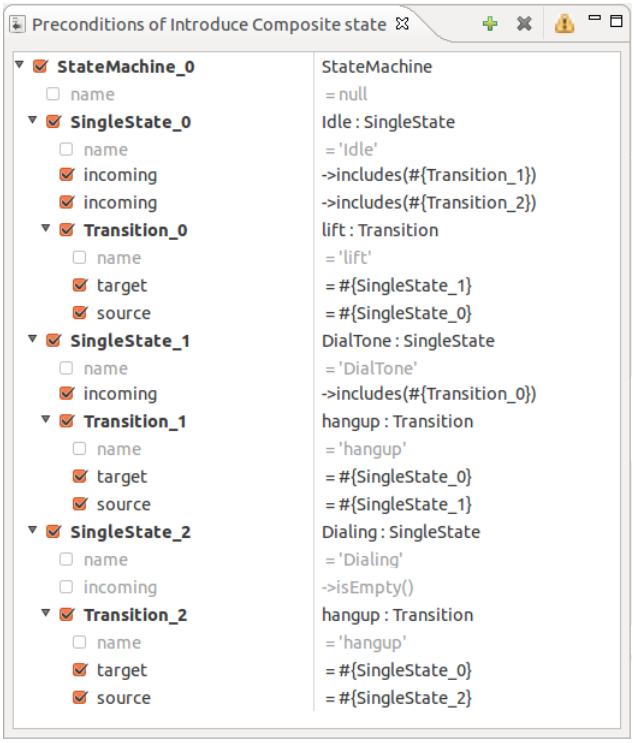
Figure 4.4: Example Models for Specifying *Introduce Composite State*

Step 1: Create initial model. The user starts with modeling the initial example model. For this task, the user may use any editor, such as GMF[4]-based graphical editors, EMF's tree-based editor, or even a text editor for directly modifying the model's XMI serialization, as EMO is independent of editor-specific operation tracking and solely relies on state-based model comparison. In this step, every model element has to be introduced that is *necessary and essential*

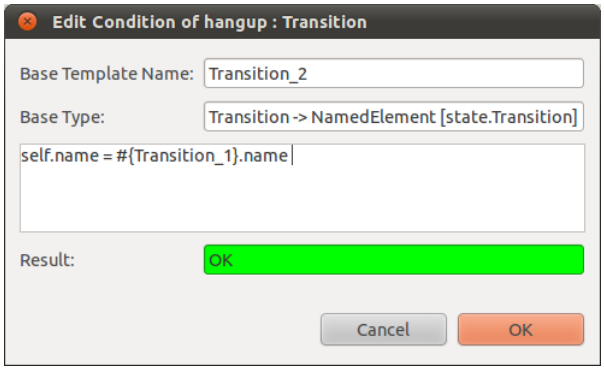
[3] http://www.modelversioning.org/emf-modeling-operations
[4] http://www.eclipse.org/modeling/gmf



(a) Differences



(b) Derived Preconditions



(c) Edit Preconditions

Figure 4.5: Screenshots of the User Interface of EMO

to demonstrate the composite operation. It is not necessary to create every state of the diagram shown in Figure 4.2a. Therefore, in the initial model, only those states are created that are essentially required for the refactoring and that will be modified differently later. Ultimately, the initial model consists of three states (cf. Section 4.4a). First, the initial model contains the state Idle, which will remain outside the composite state introduced in the course of the refactoring. Second, it comprises the state DialTone, which will be moved to the newly added composite state acting as first state, and, finally, the state Dialing, which will only be moved to the composite state losing its transition to Idle. There is no need to model, for instance, the state Connecting shown in Figure 4.2a as it is equally modified as Dialing. For these equally handled states, EMO provides techniques to define *iterations* in the configuration phase that we discuss later.

Step 2: Copy initial model. When the user confirms the initial model, the automatic copy process is initiated, which first adds a unique ID to every model element of the initial model before the working copy is finally created.

Step 3: Perform updates. After the ID-annotated working copy is created, it is opened in the user-selected editor ready to be modified for demonstrating the composite operation. The user applies each operation of the composite operation to this copy. In our example, the user has to add a composite state named Active, move the single states DialTone and Dialing into it, introduce a new initial state in Active, connect it with DialTone and change or remove the other transitions. The final revised model is depicted in Figure 4.4b.

Step 4: State-based comparison. In this step, the state-based model comparison between the initial model and the revised model is executed to identify automatically the previously demonstrated atomic operations. Internally, the comparison is realized by an extension of EMF Compare. Actually, the same model comparison component is used for this task that is also applied for obtaining atomic operations for model versioning purposes as presented in Chapter 5. When the comparison is completed, the detected atomic operations are saved in terms of a *diff model*, which is depicted in Figure 4.5a. For a precise specification of the composite operation, it is important that the user performs only those operations that directly represent the composite operation.

Step 5: Imply conditions. Next, EMO automatically derives the preconditions from the initial model and the postconditions from the revised model. The generation process works similarly for the pre- as well as for the postconditions: for each model element in the respective model, a so-called *template* is created. A template describes the role a model element plays in the specific composite operation. For each template, *conditions* are generated, which describe the required characteristics of a model element to be a valid match for a template. Thereby, for each feature value of the model element in the example model, an according condition is generated. For our example refactoring, this generation process creates the preconditions depicted in Figure 4.5b. In particular, this figure shows the template StateMachine_0 representing the root container of the initial model. Furthermore, it contains templates representing the three states Idle, DialTone, and Dialing and their respective preconditions. These templates have a symbolic name (e.g.,

SingleState_1), and are arranged in a tree hierarchy to indicate their containment relationships, which reflects the containment hierarchy of the corresponding example model. For expressing the condition bodies, OCL is used. However, we extended OCL in order to refer from within a condition body to other templates in order to express generically a reference to other model elements or their feature values; therefore, a dedicated syntax is introduced. For instance, the expression `incoming->includes(#{Transition_1})` in the template SingleState_0 indicates that its feature incoming must include a model element that fulfills the conditions of the template Transition_1. The scope of a template is either the initial model or the revised model. Nevertheless, it is still possible to access the template of the opposite model in the conditions using the prefixes `initial:` and `revised:` in template names, respectively. We discuss templates, conditions, and how they are evaluated in more detail in Section 4.1.4.

Step 6: Edit conditions. The automatically generated conditions might not always perfectly reflect the intended pre- and postconditions of the composite operation. They only act as a seed for accelerating the operation specification process and may be refined manually in this step. EMO allows to adapt the generated conditions in three different ways.

First, the user may *relax* or *enforce* conditions. This is simply done by activating or deactivating the check boxes beside the respective templates or conditions. If a template is relaxed, all contained conditions are deactivated. By default, conditions constraining *String* or *Boolean* features and *null*-values are deactivated (cf. Figure 4.5b), because in our experience, they are not relevant in most of the cases. Due to this default configuration, we do not have to relax any further conditions in order to reflect the true conditions of the refactoring. However, we have to enforce and modify one condition as discussed later.

Second, the user may *modify* conditions by directly editing them. For our example, it is necessary to specify that a state, which is moved into the composite state must contain an outgoing transition having the same name as the transition to be folded (in our example hangup). The condition ensuring that every state that is moved to the composite state must have a transition has already been generated: `SingleState_2.outgoing->includes(#{Transition_3})`. However, the condition restricting the transition's names to be equal has to be reactivated and modified. In particular, we have to change a condition in the template Transition_2, which is contained by the template SingleState_2 (representing the state Dialing). For this transition template, we modify the condition constraining the name feature as depicted in Figure 4.5c. This condition ensures that the transition must have the same name as the transition (represented by template Transition_1) that will be moved to the composite state and, therefore, acts as outgoing transition for all states (represented by the template SingleState_2) that are moved into the composite state. As depicted in the screenshot in Figure 4.5c, the user is assisted when modifying conditions by immediately checking the condition against the initial model or revised model, if the edited condition is a postcondition, to indicate the correctness of the condition. Thereby, the user gets immediate feedback whether the condition is syntactically correct, but also whether at least the example model fulfills the modified condition. If the condition is not fulfilled by the example model, it is very likely that the user made a mistake and specified a semantically incorrect condition. Furthermore, users are assisted while editing conditions by context-sensitive code completion.

Finally, users may adapt the composite operation specification by *augmentation*. Thereby, users may introduce custom conditions, define iterations, and annotate necessary user input for setting parameters of the composite operations. In our example, the user has to introduce one iteration for the template SingleState_2. This iteration specifies that all atomic operations that have been applied to the model element represented by this template have to be repeated *for all* its matching model elements when applied to an arbitrary model. The reason why we attach iterations to templates and not directly to the to-be-repeated operations is that we feel that attaching them to templates is more in tune with the general idea of the by-example concept; users are more familiar with the example models they provide than the automatically derived atomic operations. The impact of iterations on the execution of composite operations is elaborated in more detail in Section 4.1.5. Besides the iteration, the user also has to introduce a user input annotation for the name feature of the template CompositeState_0 to indicate a value, which has to be set by the user of the refactoring. Obviously, iterations may only be specified for templates from the initial model and user input for features of templates from the revised model.

In the course of applying a composite operation, certain values in the revised model often have to be computed from specific values in the initial model. Therefore, users may modify or add *postconditions*. Thus, postconditions are not only used to ensure the correct revised model, they may also yield value computations. Although not necessary for our running example, consider for instance the composite operation called *Encapsulate Field* [FBB+99] for UML class diagrams. This composite operation generates one method for getting and one method for setting the value of a public attribute and, finally, turns the visibility of the attribute to private. For this composite operation, the method name, the return type of the getter method, and the parameter name and type of the setter method have to be computed from the source model. For example, the postcondition `self.name = 'get' + #{Attribute}.name.firstToUpper()` can be used to compute the correct name of the getter method for an attribute.

Step 7: Generate Operation Specification. After the user finished editing the conditions and augmenting the operation specification, the Operation Specification is generated. This model-based representation of the composite operation contains all necessary information for its further usage, such as applying the operation to arbitrary models (cf. Section 4.1.5), detecting applications of the operation a posteriori (cf. Chapter 5), and revealing conflicts coming from the preconditions of the composite operation (cf. Chapter 6). Operation specifications conform to the metamodel depicted in Figure 4.6. The class CompositeOperationSpecification contains general information, such as the composite operation's name, the modeling language, for which it can be used, as well as the initial and revised model, the pre- and postconditions, the iterations, and the DiffModel comprising the atomic operations. In particular, the initial and the revised model are kept in the attributes initialModel and revisedModel of the class CompositeOperationSpecification and the pre- and postconditions are each saved in terms of instances of ConditionModel via the references preconditions and postconditions, respectively. Each condition model contains one root Template representing the root model element of the initial or revised model. This root template contains a number of sub-templates, which may have sub-templates corresponding to the containment hierarchy of the model elements in the initial or revised model. The specific model element that is represented by the respective template is ref-

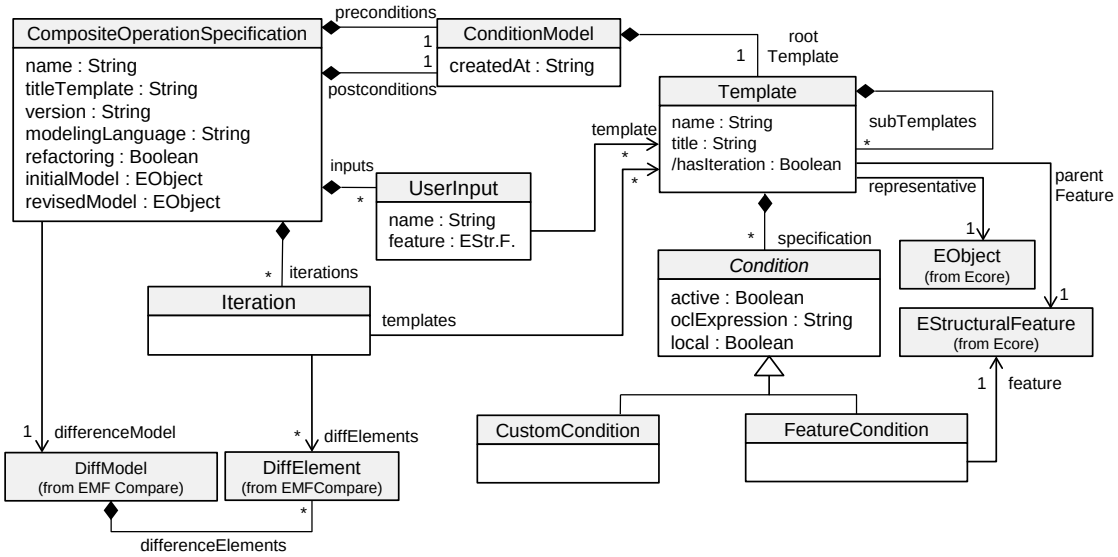


Figure 4.6: Operation Specification Metamodel

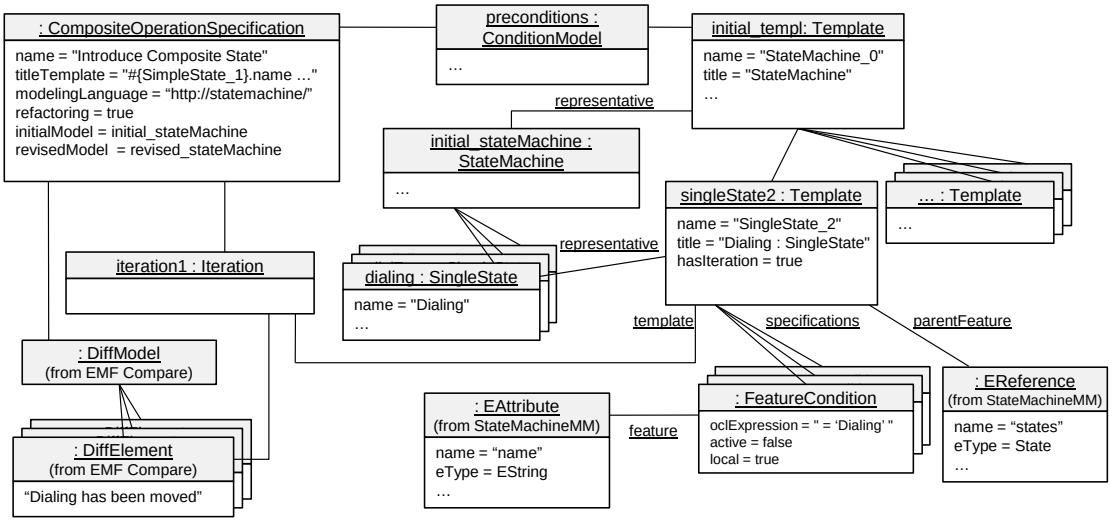


Figure 4.7: Excerpt of the Operation Specification for the Running Example

erenced through the reference representative. Furthermore, instances of Template are specified by a list of custom conditions and feature conditions. Instances of FeatureCondition constrain the value of a specific feature and are generated automatically in step 5. We will discuss the concepts behind templates and how they are bound to model elements in more detail in Section 4.1.4.

Figure 4.7 illustrates an excerpt of the object diagram representing the operation specification for the previously described example refactoring. In particular, this figure highlights some

objects, such as the introduced iteration, the template hierarchy and its references to the concrete model elements, as well as an instance of a FeatureCondition for the feature name of template SingleState_2. All of these components have their counterpart in the user interface to be modified easily by the user.

4.1.4 Condition Models, Templates, and Template Bindings

Before we show how composite operation specifications can be applied to arbitrary models in the next section, we first discuss condition models and templates, and how they are matched with models to obtain valid template bindings.

Condition Models. As depicted in the metamodel of operation specifications in Figure 4.6, an OperationSpecification holds one ConditionModel for the operation's preconditions and one for the operation's postconditions. Such a condition model contains a set of templates by which it generically describes the characteristics a model should satisfy. A condition model as a whole successfully matches with a model part if each of its templates has a matching model element within a model.

Templates. As already mentioned, the purpose of templates is to describe the required characteristics a model element must have in order to be a valid match, and which relationships to other model elements within the described model must exist. These required characteristics and relationships are defined in terms of conditions contained by the respective template. As each template is generated from an existing model element in the example model in step 5 of the operation specification process, each template preserves the relationship to the original model element it has been generated from, through the reference representative (cf. Figure 4.6). According to the containment hierarchy of the example model, templates are organized in a tree structure having one root template (representing the example model's root model element), which has sub-templates (representing the root element's children), which may have sub-templates. For explicating this containment structure, each template, except for the root template, refers to the structural feature of the respective modeling language's metamodel, through which the represented model element is contained. For example, instances of the metaclass SingleState are contained by instances of StateMachine through the structural feature states (cf. state machine metamodel in Figure 4.1). Consequently, the template SingleState_2, which indeed represents an instance of such a SingleState, refers to this containment reference by the reference parentFeature (cf. object diagram in Figure 4.7). Thereby, templates that contain sub templates have further *implicit conditions* regarding their containments, besides their explicitly contained conditions. For instance, the template StateMachine_0, has, among others, the implicit containment condition `self.states->includes(#{SingleState_2})` coming from its contained template SingleState_2.

Conditions. Templates are further defined by a set of explicitly contained conditions that must be fulfilled by a matching model element. As already mentioned, conditions are expressed using OCL expressions, which are saved in the condition's attribute oclExpression (cf. Figure 4.6).

Thus, the full expressive power of OCL may be used for constraining matching model elements. Condition models may contain two types of conditions: instances of FeatureCondition, which constrain the value of a certain feature indicated by the reference feature in the metamodel in Figure 4.6, and instances of CustomCondition, which are not explicitly tied to a specific feature. As for each feature of a template's represented model element, a dedicated condition is generated. Only instances of FeatureCondition are automatically created in step 5 of the operation specification process. The explicit link to the feature that is constrained by an instance of FeatureCondition allows for easier processing and reasoning. For instance, a FeatureCondition for the feature name having the oclExpression = `"Dialing"` is rewritten to the OCL expression `self.name = "Dialing"`, whereas `self` is bound to the model element to be evaluated. If this condition is not fulfilled by a model element, we easily may conclude, without having to analyze the contents of the OCL expression in detail, that the model element's value at the name feature causes the condition to fail. This is not as easily possible for the equivalent CustomCondition, which would have the OCL expression `name = "Dialing"`, because we would have to interpret this OCL expression in detail to find out the specific feature value that causes the condition to be invalid. As mentioned above, we extended OCL to allow for referring to other templates and its values. Therefore, we introduced a dedicated syntax: by using `#{<template-name>}` in a condition, users may refer to the model elements that are bound to the referenced template. For evaluating OCL expressions that contain such template references, occurrences of these references are replaced with expressions navigating to the model element that is currently bound to the referenced template. For instance, the OCL expression = `#{Transition_1}.name` in a FeatureCondition constraining the name feature is replaced with the following OCL expression[5], whereas `1` is the index of the state and `0` is the index of the transition that is currently bound to the template Transition_1:

```
self.name = self.eContainer().eContainer().states.select(1).
            transitions.select(0).name
```

Such replacements are computed by first finding the closest common parent container of both currently bound model elements in the model and then deriving the direct navigation from the source model element to the target model element. To enable a more efficient processing and reasoning, conditions additionally save whether they refer to other templates or whether they are *local*; that is, no reference to other templates are involved (cf. attribute local in the metamodel depicted in Figure 4.6).

Template Bindings. When matching model elements to condition models, the mappings between templates and their matching model elements are described by so-called *bindings*. These bindings are realized by a weaving model conforming to the metamodel depicted in Figure 4.8. As a condition model contains arbitrarily many templates, a ConditionModelBinding contains *for each* of a condition model's template exactly *one* TemplateBinding. Such a TemplateBinding connects *one* template with *one* model element that is bound to the respective template. In other words, one instance of a ConditionModelBinding constitutes an intrinsically valid set of

[5]Please note that we omitted required type castings (`oclAsType()`) and collection castings (`asSequence()`) in the OCL expression for the sake of readability.

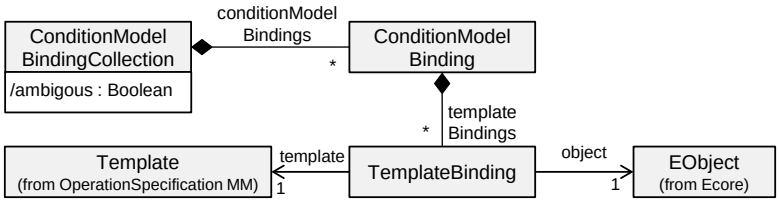


Figure 4.8: Template Binding Metamodel

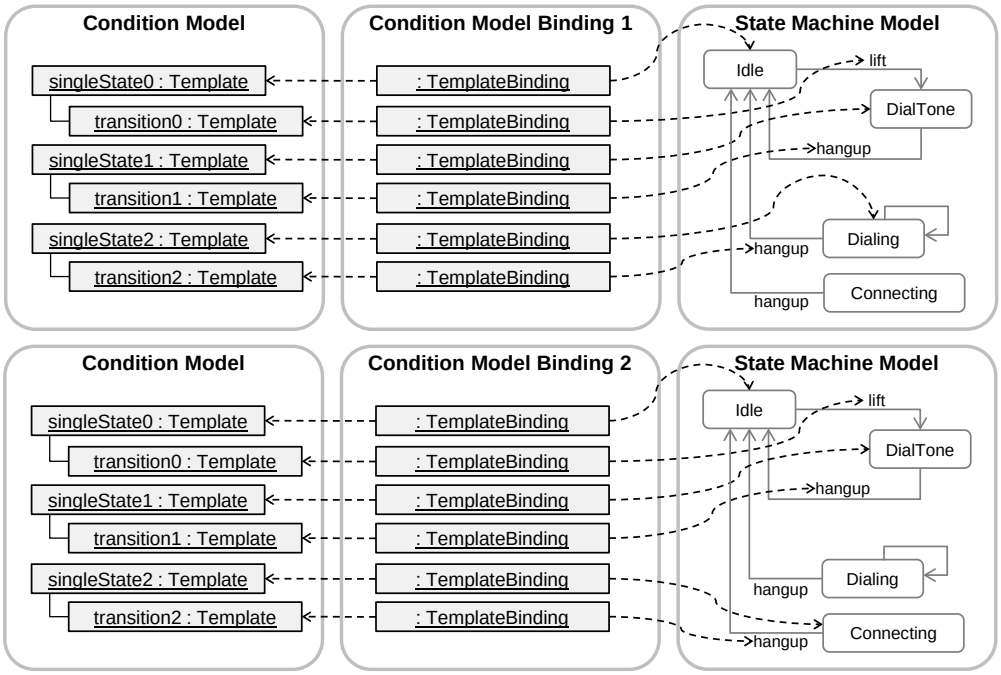


Figure 4.9: Example for Condition Model Bindings

distinct one-to-one relationships between templates and model elements, whereas each template of the condition model is bound to exactly one model element and one model element is bound only once. Because of multiple matches of a condition model in a model or because of iterations attached to templates, one template may also be bound to multiple model elements. This is realized by having a ConditionModelBindingCollection, which may contain multiple intrinsically valid and unique ConditionModelBindings. If there are multiple ConditionModelBindings, they may overlap regarding a subset of their template bindings. Consider, for instance, the example depicted in Figure 4.9. In this example, we have the condition model expressing the preconditions of our example refactoring on the left. On the right, there is an excerpt of our running example's state machine. Please note that the upper condition model as well as state machine are the same as the lower ones; we graphically split them for the sake of readability. Between the condition model and the state machine, there are two condition model bindings, Condition Model Binding 1 and Condition Model Binding 2. Both are intrinsically valid and unique, how-

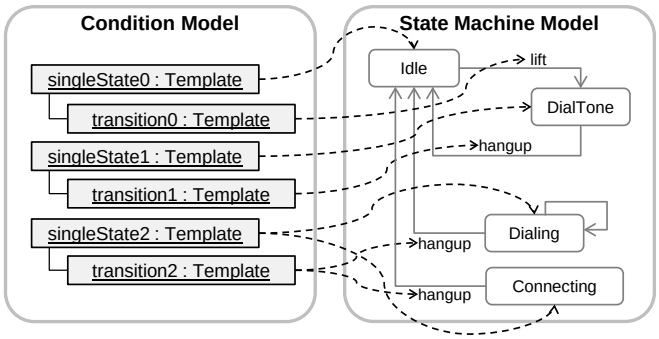


Figure 4.10: Condition Model Bindings Combined Within One Binding Collection

ever, Condition Model Binding 1 binds the template objects singleState2 and transition2 to other model elements in the state machine than Condition Model Binding 1 does. The rest of the bindings are overlapping. If now both of the condition model bindings are combined within one ConditionModelBindingCollection, we obtain the combined template binding depicted in Figure 4.10. In this combined binding, the template objects singleState2 and transition2 are now multiply bound.

As already mentioned, it is only allowed to bind multiple model elements to one template, if there is an iteration attached to the respective template; it is, however, also valid if the multiple binding is only due to an iteration attached to its direct or indirect *parent template*. Consequently, the ConditionModelBindingCollection in Figure 4.10 is valid, if the template object singleState2 has an iteration attached to it. Note that there is no need for an iteration at transition2 despite there are multiple transitions bound to it. This is because there is indeed an iteration at its container template singleState2 and *in the context* of each state bound to singleState2, only one *single* transition has been bound; more than one bound transition within the context of one state bound to the template singleState2 would be disallowed. If this would be intended, we would have to attach another iteration to the template transition2.

If, on the other hand, a ConditionModelBindingCollection comprises multiple bindings to a template that does not directly or indirectly have an iteration attached to it, the ConditionModelBindingCollection is *ambiguous* as it is not clear which model elements should be transformed without repeatedly applying atomic operations demonstrated in the specification process. Therefore, the user has to remove one of the ambiguous bindings, before the composite operation may be executed. Assuming that we have no iterations configured in the example depicted in Figure 4.9, the user would have to remove either the binding between singleState2 and state Dialing or the binding between singleState2 and state Connecting. By doing so, the entire condition model binding (either Condition Model Binding 1 or Condition Model Binding 2) is discarded from the ConditionModelBindingCollection that originally held both bindings. Thereby, not only the ambiguous binding regarding the state, but also the ambiguous binding of its contained transition, is removed. Consequently, as long as there is still one binding left in the collection, the user always ends up having at least one valid and complete binding.

Finding valid template bindings. Finding valid bindings in a model for a given condition model is, basically, graph-based pattern matching [Gal05] or, more precisely, the problem of finding a *subgraph isomorphism* [Ull76], whereas the condition model corresponds to the *pattern graph* or graph query and the model corresponds to the *data graph*. According to the categorization of graph-based pattern matching problems by Gallagher [Gal05], the problem of finding valid bindings for a condition model deals with *exact matching* to find all *optimal solutions* because applying inexact matching to only achieve approximate solutions is insufficient for our use case. Exact matching for obtaining all optimal solutions is an NP-complete problem [Gal05, Ull76]. One of the earliest approaches to exact pattern matching is the subgraph isomorphism algorithm proposed by Ullmann [Ull76], which uses a depth-first tree-search algorithm. Thereby, a search tree is built, whereas each tree-hierarchy level maps to a node of the pattern graph and the tree nodes are constituted by nodes in the data graph. The algorithm traverses through that tree depth-first and checks whether all conditions down the way to the tree leaves are fulfilled. If the algorithm finds an invalid node or transition, it discards the whole remaining branch of the tree and goes on. Ultimately, the remaining tree contains all exact matches; each match is a path in the tree from the root element to its leaves.

We use a similar approach to find valid bindings. However, we do not enumerate all potential binding combinations in a tree in advance; we rather employ a recursive backtracking algorithm, which dynamically selects the next model elements to be evaluated. The input for the matching algorithm is the model to be matched and the condition model. Additionally, the user has to provide an initial binding for at least one template to one model element. Basically, the algorithm iterates depth-first through the condition model template by template. For each template, the algorithm checks whether a binding for that template already exists. If the template is not bound yet, it selects all heretofore unbound model elements that are a potential match and evaluates them with the current template's conditions. At this point, only local conditions or conditions that refer to already bound templates can be evaluated immediately. For the remaining conditions, which refer to currently unbound templates, the algorithm first again selects candidate model elements for all unbound templates these conditions refer to. As a result, the algorithm obtains a set of base candidates for the current base template, and, for each of these base candidates, a set of referenced candidates for each referenced template. To explore all potential branches, the algorithm builds the permutation of all unique element-to-template binding combinations. Each base candidate is now evaluated against the remaining conditions with each of its permutation of referenced candidate bindings. If more than one valid base candidate in the context of the referenced candidate bindings remains, the algorithm proceeds with accepting only one of these base candidates and referenced bindings and starts a recursion for each remaining combination of base candidates and its referenced bindings. Thereby, each potential branch is evaluated by one recursion. If a recursion reaches a point in which no valid model element can be found for the next template or the model element bound to the current template is invalid, the branch is discarded. Otherwise, the recursion stops as soon as a complete condition model binding has been built finally. Thereby, one recursion only builds one unique and intrinsically valid ConditonsModelBinding, which only contains one-to-one bindings. Ultimately, all detected valid ConditonsModelBinding are put into a ConditionModelBindingCollection, which represents all valid matches, being ambiguous or not.

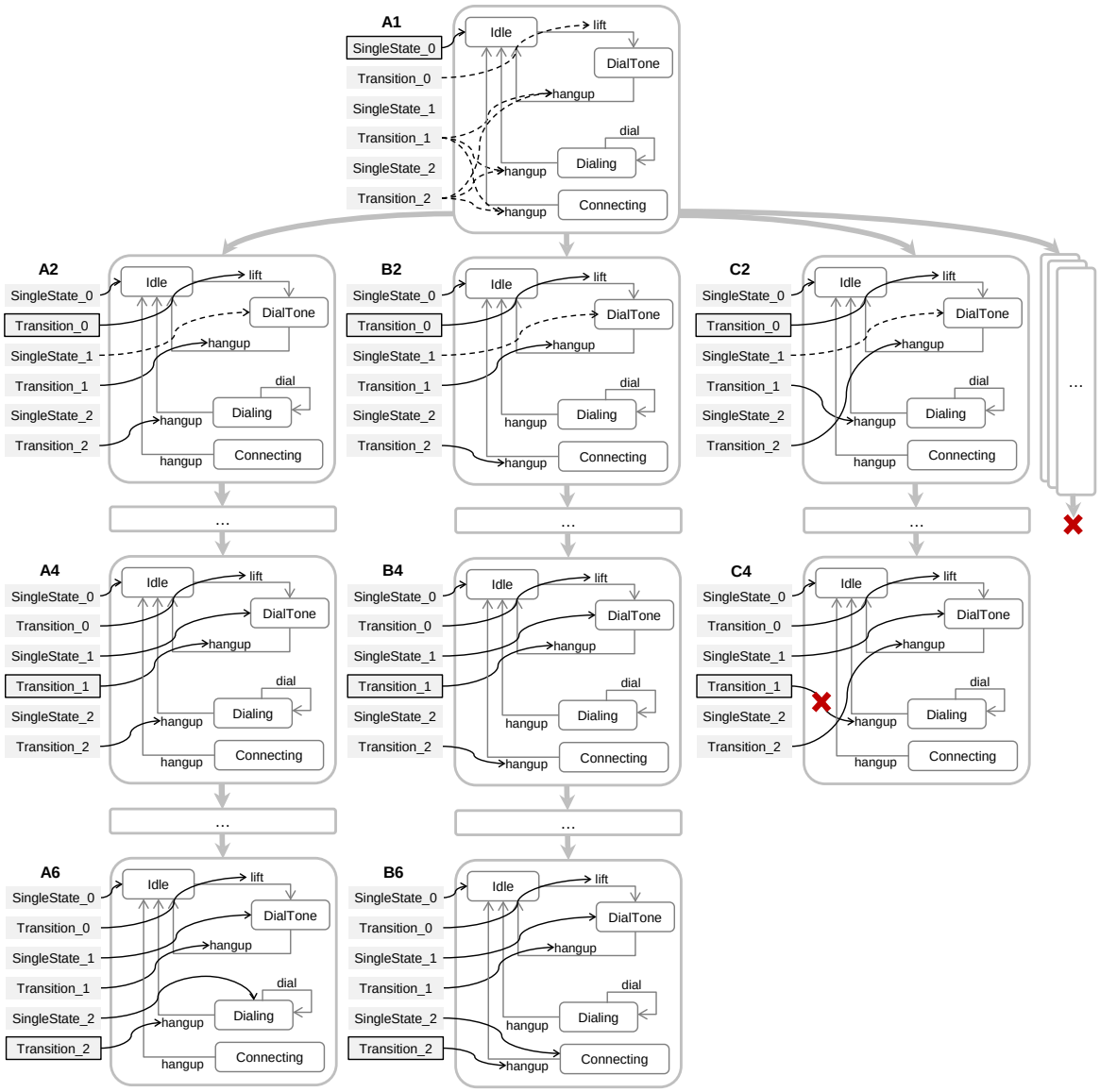


Figure 4.11: Example for the Template Matching Algorithm

Example. For making our algorithm for finding valid condition model bindings more clear, we go through a small example, which is depicted in Figure 4.11. In particular, we show how the precondition model of our running example (cf. Figure 4.5b) is matched with an excerpt of the phone state machine model (cf. Figure 4.11). Assume the user specified an initial binding that maps the state Idle to the template SingleState_0. The algorithm traverses depth-first through the condition model. Thus, the first template to be considered is SingleState_0. As this template has already been bound in the initial binding, we may directly proceed with checking its conditions. First, we consider the implicit condition `transitions->includes(`

`#{Transition_0})` coming from SingleState_0's contained template Transition_0, which is heretofore not bound to a model element. Consequently, we first have to obtain candidates for Transition_0 before we may evaluate the implicit condition. As already mentioned, templates explicitly refer to the feature through which they are contained by their parent element (cf. reference parentFeature of templates in the operation specification metamodel in Figure 4.6). For the template Transition_0, this is the reference called transitions in single states. Therefore, all model elements from `Idle.transitions` are retrieved to obtain the referenced candidates for Transition_0, which is just the transition lift in our example. Additionally, our current base template SingleState_0 contains two explicit conditions referring to other templates, which is on the one hand `incoming->includes(#{Transition_1})` and on the other hand `incoming->includes(#{Transition_2})`, whereas the referenced templates, namely Transition_1 and Transition_2, have not been bound yet. Thus, we first have to select candidates for these two templates before we may evaluate these two conditions of SingleState_0. As we have no explicit hints on suitable candidates for Transition_1 and Transition_2, we have to consider all heretofore unbound model elements having the type Transition in the state machine as candidates for both referenced templates. Therefore, all transitions in the state machine, except for lift, are now evaluated against the conditions of template SingleState_0. According to these conditions, only those transitions remain relevant that are *incoming* to the state Idle (i.e., all transitions named hangup). All other transitions are discarded as candidates for Transition_1 and Transition_2 (cf. A1 in Figure 4.11). To explore all potential branches arising from these candidates for these two templates, we now have to proceed with all *k-permutations of n*, whereas k is the number of templates (i.e., Transition_1 and Transition_2) and n is the number of model elements (i.e., the three transitions named hangup). This leads to six combinations. The first combination (cf. A2 in Figure 4.11) is further considered in the current branch named A and for each of the remaining combinations, a new recursion is started (branch B to F. For the sake of readability, in Figure 4.11, only three branches (A, B, C) are depicted.

The next template to consider is Transition_0, which has been already bound in all branches. Therefore, we directly proceed with evaluating its conditions. Fortunately, one of its conditions, namely `source = #{SingleState_0}`, refers to a template that is bound already and, as a result, may immediately be proved to be valid in all branches. The other condition, `target = #{SingleState_1}`, refers to the heretofore unbound template SingleState_1. Thus, we first have to find suitable candidates for the referenced template before we may evaluate this condition. The only state that fulfills this condition is DialTone, so we proceed with this candidate for SingleState_1 in each branch. The next template to be evaluated is the beforehand referenced template SingleState_1. This template contains one condition, that is, `incoming = #{Transition_0}`. As the referenced template has been bound already in all branches, we may directly evaluate it, which leads to accepting DialTone for SingleState_1 in all branches.

The next template to be considered is Transition_1, which also has been bound already. This template contains the conditions `source = #{SingleState_1}` and `target = #{SingleState_0}`. As we also have a binding for both referenced templates, we may evaluate these conditions right away. However, in branch C, the condition concerning the source is not fulfilled because the `source` of the transition is actually Dialing and the model element bound to SingleState_1 is DialTone (cf. C4 in Figure 4.11). Consequently, branch C is

discarded. The bindings in the branches A and B, however, fulfill both conditions. Thus, we proceed with these branches by evaluating the next template, SingleState_2. This heretofore unbound template comprises only the implicit condition regarding its contained template Transition_2, which has been bound already in both remaining branches. Therefore, we first have to select all remaining states as candidates and check the implicit condition for each candidate. In branch A, only the state Dialing fulfills this condition as it contains the transition bound to Transition_2. In branch B, the only state that fulfills this condition is, on the contrary, Connecting because, in this branch, a different transition is bound to Transition_2. Anyway, in both branches, we found a valid state.

Therefore, we may proceed with evaluating the last template Transition_2. As we already have bindings in all branches, we may directly evaluate all three conditions of this template. The first condition concerns the name of the transition (cf. Figure 4.5c); as both transitions are named equally to the transition bound to Transition_0, this condition is fulfilled in both branches. The same is true for the remaining two conditions concerning the transition's source and the target. Consequently, we end up having two valid ConditionModelBindings, which are depicted in A6 and B6 of Figure 4.9.

4.1.5 Execution of Operation Specifications

In this section, we show how operation specifications are executed to arbitrary models that fulfill the operation's precondition. When executing operation specifications, we first have to obtain a precondition model binding based on an initial binding specified by the user (cf. Section 4.1.4). Having a complete precondition model binding, we now aim to apply the same operations that have been demonstrated by the user when specifying the operations to the bound model elements.

Diff elements in operation specifications. The atomic operations that have been demonstrated during the specification process of a composite operation are saved in the operation specification in terms of a *diff model*. For obtaining such a diff model from the user-provided example models, we employ an extension to the state-based model comparison, which is realized by EMF Compare. For more information on obtaining atomic operations, we kindly refer to Chapter 5. In the context of executing operation specifications, it is sufficient to know that the obtained diff model contains *diff elements*, which precisely describe the applied atomic operation. In particular, such diff elements indicate the operation type (e.g., addition, deletion, update) and refer to the modified model elements and, where required, to the updated feature. Thus, such diff elements contain enough information to apply the described operation.

EMF Compare merge API. Fortunately, EMF Compare provides, besides its model comparison features, also a *merge API*, which is capable of applying detected *diff elements* to the compared models. For instance, if a model comparison detected the diff element "feature f of model element e has been updated from value v_1 to value v_2", we may apply the difference to the opposite model version (i.e., the concurrently modified version) so that the value v_1 in feature f is updated to v_2 in the corresponding model element in the opposite model of e. Thereby,

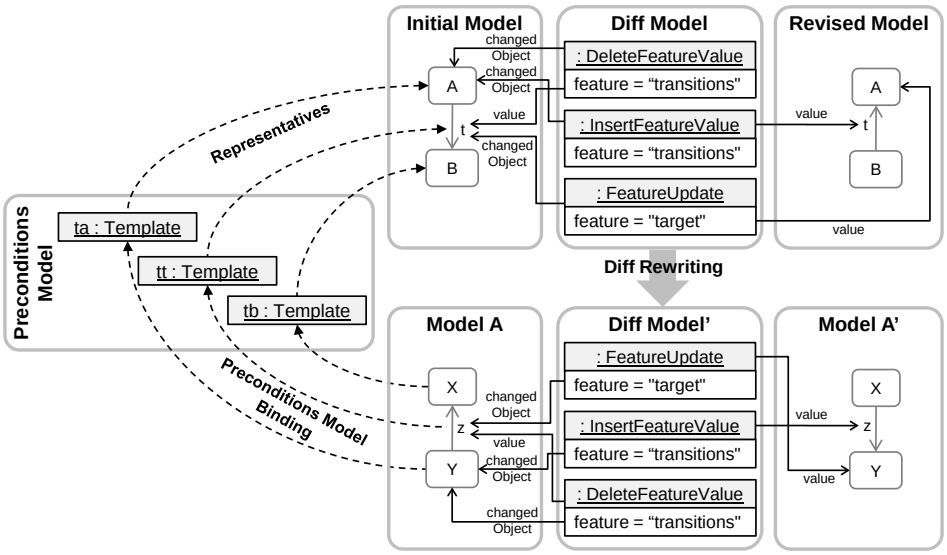


Figure 4.12: Example for Rewriting and Executing Diff Elements

EMF Compare allows to merge two models by applying all changes to a model that have been applied to the opposite model. We exploit this merge API to realize the execution of operation specifications.

Rewriting diff elements. EMF Compare, however, allows to apply diff elements only to the *compared models* directly and not to other models. Therefore, we first clone the operation specification's diff model and *rewrite* this diff model copy so that the references that originally refer to the model elements of the operation specification's initial model ultimately refer to the model elements to which the composite operation shall be applied.

This rewriting mechanism is illustrated by a small example in Figure 4.12. To keep the diff models small and clear, a new exemplary composite operation is used. More precisely, this composite operation changes the direction of an existing transition. Therefore, the initial model comprises two states, A and B, and one transition t. The precondition model accordingly contains three templates; one for each model element. In the revised model, the container state of transition t is changed from its original container A to the state B and the transition's target is changed to state A. Consequently, the diff model contains three diff elements, namely, a DeleteFeatureValue for detaching the transition from its original container state, an InsertFeatureValue for adding the detached transition to the new container state again, and a FeatureUpdate for changing the target of the transition. Please note that we use our own terminology for diff elements and our own metamodel for representing diff elements in this thesis as we feel that EMF Compare's diff model might be not as concise and clear to readers of this thesis. For a detailed discussion of diff models, we kindly refer to Chapter 5. This diff model refers to model elements in the initial and revised example model by the references changedObject and value for indicating the affected model elements (cf. Figure 4.12).

For applying this operation specification to an arbitrary model, named Model A in Figure 4.12, the respective model elements have to be bound first to the three templates in terms of a precondition model binding. The next step is to create a copy of the original diff model, called Diff Model', and rewrite it accordingly. In particular, the references to the initial model are changed so that they refer to the corresponding model elements in Model A. For instance, DeleteFeatureValue originally refers to state A through the reference changedObject. We know that state A is represented by the template ta, which is bound to state Y in Model A. Therefore, we may rewrite the reference target of changedObject from A in the operation specification's Initial Model to Y in Model A. The same mechanism applied to all remaining references going from diff elements to model element in the operation specification's Initial Model. For rewriting the reference targets going to the operation specification's Revised Model, it is required to make use of the *match model* between the Initial Model and the Revised Model. Please note that this match model is not depicted in Figure 4.12 for the sake of readability. It maps each model element in the Revised Model to its corresponding original model element in the Initial Model. Thus, when rewriting, for instance, the reference value going from InsertFeatureValue to the Revised Model's transition t, we may obtain its corresponding transition t in the Initial Model through the match model, get its representing template tt and, finally, recall the bound transition z in Model A. The same mechanism can be applied for the reference value of FeatureUpdate.

Having rewritten all diff elements in Diff Model', we may use EMF Compare's merge API to apply them to Model A. Although it is not required in our example, additions of model elements pose another challenge. Added model elements in the operation specification's Revised Model certainly have no corresponding model element in the Initial Model. Thus, we may not rewrite the references going from the diff element to the added model element in the Revised Model as easily. Therefore, we apply a two-phase diff rewriting and execution. First, we apply only additions by copying the added elements to the respective location in the model to which the composite operation is applied and keep the relationship between the originally added model element to the created copy in an intermediate trace model. Thereby, we have to apply the additions starting from top-level elements to bottom-level elements in terms of the containment hierarchy. Otherwise, we could not add a child to a parent that has not been created yet. Next, we have to rewrite the reference values of these added elements as they still might refer to model elements in the operation specification's Revised Model. Consider a scenario in which a new transition has been added; by only copying it to the model to which we aim to apply the composite operation (named again Model A hereafter), the transition would still refer to the target state of the operation specification's Revised Model. Therefore, after copying all added model elements, we have to walk through all of their feature values, check whether these are model elements in the operation specification's Revised Model, and, if so, change the value to the corresponding model element in Model A. For that, we have to query either the match model or the intermediate trace model containing the correspondences between added model elements in the Revised Model and their copies in Model A. Subsequently, we may process all other diff elements.

Handling iterations in the execution. To recall, iterations are attached to precondition templates to indicate that the diff elements affecting these templates (called *iterative templates* hereafter) shall be repeated for all model elements bound to such templates. Thereby, iterations have

two consequences concerning the composite operation execution: first, an iteration enables the multiple binding of model elements to the iterative template (cf. Section 4.1.4), and, second, all operations that have been applied to the corresponding model element during the demonstration are repeated for each bound model element. In the following, we discuss the latter consequence in more detail.

An unambiguous ConditionModelBindingCollection that contains more than one ConditionModelBinding entails that at least one template within the bound condition model is iterative; otherwise, the binding collection would be ambiguous. As a consequence, each unique and intrinsically valid ConditionModelBinding within the collection describes the context within which one iteration of the operation specification's diff model shall be performed. Thus, iterations are realized by creating and executing a rewritten copy of the operation specification's diff model *for each* unique ConditionModelBinding within one collection. However, we may not naively repeat all diff elements in each iteration; otherwise, we might, for instance, inadvertently add more than one model element to a container model element that is indeed not bound to an iterative template. Moreover, it is only possible to delete one model element or one feature value once and not repeatedly in each iteration. Therefore, we have to regard certain rules when copying the diff model for repeating the execution. In particular, only those diff elements are copied that refer, either by the reference changedObject or value, to a model element that is represented by an iterative and multiply bound template. One exception, however, are FeatureUpdates, which set the value of a single-valued feature; such diff elements are only copied if the reference changedObject, and not only the reference value, refers to a model element represented by an iterative and multiply bound template; otherwise, we would overwrite the same value in each iteration over and over again. Please note that the reference value might refer to a model element in the Revised Model so that we have to use the match model again to infer whether the Revised Model's element is represented by an iterative and multiply bound precondition template. Subsequently, these copied diff elements are each rewritten for one unique ConditionModelBinding. Thereby, we ensure that each diff element is tailored precisely to be executed within the correct context.

4.1.6 Considering More Sophisticated Composite Operations

In this section, we discuss advanced features of EMO in order to address more sophisticated composite operations. Please note that these features mainly concern the expressive power of operation specifications, rather than automating their specification.

Notation

Before we present the advanced features, we first introduce the notation used for depicting operation specifications. Therefore, in Figure 4.13, an example of an operation specification is shown. Directly below the <Operation Name>, there are two areas, namely *Intial Model* and *Revised Model*, illustrating the initial and the revised model in the concrete syntax, respectively. In particular, for presenting the advanced features, we use Ecore models and use the concrete syntax of UML class diagrams. Each model element is annotated with an Object ID in brackets (e.g., `[1]`) to indicate the mapping between the initial and revised models, as well as their

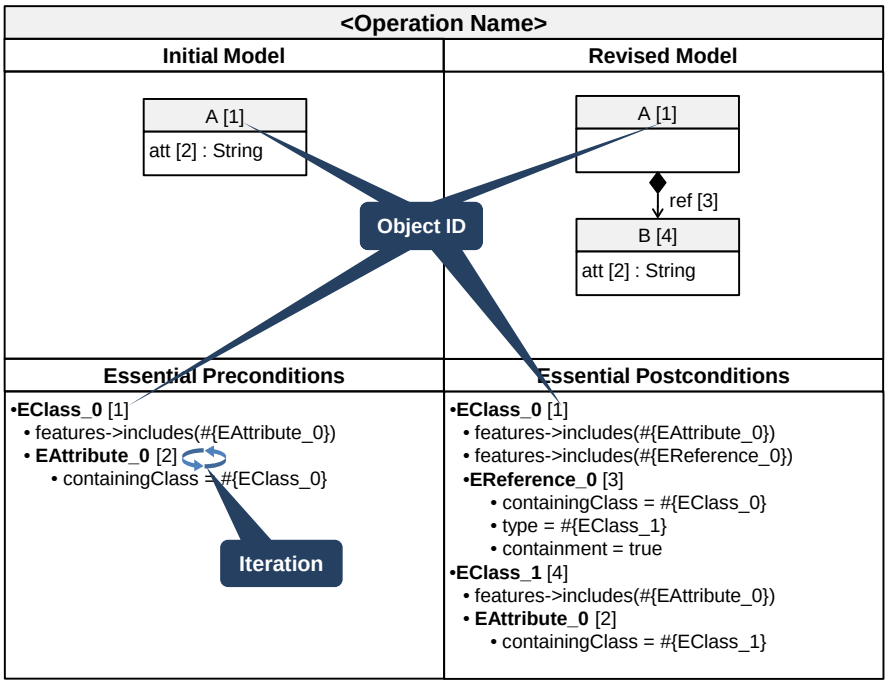


Figure 4.13: Notation for Illustrating Operation Specifications

corresponding templates in the condition models, which are depicted below in two dedicated boxes entitled Essential Preconditions and Essential Postconditions. In these condition models, template names are printed in **bold**. The condition models are organized in a hierarchical enumeration according to the containment hierarchy of the initial or revised model. If iterations are attached to precondition templates, this is indicated by the icon ⟲ next to the iterative template's name.

Introducing Copies

In the initial version of EMO, the supported diff element types that can be detected between the initial model and the revised model after the demonstrations were additions, deletions, moves, and updates (of reference and attribute values). However, in many operations, a model element should be copied, including its containments, instead of simply added. Therefore, we introduce the diff element type *copy*. EMF Compare, however, does not support detecting copies. Thus, users may convert detected additions into copies in the configuration phase by selecting the respective diff element representing the addition in the list and convert it into a copy, whereas the copy source has to be selected manually from the initial model. Obviously, only those model elements are allowed to be selected as copy source, if they have the same metamodel type and the same properties as the originally added model element. After choosing the copy source, there is an explicit reference from the *copy-typed* diff element to the copy source being a model element in the initial model; thus, also iterations attached to the specified copy source can now

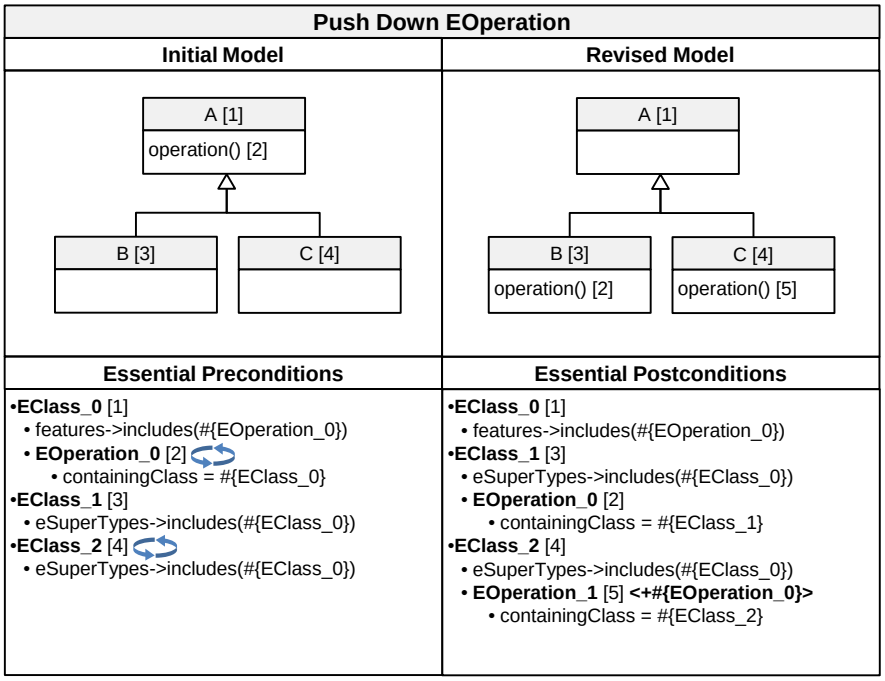


Figure 4.14: *Push Down EOperation* for Illustrating the Benefits of *Copy*

be supported. In particular, if the copy source element is represented by an *iterative* template, the execution engine repeats creating the copy for each bound model element. Having only usual diff elements representing plain additions, no such reference to the initial model's elements exists and, consequently, no iterations can be attached. For both additions and copies, the execution engine repeats applying the respective diff element for each *target* model element (i.e., the new container of the added or copied model element), if the target's template is iterative.

The benefits of supporting model element copies in composite operations is illustrated in the example depicted in Figure 4.14, which shows the specification of the refactoring *Push Down EOperation*. By applying this composite operation, an EOperation contained by a specific EClass is pushed down to all its subclasses. To support several EOperations to be pushed down to two or more subclasses, this operation specification contains two iterations, one for the EOperation and one for the second subclass. The detected diff elements after the user's demonstration are a *move* of EOperation_0 from EClass_0 to EClass_1 as well as the addition of EOperation_1. In fact, however, by this addition a copy of the model element represented by the template EOperation_0 is created. Therefore, the user may select the diff element representing the detected addition and turn it into a *copy* and specifies EOperation_0 to act as the copy source. This diff element is denoted with `<+#{EOperation_0}>` in the postconditions area in Figure 4.14[6], whereas EOperation_0 indicates the copy source. Thereby, when applying this operation specification to an arbitrary model, instead of adding the operation() model

[6]Please note that this syntax is only used here; in EMO, these annotations are attached using a more user-friendly user interface.

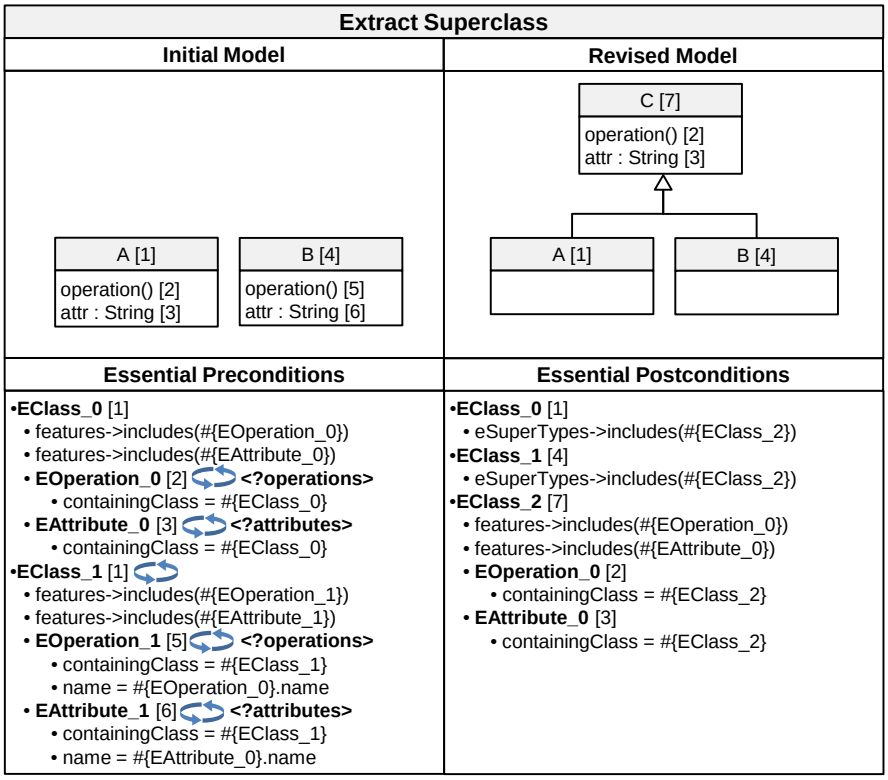


Figure 4.15: *Extract Superclass* for Illustrating Optional Templates

element once, all model elements bound to the template EOperation_0 are copied to the second subclass. As a result, the correct properties of the EOperation are preserved and also containments (e.g., contained parameters) are regarded. Furthermore, the iteration attached to template EOperation_0 is accordingly applied.

Optional Templates

By default, every model element in the initial model is considered to be *mandatory* for the composite operation. When parts of a composite operation, however, should only be processed if a specific model element exist, the user would have to create separate operation specifications. To allow for optional parts within one specification, we introduce *optional templates*. Thereby, the diff elements that affect the optional template's model element is only performed if the optional template is matched with a corresponding model element; otherwise, these diff elements are omitted.

In several scenarios, however, it is not sufficient to consider only single optional templates. More precisely, if a model part (consisting of several model elements) in an arbitrary model should only be processed and if *all* model elements belonging to this part exist, we need a mechanism to bundle together several optional templates into coherent groups. Therefore, we introduce *optional groups*, which contain one or more optional templates. These groups indicate

that the model elements bound to optional templates should *only* be processed if *all* optional templates within one coherent group can be matched successfully. In this case, the whole group is be processed altogether; otherwise the whole group is omitted. Consider, for instance, the refactoring *Extract Superclass*: from two or more classes, a new superclass is extracted. This extracted superclass should contain all common attributes *and/or* operations of its subclasses after the composite operation has been applied (cf. Figure 4.15). It is, however, very important that only common attributes *and/or* operations are pulled up that are contained by *all* subclasses. Otherwise, the refactoring application would change inadvertently the model's semantics because at least one subclass that originally did not have the attribute or operation would now inherit it from its new superclass. Consequently, the templates representing the attributes (EAttribute_0 and EAttribute_1) as well as the templates for the operations (EOperation_0 and EOperation_1) can not be optional on their own. Instead, we use two option groups—one for the two templates representing both operations and one for both attributes. Optional groups are notated with `<?groupid>` next to the names of its contained templates. Consequently, we have the optional group `<?attributes>` and `<?operations>` in the preconditions area in Figure 4.15.

To support option groups, we apply a multi-phase template matching process. In particular, the first phase searches for valid template bindings *without* optional templates; that is, only mandatory templates are bound. During this run, all conditions referring to optional templates are deactivated. In the second phase, the template matching engine tries to *extend* each previously found ConditionModelBinding by matching all optional templates of each optional group. If it succeeded to find an extended binding for all templates of an optional group, we may apply either one of two different integration strategies. The first strategy, called *replace base binding*, replaces the extended base binding (i.e., the binding for which a valid extension for one option group could be found) with the extended binding. With this strategy, the optional model elements must be processed by the composite operation, given the model elements exist. The second strategy, called *optionally extend base binding*, adds the extended binding to the collection of all valid ConditionModelBindings while retaining the extended base binding as it is. As users may still remove valid bindings from the collection of all bindings before proceeding with applying the composite operation, with this strategy, the user may choose whether to incorporate the optional model elements in the composite operation application or not. These two strategies are configured in terms of a flag in the optional groups by the user who creates the operation specification.

So far, optional groups have to be configured manually after the demonstration. The only way to infer such option groups automatically from a demonstration is to ask the user for additional initial models and demonstrations: if templates could not be matched in an additionally specified initial model, these unmatched templates can be considered as optional and configured accordingly.

Negative Application Conditions By Demonstration

Negative application conditions (NACs) proved successful in graph transformations. With NACs, the user has a powerful mechanism at hand to specify descriptively forbidden model patterns.

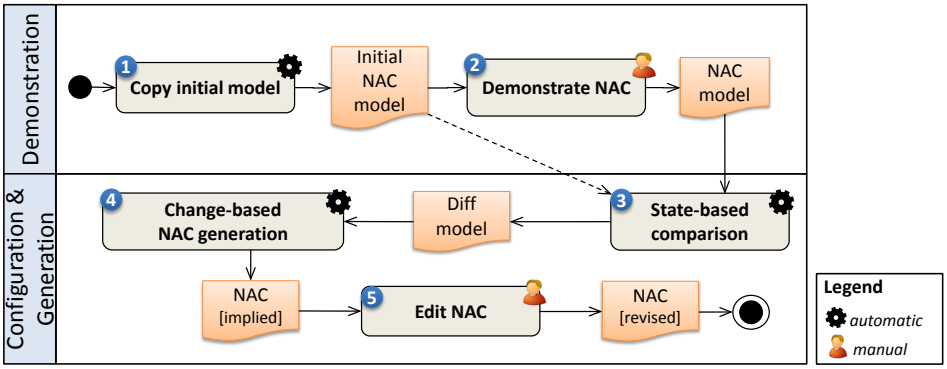


Figure 4.16: Process of Specifying Negative Application Conditions (NAC) By Demonstration

Thus, we introduce *Negative Application Conditions By Demonstration* in EMO to include this powerful mechanism while still adheres to the specification by demonstration paradigm.

The process for specifying NACs by demonstration is depicted in Figure 4.16. This process comprises five steps. In particular, when the user decides to create a new NAC, a copy of the initial model is created in step 1. This copy, called initial NAC model, is opened in a diagram editor, in which the user demonstrates the forbidden scenario in step 2 by either *modifying* existing model elements or *adding* new model elements. In the next step, a diff model is created, which explicates the atomic operations that have been applied to the initial model for demonstrating the NAC. Next, the conditions constituting the NAC are generated based on the detected diff model. In contrast to the usual condition generation applied in the general operation specification process, *refined templates* are created for model elements that already exist in the initial model. These refined templates refer to the corresponding original template in the precondition model of the operation specification (i.e., the template that is refined by the refined template). For refined templates, only conditions for modified feature values according to the diff model are generated. For unmodified feature values, no conditions are created. For instance, if the user only modifies the name of an originally existing model element to "a", only the condition `name = 'a'` is generated and added to the refined template. If new model elements have been added during the NAC demonstration, the usual templates and conditions are generated. The generated NAC condition model (NAC [implied]) may be fine-tuned manually by the user in step 5 as the automatic generation might not always reflect the user's intention. Finally, the resulting revised NAC is added to the operation specification.

An example for a composite operation, in which NACs are practical, is depicted in Figure 4.17. In this composite operation, common attributes are pulled up from two or more subclasses to their common superclass. The depicted NAC (NAC1) ensures that the common superclass has no attribute already with the same name. The actual negative application conditions of NAC1 are depicted in the lower right area of Figure 4.17. Refined templates are printed in *italic* and contain only conditions for features, which have been modified during the NAC demonstration process. The only new model element that has been introduced during the NAC demonstration is represented by the NAC template called EAttribute_2 [6].

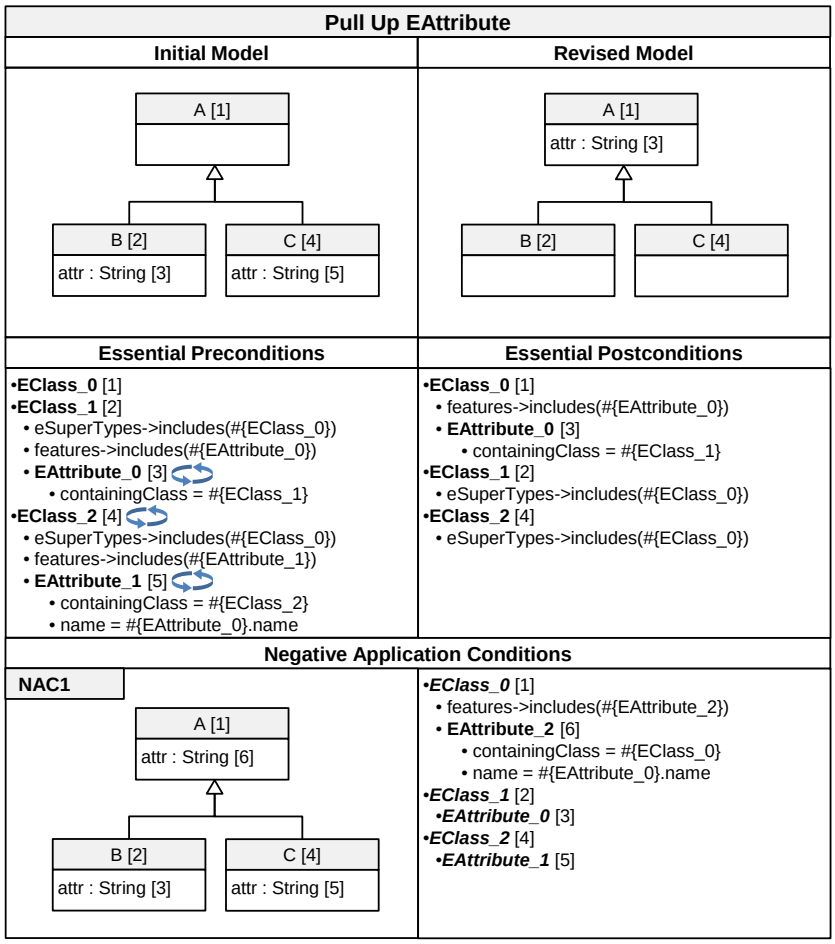


Figure 4.17: *Pull Up EAttribute* for Illustrating Negative Application Conditions

To also regard NACs in the execution of operation specifications, we extended our template matching process. In particular, we first evaluate the precondition model as usual (cf. Section 4.1.4). Subsequently, each of the resulting precondition model bindings is then separately evaluated against each NAC in the operation specification. Thereby, model elements that are bound in the precondition model binding are used as initial binding of model elements to refined templates for the evaluation of the NACs. Thus, in our example in Figure 4.17, the initial binding contains a binding of all refined templates, namely EClass_0, EClass_1, EAttribute_0, EClass_2, and EAttribute_1. Now, these bindings are checked again with the conditions contained in the respective refined templates because additional conditions, introduced by the NAC, may invalidate them. Then, the template matching engine tries to complete this initial binding by searching for bindings for the introduced NAC templates (i.e., model elements that have been introduced during the NAC demonstration such as EAttribute_2 in Figure 4.17). If a complete valid binding for a NAC is found, the base binding is removed from the collection of valid precondition model bindings.

Listing 4.1: Corresponding OCL Constraint for NAC1 of Figure 4.17

```
context EClass_0
self.features->forAll(a | a.name <> #{Attribute_0}.name)
```

NACs have the same impact as negatively formulated conventional preconditions. In our example, for realizing the same semantics as NAC1, a custom precondition could be added manually for ensuring that the superclass has no attribute with the same name as EAttribute_0 (cf. corresponding OCL constraint in Listing 4.1). However, we feel that NACs are in many cases more straightforward and easier to specify due to their descriptive and *demonstrate-able* nature than custom hand-written conditions. Furthermore, the major difference between preconditions and NACs lies in the multi-phase matching process. That means, preconditions are employed for *finding a valid binding* and NACs are employed for *rejecting forbidden bindings* among those bindings obtained from matching precondition templates.

Non-Existence Templates

An interesting alternative to NACs for specifying forbidden model elements is to mark these forbidden model elements directly in the initial model. Therefore, we introduce so-called *non-existence templates*, whereby users create the initial model as usual, but they also incorporate forbidden model elements in this model. After the preconditions have been generated, the user marks the templates that represent the forbidden model elements as non-existence templates. After these templates are turned into non-existence templates, a valid binding may not include model elements matching these non-existence templates. Similar to optional template groups, as introduced above, non-existence templates can be grouped to non-existence groups. With non-existence groups, a valid binding is only invalidated if *all* non-existence templates within one non-existence group can be matched. As non-existence templates must not be bound in a valid template binding, the model elements represented by non-existence templates cannot be modified in the course of the composite operation. Thus, there must be no diff element in the operation specification's diff model that affects non-existence templates.

As an example for non-existence templates, we depicted the operation specification for *Pull Up EAttribute* in Figure 4.18, in which non-existence templates are used to ensure that no attribute already exists in the superclass having the same name as the attribute to be pulled up. The non-existence template is marked in this figure with the annotation <!attribute> in the preconditions area and the corresponding model element is crossed out in the concrete syntax representation of the initial model in Figure 4.18. Ultimately, this operation specification expresses exactly the same as the one depicted in Figure 4.17 using NACs.

Non-existence templates are semantically equal to NACs. Moreover, NACs are even more powerful because they may also be used to prohibit *characteristics of bound model elements* by modifying these model elements contained by the initial model during the NAC demonstration. With non-existence templates, only *additional* model element patterns "surrounding" existence templates (i.e., usual templates) may be prohibited. It might be a matter of the user's preferences whether to prefer NACs or non-existence templates. Nevertheless, non-existence templates are

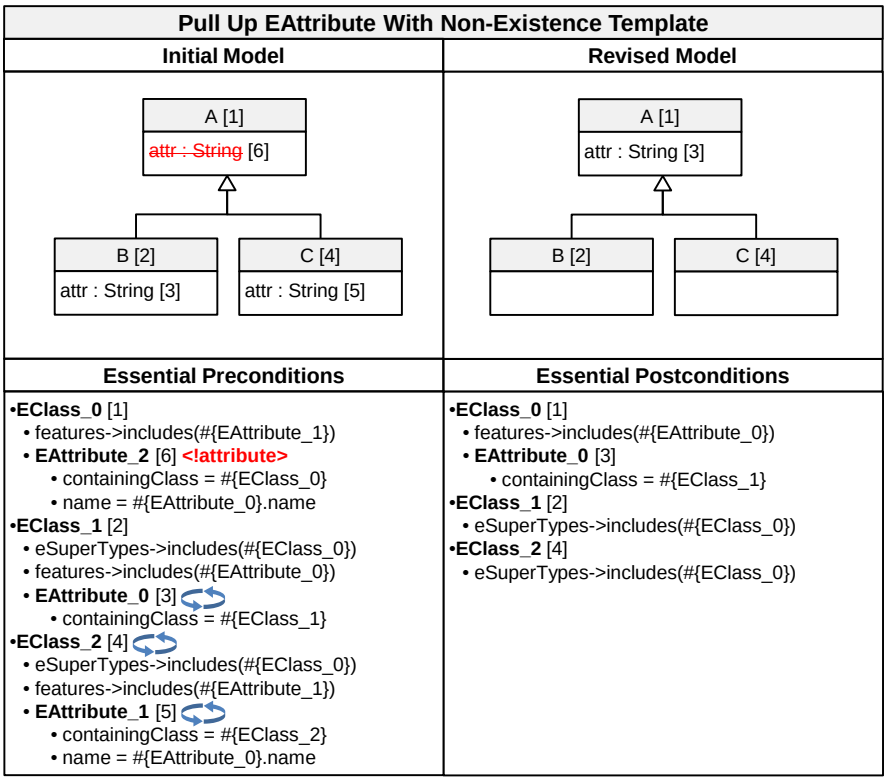


Figure 4.18: *Pull Up EAttribute* for Illustrating Non-Existence Templates

definitely very powerful *in combination* with NACs. A non-existence template used in a NAC ensures that a specific model element *must exist* due to the double negation of NACs and non-existence templates in combination. This can be important particularly in some scenarios.

Consider the operation specification depicted in Figure 4.19. This operation specification specifies the refactoring *Pull Up EAttribute* using a non-existence template for ensuring that the superclass does not already contain a property having the same name as the property to pull up. However, additionally, there is a NAC also incorporating a non-existence template. This NAC introduces a new subclass with an attribute of the same name as the attribute to pull up. The corresponding template (EAttribute_3 in Figure 4.19), however, is marked as a non-existence template. This combination of a NAC and a non-existence template ensures that there *must not be another subclass that does not contain an attribute* having the same name as the attribute to be pulled up. Otherwise, this class would inherit the pulled up attribute accidentally after the composite operation has been applied. The same constraint expressed with plain OCL is much more complicated and would have to be written manually instead of being modeled (cf. Listing 4.2 for a semantically equivalent OCL constraint). Without such a specification, subclasses not having this attribute would simply not match; as soon as there are some other subclasses having the correspondingly named attribute, the operation specification, however, would still be applicable, although it would modify the semantics of the model inadvertently.

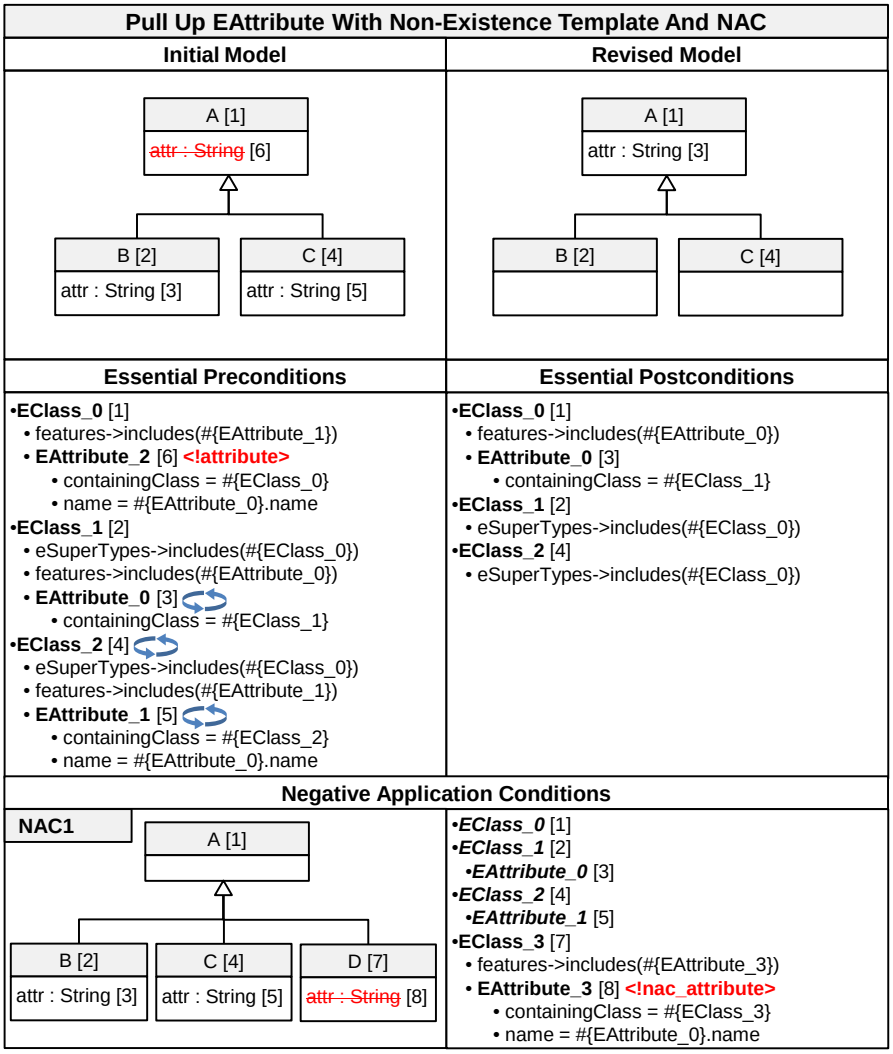


Figure 4.19: *Pull Up EAttribute* for Illustrating Non-Existence Templates in NACs

Similar to the realization of optional templates, non-existence templates are implemented using a multi-phase matching process. The first phase searches for valid precondition model bindings *without* non-existence templates. During this run, all conditions involving non-existence templates (i.e., referring to non-existence templates using `#{NonExistenceTemplate}`) are deactivated. In the second phase, the engine tries to extend each found binding with all non-existence templates of each non-existence group. If a valid binding could be extended by a valid match for one non-existence group, its base binding is removed from the collection of valid bindings because it is extensible to a valid match for at least one non-existence group. Note that model elements, which are represented by a non-existence template, are removed from the revised model and are also omitted when creating the model copy for demonstrating NACs.

Listing 4.2: Corresponding OCL Constraint for NAC1 of Figure 4.19

```
context EClass_0
self.ePackage.eClassifiers->select(c |
  c.oclIsTypeOf(EClass) and
  c.oclAsType(EClass).eSuperTypes->includes(#{EClass_0}))->forAll(c |
    c.features->exists(a |
      a.oclIsTypeOf(EAttribute) and a.name = #{EAttribute_1}.name
    )
  )
)
```

4.1.7 Related Work

In this section, we give an overview of work particularly related to the presented model transformation by demonstration (MTBD) approach for endogenous model transformations. The related work is organized in the categories *composite operations for models*, *user-friendly model transformation*, and *model transformation by example*.

Composite operations for models. Most existing approaches for developing composite operations focus solely on model refactorings. One of the first investigations in this area was done by Sunyé et al. [SPLTJ01]. In particular, they recognize the importance of model refactorings as an essential element in the software development lifecycle to increase the quality of the models. They illustrate a set of refactorings of UML class diagram and UML state machines. These simple transformations are defined by complex pre- and postconditions using OCL. Boger et al. [BSF02] present a refactoring browser for UML supporting the automatic execution of predefined UML refactorings within a UML modeling tool. Whereas these two approaches only focus on predefined refactorings, approaches by Porres [Por05], Zhang et al. [ZLG05], Kolovos et al. [KPPR07], and Verbaere et al. [VEdM06] allow the introduction of user-defined refactorings by describing the preconditions and the effect of refactorings in dedicated textual programming languages. A similar idea is followed by Mens [Men06] and Biermann et al. [BEK+06], who use, instead of textual programming languages, graph transformations to describe the refactorings within the abstract syntax of the modeling languages. The application of this formalism comes with the additional benefit of formal analysis possibilities of dependencies between different refactorings. Most of the approaches cited above provide automatic execution facilities of predefined or user-defined refactorings. However, the definition of new refactorings requires extensive knowledge of the modeling language's metamodel, of special APIs to process the models, or a dedicated programming language. In other words, very specific expertise is demanded. More recently, Reimann et al. [RSA10] proposed to specify model refactorings in a more generic role-based manner in order to avoid having to specify similar refactorings for each modeling language over and over again. In particular, refactorings are specified against *roles* and not against concrete metaclasses of a modeling language's metamodel. In this context, roles encapsulate the behaviour of a model element, comparable to our notion of templates. Therefore, Reimann et al. introduce a dedicated refactoring specification language having a proprietary graphical as

well as textual syntax. The roles of a specified refactoring may then be bound (e.g., by the language designer) to metaclasses of modeling languages later, to allow for applying the generic refactoring to models conforming to the respective modeling language. In comparison to EMO, our approach does not allow for specifying cross-language refactorings. Nevertheless, the refactoring specification using the approach by Reimann et al. requires dedicated knowledge on the applied transformation language as well as the modeling language's metamodel.

EMO yields an orthogonal extension of existing approaches by providing a tool-guided process tailored to be applied by users, who are not familiar with programming or transformation languages, as well as a modeling language's metamodel, for defining the refactorings by modeling examples. The otherwise manually created refactoring descriptions are automatically generated from which representations in any language or formalism (e.g., graph transformation) may be derived.

User-friendly model transformation. Defining model transformation rules by using the abstract syntax of graphical modeling languages comes, on the one hand, with the benefit of the generic applicability; on the other hand, however, the creation of such transformation rules is often complicated and their readability is much lower compared to working with the concrete syntax, as reported in several papers [BW07, dLV02, SW08, Var06]. As a solution to this issue, the usage of the concrete syntax for the definition of the transformation rules has been proposed (e.g., in AToM^3 [dLV02]). More recently, Baar and Whittle [BW07] discuss requirements and challenge how to develop transformation rules in concrete syntax within current modeling environments and propose an approach to specify refactorings using graph transformations in the concrete syntax of the modeling language. Therefore, they change the metamodel by adding an attribute called label to each class, by setting each class to non-abstract, and, finally, they turn all attributes into optional attributes. This modified metamodel may be used to specify the left and right side of a graph transformation rule. Additionally, they allow the specification of preconditions to express the constraints for applying the refactoring. In these preconditions, classes can be accessed using the previously mentioned labels, which is comparable to our concept of preconditions and templates. Still, there are some major differences between EMO and the approach by Baar and Whittle. Baar and Whittle modify the metamodel; consequently, the original editors, which only support the original metamodel, cannot be used anymore. Moreover, their proposed graphical syntax has to be adopted to the syntax of the concrete metamodel to avoid ambiguity of the graphical notation. In contrast, when using EMO, the metamodel does not have to be modified, thus, users may still use the original modeling environment and no graphical syntax adaptation has to be done for each modeling language. However, in contrast to the approach by Baar and Whittle, EMO does not allow to specify composite operations against abstract classes, because they do not have a representation in the concrete syntax and, hence, cannot be specified in the concrete example models. A workaround for this limitation is modeling a concrete example and then modifying the type information to the abstract superclass. There are also tailored approaches for defining specific kinds of transformations in concrete syntax. For example, Lechner [Lec04] presents an approach that is tailored to describing transformation rules for WebML models. In the field of aspect-oriented modeling, transformations are also required for weaving aspect models into base models. Whittle et al. [WMA$^+$07] describe aspect composition speci-

fications particularly for UML models by using their concrete syntax. In contrast, EMO is not restricted to one modeling language and can be used for any language residing in the EMF.

Model transformation by example. Strommer and Wimmer [SW08] and Varro [Var06] go one step further by proposing to specify transformations purely by example. Instead of developing transformation rules, an example input model and the corresponding output model are given. From these example pairs, the general transformation rules are derived by a reasoning component. A more detailed discussion of model transformation by example (MTBE) is given in Section 2.3.2.

However, the focus of current MTBE approaches lies on exogenous model transformations between different languages such as UML class diagrams to relational models. Endogenous transformations required for composite operations such as refactorings have not been considered by MTBE approaches. Besides that EMO is tailored for endogenous model transformation, it is characterized by another major difference to current MTBE approaches. As shown in Section 2.3.2, all MTBE approaches are based on model correspondences between the source model and the target model. From this correspondences, which usually have to be created manually, these approaches deduce the exogenous transformation rules. In contrast, EMO follows a *demonstration-based* approach; that means, it does not require correspondences to be set up first. Instead, fine-grained operations are computed between the source and the target model, which are exploited to derive a generically applicable transformation.

With EMO, we fill the gap between composite operation definition approaches and model transformation by example. Although the need for introducing refactorings by the user of modeling tools as well as the need for describing transformations in a more user-friendly way have been frequently reported, at the time, when we initially published our approach, EMO was the first attempt to address a demonstration-based specification for endogenous model transformations. Back then, the only comparable work, we are aware of, is [RL08] which allows to specify composite operations by demonstration for program code using the Squeak Smalltalk IDE [BDN+07]. Although the general idea is similar to ours, three fundamental design differences exist: EMO operates on models and not on code, EMO is independent from any specific modeling language, and EMO may be employed for any modeling environment.

At about the same time as we published EMO in [Lan09, BLSW09, BLS+09], a very similar approach by Sun et al. [SWG09], called MT-Scribe, emerged. Besides the fact that MT-Scribe is based on the Generic Eclipse Modeling System[7] (GEMS), there are also other differences between EMO and MT-Scribe. In MT-Scribe, the changes applied during the demonstration are *recorded* and not derived by a state-based model comparison. After the demonstration, an inference engine generates a general transformation pattern, which comprises the transformation's preconditions and its sequence of operations. This pattern may, as in EMO, also be refined by the user in terms of adding preconditions and attribute value computations. In contrast to EMO, however, *Groovy*[8], a script language for the JVM, is employed to express these conditions and computations instead of OCL. In a more recent publication [SGW11a], Sun et al. extended this

[7] http://www.eclipse.org/gmt/gems

[8] http://groovy.codehaus.org

step so that users may also identify and annotate *generic operations*, which corresponds to the concept of iterations in EMO. However, these annotations are attached directly to the change model; in EMO, they are attached to templates in the preconditions and their influence on the operations in terms of repetitions is inferred. MT-Scribe does not offer optional parts in a model or negative application conditions. Nevertheless, Sun et al. present a very interesting orthogonal extension to MTBD called LiveMTBD in [SGW+11b], which is not available in EMO. The proposed concept of LiveMTBD extends MTBD by three features, in particular, *live demonstration*, *live sharing*, and *live matching*.

Live demonstration addresses the issue that users must plan ahead when demonstrating a transformation. That is, users must explicitly indicate that they aim to specify a demonstration by starting the MTBD process and explicitly end the demonstration to invoke the generalization phase. However, users often do not immediately realize that they are currently applying a pattern, which has the potential to be reused later, until they are part-way through with the transformation. Currently, after realizing that they want to demonstrate the transformation, users have to build an initial model and demonstrate the transformation all over again. This inconvenience is addressed with live demonstration by continuously recording every editing operation performed in the editor. As soon as a user realizes the need to specify and summarize a certain model transformation pattern, users can indicate the starting point of demonstration a posteriori and select all the applied operations that shall be part of the specific transformation they want to reuse later. Live sharing allows users to share specified transformations instantly via a central transformation repository across the network with others. This enables users to exchange and benefit from each other's transformations. Users are often not aware of the existence of specified transformations that, in a specific situation, might be appropriate for them. Therefore, live matching allows users to see a list of specified transformations that are applicable in the user's current modeling scenario. Being aware of currently applicable transformations, users may select the one that seems to be appropriate and apply it.

Sun et al. also address a very challenging issue of model transformations, which is not implemented in EMO. This issue concerns the concrete syntax of a model after the transformation has been applied. Currently, the transformations only regard the abstract syntax of a model and not it's graphical layout. As a consequence, the graphical layout is often obfuscated after the model's abstract syntax has been modified by a transformation. For instance, added model elements do not show in the graphical view or are added at an arbitrary position in the graphical view. Therefore, in [SGK+11,SGL+10], Sun et al. proposed to enable users to specify the model layout in a model transformation. In particular, MT-Scribe has been extended to let users specify the layout information using the concept of "What You See Is What You Get" (WYSIWYG), so that the complex layout specification can be simplified. By using this extension, the position of existing and new model elements may be indicated in a WYSIWYG-like manner during the demonstration of the model transformation. As the position of model elements in a layout often depend on the outermost borders of a diagram or on the position of other model elements, the user may specify also relative locations. EMO does not allow for any configurations regarding the graphical layout.

4.2 Exogenous Model Transformation By Demonstration

Exogenous model transformations are an essential constituent in model-driven engineering [FR07]. Therefore, several approaches have been proposed for easing the burden of writing model transformation rules by hand. One of the most prominent approaches is *model transformation by example* (MTBE) [SW08, Var06]. The main idea of MTBE is the semi-automatic generation of transformation rules from so-called correspondences between source and target model pairs. A similar idea is followed by model transformation by demonstration (MTBD) approaches (cf. Section 4.1). However, in contrast to MTBE, which is based on correspondences between source and target models, MTBD exploits the operations performed on an example model to gain a transformation specifications, which are executable on arbitrary models.

Until now, however, MTBD approaches were only available for endogenous model transformations such as refactorings. An open challenge is how to adapt MTBD to be applicable also for endogenous model transformations due to the following heretofore unsolved issues. First, when using MTBD for endogenous model transformations, the trace model between the initial model (before the transformation execution) and the revised model (after the transformation execution) comes nearly for free. That trace may either be achieved using an ID-based comparison [BLS+09] or by directly recording all performed actions [SWG09]. Unfortunately, these methods cannot be used for exogenous model transformation examples, because the corresponding elements in the source and target model are independently created and, consequently, have different IDs. Additionally, they are in most cases structurally heterogeneous. Second, state-of-the-art MTBD approaches for endogenous model transformations allow to specify one composite operation performed on the initial model. After generalization, the resulting composite operation is executed on arbitrary models separately from other composite operations. However, in exogenous model transformation scenarios, the whole target model has to be established from scratch based on the source model by applying a set of different, strongly interdependent transformation rules.

In this section, we address the mentioned challenges by proposing a novel MTBD approach for developing exogenous model transformations. In particular, we elaborate on how the approach presented in the previous section for developing endogenous model transformations is adapted for exogenous model transformations. To specify an exogenous model transformation, the user iteratively demonstrates each transformation rule by specifying an example using her preferred editor. Subsequently, the example models are generalized automatically to templates which the user may configure and customize by following a well-defined annotation process. Finally, transformation rules are derived from these annotated templates automatically. As the user only gets in touch with templates representing the user-specified examples, she is able to develop general model transformations without requiring in-depth knowledge of the underlying model transformation language. Please note that our approach is orthogonal to existing high-level model transformation approaches, such as triple graph grammars [Sch94] and QVT Relations [OMG05a], because instead of directly developing the generalized templates, the user first develops concrete examples which are then systematically generalized. For showing the applicability of the approach, we developed an Eclipse-based prototype that supports the generation of ATL code out of EMF-based example models.

The remainder of this section is organized as follows. Starting with a motivating example in Section 4.2.1, we outline the process of developing exogenous model transformations by demonstration in Section 4.2.2. Section 4.2.3 provides an example-based presentation of our approach. Finally, in Section 4.2.4, we discuss existing work in the area of generating exogenous model transformations from example models.

4.2.1 Motivating Example

To emphasize our motivation for developing a by demonstration approach for exogenous model transformations, we introduce a well-known model transformation scenario; namely, the transformation from UML class diagrams to Entity Relationship (ER) diagrams. Figure 4.20 illustrates the scenario which serves as a running example for presenting our approach. Although the involved modeling languages provide semantically similar modeling concepts, this scenario also exhibits challenging correspondences between metamodel elements. For example, the UML class diagram provides the modeling concept *bidirectional relationship* by modeling a pair of opposite references. In contrast, in the ER diagram such relationships are represented explicitly.

In the following, the main correspondences between the UML class diagram and the ER diagram are described briefly. Simple one-to-one correspondences exist between the root containers SimpleCDModel and ERModel, as well as between Class and Entity. However, the example also contains more complex correspondences. In particular, these are the correspondences between (1) the class Property and the classes Attribute and Type, as well as (2) between the class Reference and the classes Relationship, Role, and Cardinality. In the first case, for each property, an attribute has to be generated. However, only for each distinct value of Property.type a type should be generated. When a type already exists with the same name, it should be reused. In the second case, for *every unique pair* of References that are marked as opposite of each other, a corresponding Relationship has to be established containing two Roles, which again contain their Cardinalities. With *every unique pair*, we mean that the order in which the references are matched does not matter. For example, if Reference r1 and Reference r2 are marked as opposite, then the transformation should produce *one* relationship for the match <r1,r2>, instead of creating another one for <r1,r2>. Therefore, we speak about the matching strategy *Set* if the order of the matched elements does not matter, and *Sequence* if the order does matter. On the attribute level, only simple one-to-one correspondences occur. On the reference level, some references can be mapped easily, such as SimpleCDModel.classes to ERModel.entities. However, some references on the target side have to be computed from the context of the source side, because they miss a direct counterpart such as ERModel.relationship.

4.2.2 Exogenous Model Transformation By Demonstration at a Glance

The design rationale for our MTBD approach is as follows. Exogenous model transformations may be seen as a set of operations that are applied to the target model for each occurrence of a pattern of model elements in the source model. Thus, the target model is incrementally built by finding patterns in the source model and by applying the appropriate operations to the target model. Target elements created by these operations might need to be added to and refer to already existing elements, which had been created in prior transformation steps. Therefore,

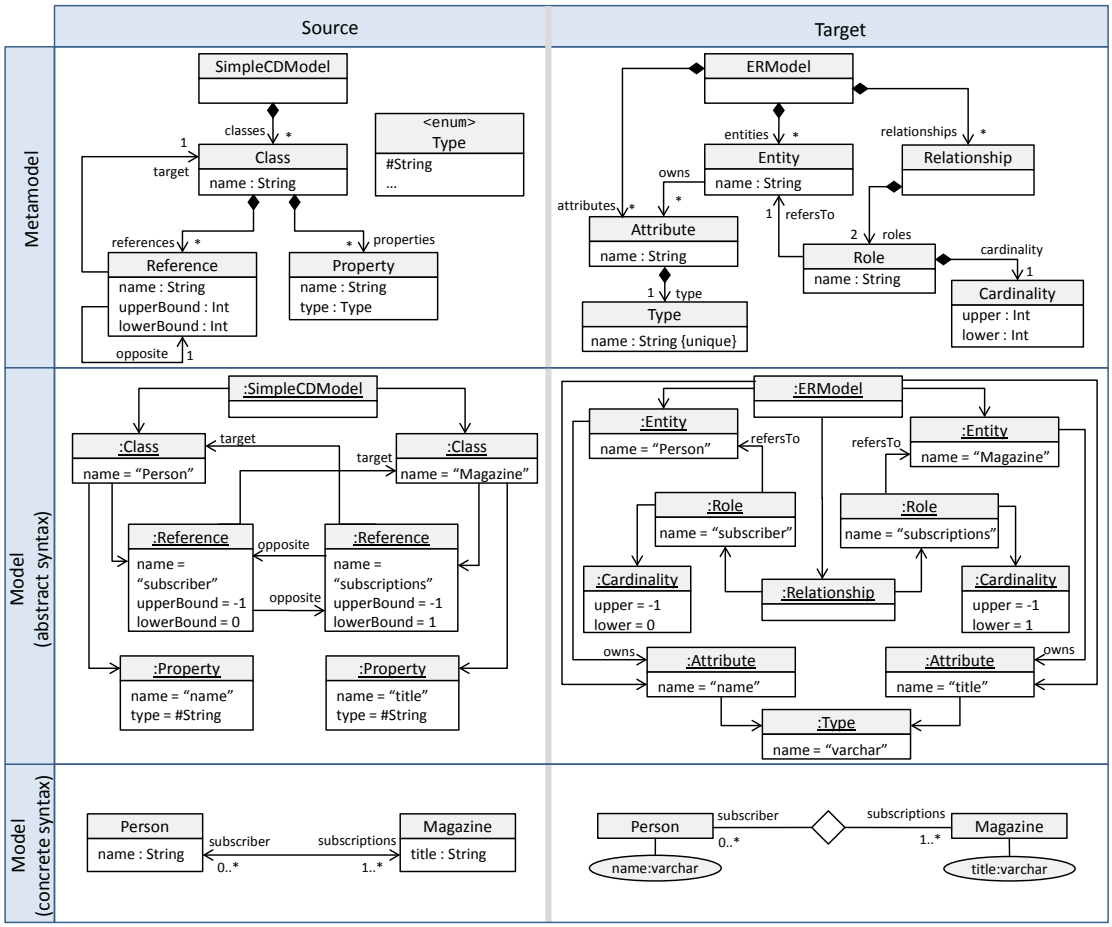


Figure 4.20: Motivating Example for Exogenous Model Transformation By Demonstration

operations have to be applied within a context. To enable the derivation of a transformation rule from examples, we apply the *model transformation by demonstration process* for exogenous transformations as depicted in Figure 4.21.

Phase 1: Modeling. The user demonstrates a single transformation rule by adding model elements to the source model and by modeling the desired outcome in the target model. As mentioned before, the elements in the source model represent the pattern for which the elements in the target model are created. A transformation usually consists of several transformation rules. If a rule does not depend on other rules, no context elements are necessary to illustrate the rule, thus the user creates empty models. But usually, rules depend on other rules, which must have been applied previously forming the context. Thus, they are called *context rules*. Therefore, the user might select a context in which a new rule is demonstrated. If a context rule is selected, the source and target example model contained by the context rule is extended by the user to demonstrate the new context-dependent rule. For ensuring a high reusability of rules as context rules, they should be as small as possible.

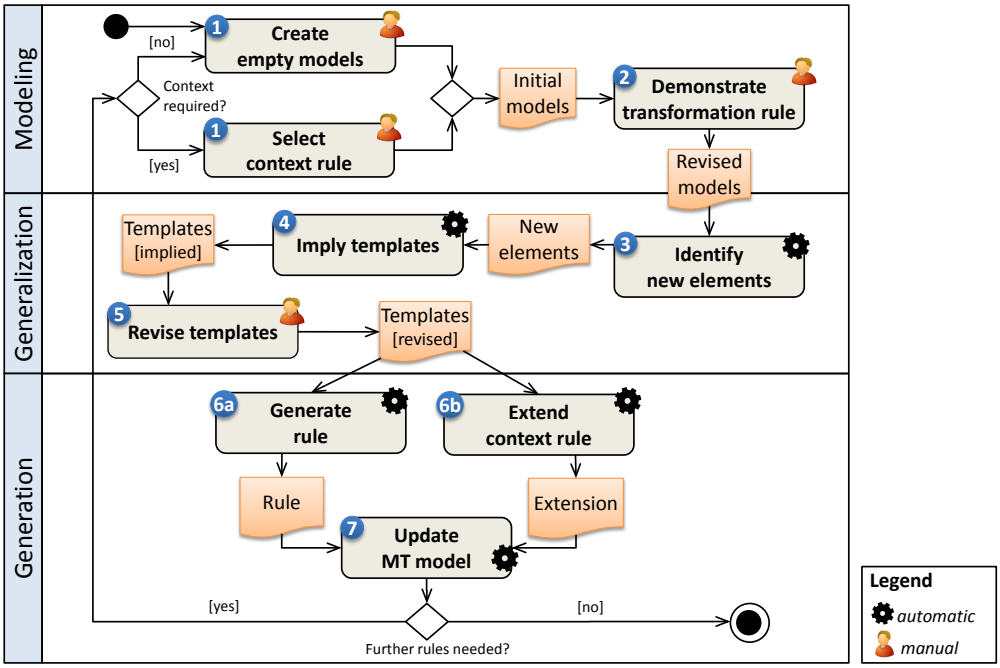


Figure 4.21: Process of Exogenous Model Transformation By Demonstration

Phase 2: Generalization. Added elements are identified and the illustrated transformation scenario is generalized. To determine the new elements if the demonstrated rule is context-dependent, we conduct a comparison between the revised models (source and target) to the respective models of the selected context rules. If the rule is context-free, all elements are considered as new. The new elements in the source model act as *trigger elements*, which trigger to create the detected new elements in the target model. The most obvious way to identify the new elements is to record user interactions within the modeling environment. However, this would demand an intervention in the modeling environment, and due to the multitude of modeling environments, we refrain from this possibility. Instead, we apply a state-based comparison to determine the executed operations after modeling the context models and the extended models. This allows the use of any editor without depending on editor-specific extensions. After the new elements are identified, we automatically derive templates for each model element in the respective models and generate feature conditions (cf. Section 4.1.4). After an automatic default configuration, such as the deactivation of conditions restricting empty feature values is applied, the user may refine templates by adding or modifying certain conditions just the same way as in our MTBD approach for endogenous transformations.

Attribute values in the target model usually depend on values in the source model. Therefore, we search for similar values in the source and the target model's elements and, for each detected similarity, automatically derive suggestions for the user to create attribute value correspondences. Accepted correspondences are incorporated by adding them to the condition in the target template. Attribute correspondence conditions bind a feature value or a combination of

feature values in the source template model to a feature in the target template model. Unambiguous correspondences are automatically added, but the user might adjust the conditions.

In this context, we distinguish between *usual* templates and *inherited* templates. Usual templates represent model elements that have been newly created in the current demonstration. Inherited templates represent context elements that have either been introduced already in a context rule, or that conform to a template in a context rule; that is, they are processed by the context rule.

Phase 3: Generation. In contrast to our approach for endogenous model transformation by demonstration, we do not directly interpret our internal representation of the transformation by means of our own transformation engine. Instead, we directly generate ATL transformations from the demonstration by applying a higher-order transformation from our internal transformation specification to an ATL transformation module. In particular, for each demonstrated scenario, a new ATL rule has to be generated and attached to the ATL module model. Furthermore, in case a context-dependent scenario has been demonstrated by the user, the existing context ATL rules have to be extended with further reference bindings to the newly introduced elements referenced by the context elements.

4.2.3 Exogenous Model Transformation By Demonstration in Action

In the previous section, we illustrated the MTBD process for exogenous transformations from a generic point of view. In this section, we show how this process is adopted from a user's point of view. In particular, we discuss each iteration necessary to solve the motivating example of Section 4.2.1. To support the user in the demonstration process, we implemented a prototype presented on our project homepage[9].

Iteration 1: Class Diagram to Entity Relationship Diagram

In iteration 1, a context-free object-to-object correspondence is illustrated to create for each SimpleCDModel instance an ERModel instance. Please note that, for the sake of readability, we omit masking template references in OCL conditions by the character # in contrast to the syntax used in the previous sections.

Step 1: Create empty models. The user creates a context-free rule by specifying an empty source model and an empty target model. These models are extended in the following steps.

Step 2: Demonstrate transformation rule. To illustrate the transformation of SimpleCDModels to ERModels, the user just has to add these elements to the empty models, as shown in Figure 4.22.

Step 3: Identify new elements. As the demonstrated rule is context-free, all model elements are considered as new.

Step 4: Imply templates. The example models are generalized by automatically implying templates for each model element. The goal of creating these templates is to describe model elements generically. With the help of source templates, we are able to verify if arbitrary model

[9] `http://www.modelversioning.org/m2m-operations`

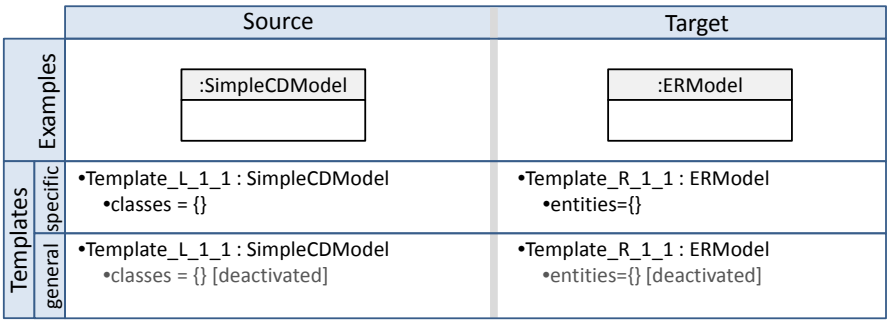


Figure 4.22: Rule 1: Class Diagram to ER Diagram

elements should be transformed equally to the illustrated model elements. Target templates indicate which properties and values should be set in the target elements according to the source model. In this example, the templates Template_L_1_1 and Template_R_1_1 (cf. *specific templates* in Figure 4.22) are implied for the respective elements in the example models. The *L* in the template name indicates the left-hand side and *R* the right-hand side. The first digit in the template name indicates the rule that has been introduced. The second digit enumerates the templates. Since both elements do not contain any classes or entities, for both templates a condition is created, which constrains these features to be empty such as `classes = {}`. After all templates are generated initially, they are pre-configured automatically and generalized. This is done by deactivating all conditions in the source templates by default. Solely, source template conditions referring to object values that are represented by other source templates are left active. Consequently, source templates only restrict the type of the elements and their dependencies to other source templates. Additionally, conditions in the target templates are deactivated if the features are not set (cf. *general templates* in Figure 4.22). This reflects an open world assumption. Only aspects are restricted that are explicitly modeled.

Step 5: Revise templates. As the templates and their contained conditions are automatically implied, they might not always reflect the user's intention. Therefore, the user may adjust the generated templates and conditions. The user may relax currently active conditions, enforce currently inactive conditions, or modify existing conditions. Additionally, templates may be augmented by adding annotations. Using these techniques, the user might for instance tighten source templates by enforcing (reactivating) or modifying certain conditions to restrict the execution of a transformation rule. However, in this iteration none of these are necessary.

Step 6-7: Generation. The revised templates are transformed into ATL transformations. The source templates are transformed into the `from` block and the target templates into the `to` block of an ATL rule. Additional conditions in source templates are used as *guards* and attribute correspondences are set accordingly via bindings. The generated ATL rule for this iteration is shown in Listing 4.3 on page 129 (line 4-6 and 11). Step 6b is not applicable for context-free rules, since no context rule has to be extended.

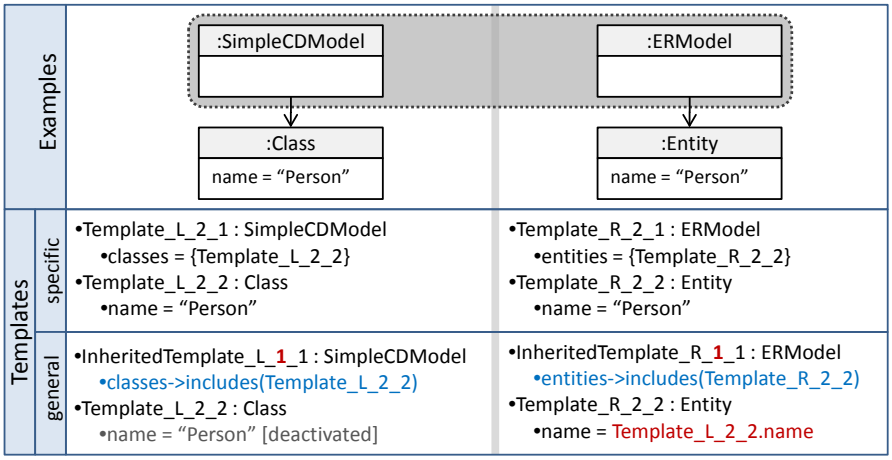


Figure 4.23: Rule 2: Class to Entity

Iteration 2: Class to Entity

In this iteration, the transformation of Classes to Entities is demonstrated. This rule requires a one-to-one object correspondence, a value-to-value correspondence, and a context—the created target elements have to be added to an ERModel instantiated in the previous iteration. The example models, the implied templates, and the generalized templates are depicted in Figure 4.23.

Step 1: Select context rule. Classes and Entities are always contained by SimpleCDModels and ERModels, respectively. Thus, the user has to select the transformation rule of iteration 1 to be the context of the rule created in this iteration. When a context is selected, a copy of the context rule's example models is created and opened in diagram editors in order to be extended.

Step 2: Demonstrate transformation rule. The user extends the loaded context models to illustrate the transformation of a Class to an Entity. An instance of both model elements have to be added in the respective models. To allow a subsequent automatic detection of attribute value correspondences, the user should use exactly the same values for which a correspondence exists. Consequently, the class is named equally to the entity (cf. Figure 4.23).

Step 3: Identify new elements. New elements are identified automatically by comparing the current source model to the source model of the context rule, as well as the current target model to the context rule's target model. Thus, the class and the entity are marked as new elements.

Step 4: Imply templates. Like in the previous iteration, for each element in the example models, a template is implied and a condition for each feature is added to the template (cf. *specific templates* in Figure 4.23). In contrast to the previous iteration, the current rule depends on a context; that is, it includes context elements to be processed by the context rule. For that reason, templates that represent a context model element are replaced during the generalization mechanism with InheritedTemplates pointing to the respective template contained by the context rule (cf. *general templates* in Figure 4.23). The first digit of the template name indicates the context rule in which the elements have been introduced, such as InheritedTemplate_R_1_1 represents the ERModel introduced in iteration 1. Note that this inherited template is refined

in this iteration by an additional condition (`entities->includes(Template_R_2_2)`. This condition indicates that the created entity has to be added to the feature ERModel.entitites. The conditions of the source templates are again deactivated by default. Additionally, for setting attribute correspondences, for each value in the target model, a corresponding value in the source model is searched. If an unambiguous correspondence is detected, the target value is automatically restricted to be the value of the source element's attribute by replacing the value assignment (`name = "Person"`) with a template reference (`name = Template_L_2_2.name`).

Step 5: Revise templates. As in iteration 1, no user adjustments are necessary due to the accurate default implications.

Step 6-7: Generation. After the generalization phase, the current rule is transformed into an ATL rule, as shown in Listing 4.3 on page 129 (line 15-17 and 20). Obviously, the generated rule only takes concrete templates and not inherited templates into account. The source template pattern is used to create the `from` block and the target template including the found attribute correspondences is transformed into the `to` block. As mentioned in step 4, we refined InheritedTemplate_R_1_1 with a new condition. This condition preserves the relationship of the context element ERModel to the newly added Entity. Hence, an assignment of generated Entities to the feature entities is added to the context rule (cf. line 7 in Listing 4.3).

Iteration 3: Property to Attribute

The transformation of Properties to Attributes is demonstrated. Properties are contained by classes whereas attributes are contained directly by the root model element. Entities only incorporate a reference to the attributes they own. Moreover, in class diagrams, the property type is expressed using an attribute. In contrast, the type of an attribute in ERModels is represented by an additional instance. Thus, we need to specify a one-to-many object correspondence, as well as two value-to-value correspondences.

Step 1: Select context rule. This transformation rule has to be illustrated within the context of rule 1 and rule 2, because Attributes are referenced by ERModels as well as by Entities.

Step 2: Demonstrate transformation rule. In the source model the user adds a property to the class created in iteration 2. Correspondingly, an attribute with the same name is appended to the entity (cf. Figure 4.24). Corresponding to the type of the property, an instance of Type has to be created in the target model and linked to the attribute.

Step 3: Identify new elements. As in the previous iterations, the new elements are identified properly using the state-based comparison.

Step 4: Imply templates. For each model element, a template is implied. Model elements that have already been created in previous iterations are represented by inherited templates. As in the previous iteration, the both inherited templates in the target template model are refined by additional conditions (e.g., `attributes->includes(Template_R_3_3)`), because the attribute is referenced by the entity and contained by the ERModel. The value-to-value correspondence regarding the attribute name is detected and annotated automatically (`name = Template_L_3_3.name`).

Step 5: Revise templates. In contrast to the previous iterations, the user now has to apply two augmentations. First, the type has to be reused because it is not intended to add a new type each time an attribute is created. Instead, Type instances have to be reused whenever a type

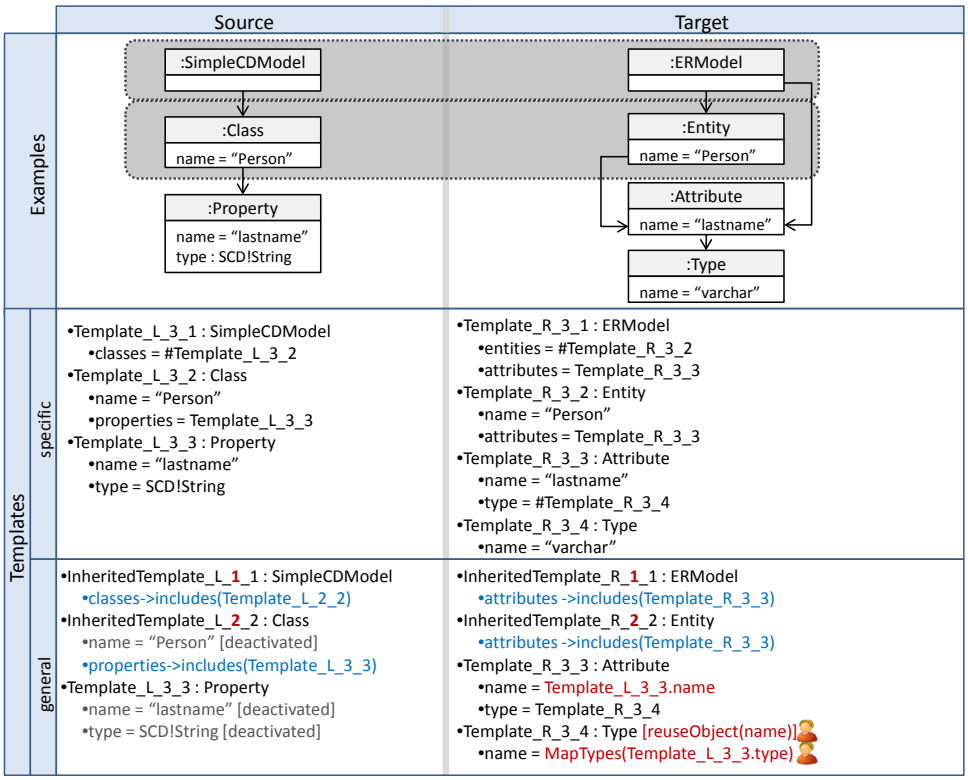


Figure 4.24: Rule 3: Property to Attribute

already exists with the same name. This is done by annotating the corresponding template with the *reuseObject* operator and providing the name feature as a discriminator for reuse. Second, the literals of the Type enumeration of the class diagram have to be converted to String values in ER diagrams. To enable such static value-to-value conversions, the user may set up a mapping table. In the template conditions, this table is used by calling its name (cf. `name = MapTypes(Template_L_3_3)`).

Step 6-7: Generation. A matched rule is created to generate attributes from properties (cf. line 27-37 in Listing 4.3 on page 129). A lazy rule and a helper is generated for creating types if necessary (cf. line 39-45 and 25). Furthermore, a helper for the mapping table is generated (cf. line 22-23). Both ATL rules created for the used context rules are extended by new feature bindings (cf. line 8 and 19).

Iteration 4: Reference to Relationship

The last iteration demonstrates the transformation of References to Relationships. For this, a many-to-many object correspondence is needed, since two references marked as opposite are transformed into a relationship with two Roles comprising Cardinalities (cf. Figure 4.25). As in the previous iteration, the context rule 2 has to be used. Furthermore, tuples of reference instances have to be processed only once by applying the *Set* matching strategy (cf. Section 4.2.1).

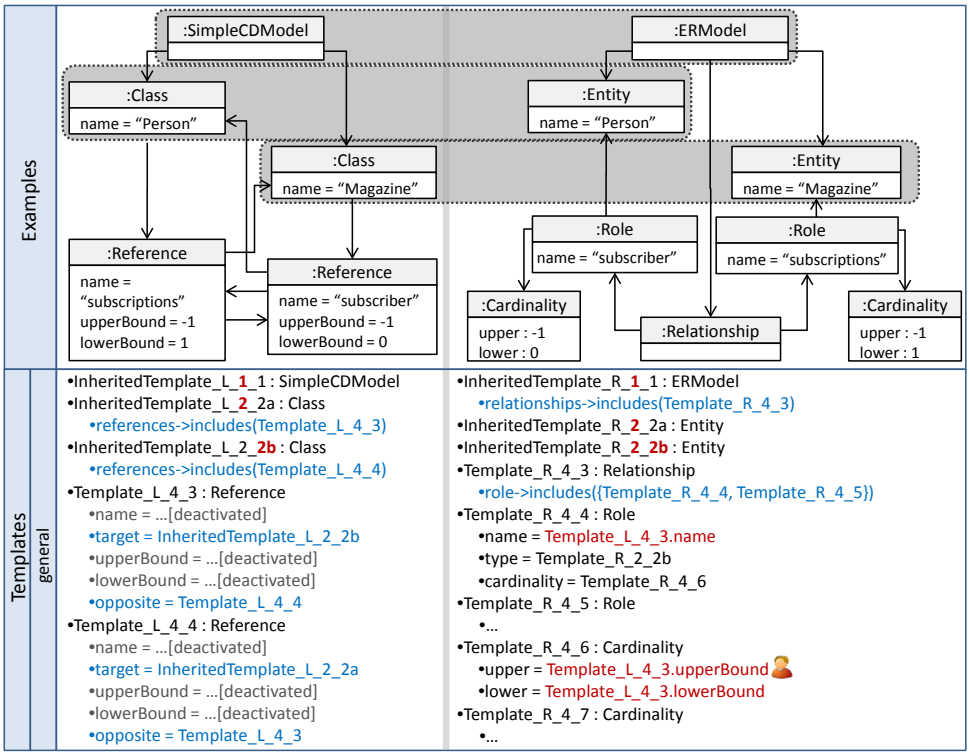


Figure 4.25: Rule 4: Reference to Relationship

In our experience, this is the intuitive matching strategy for multiple query patterns matching for the same type, thus it is used as default.

Step 1: Select context rule. The transformation of References into Relationships is in the context of rule 1 and of rule 2.

Step 2: Demonstrate transformation rule. A new class named "Magazine" and two new references ("subscriber" and "subscriptions") are added to the source model. In the target model, the user adds an Entity "Magazine", a relationship, two roles, and cardinalities for each role. All values in the target are set consciously according to the corresponding values in the source model.

Step 3: Identify new elements. Besides the Class and Entity "Person", which has been directly added in the context rule, the user also added a second Class and correspondingly a second Entity named "Magazine". To identify these two "Magazine" elements correctly to be context elements and not new elements, we match each added element in the example against all context templates. Since the left and right "Magazine" elements are matching the corresponding context templates, they are considered as context elements.

Step 4: Imply templates. For each model element, a template or an inherited template is created. Since there are now two instances of Class and of Entity, we enumerate the corresponding inherited templates with a letter (a and b). The InheritedTemplate_R_1_1 (representing a ERModel) is extended for this rule by an additional condition specifying the reference to the

added relationship. All attribute value correspondences are detected and annotated automatically. Solely, for the upper bounds of the Cardinalities, only suggestions can be made since the values of these features cannot be unambiguously mapped.

Step 5: Revise templates. The current transformation rule should only be executed for reference pairs that are marked as opposite of each other. As already mentioned before, all source template conditions are deactivated, except for those referring to an object value that is represented by another template. Consequently, the two conditions restricting the two references to be opposite of each other remain active, which is the intended configuration. Also the aforementioned difficulty regarding the matching strategy of these two instances of Reference is solved by using the matching strategy *set*, which is adopted by default. With this strategy, every combination of the reference instances irrespective of their order is "consumed" during the transformation. Consequently, two references that refer to each other are only processed once by the resulting rule. This mechanism may be compared to the *unique* keyword for lazy rules in ATL. If this is not intended, the user may annotate the templates to use the matching strategy *Sequence*.

Step 6-7: Generation. To realize the aforementioned *Set* matching strategy, we generate a *unique lazy rule* with a guard expression (cf. line 47-61 in Listing 4.3 on page 129), which is called by the root rule to create and to add the relationships to the feature relationships of ERModel (cf. line 10-12).

4.2.4 Related Work

Varro [Var06] and Wimmer et al. [WSKK07] were the first to develop exogenous model transformation by example. Both used input models, corresponding output models, and the alignments between them to derive general transformation rules. In [BV09], Balogh and Varro extended their MTBE approach by leveraging the power of inductive logic programming. As before, the input of their approach are one or more aligned source and target model pairs that are translated to Prolog clauses. These clauses are fed into an inductive logic programming engine that induces inference rules that are translated into model transformation rules. If these rules do not entirely represent the intended transformation, the user has to refine either the generated rules directly or has to specify additional model pairs and start another induction iteration. That approach might require less user interaction compared to our approach, but we follow a different conceptual aim. By our demonstration approach, we are aiming at a very interactive approach. In particular, the user is guided to demonstrate and configure each transformation rule iteratively. To ease that interaction for the user, in each iteration the user may focus only on one part of the potentially huge transformation until the current rule is correctly specified. We believe, this is a natural way of dividing and conquering the whole transformation.

For avoiding having to define alignments between source and target models manually, two further approaches have been proposed. First, García-Magariño et al. [GMnGSFF09] propose to develop exogenous transformations by annotating the source metamodel and the target metamodel with additional information, which is required to derive transformations based on given example models. Because the approach of García-Magariño et al. uses a predefined algorithm to derive the transformations purely automatically, the user has no possibility to influence the generalization process, which is in our point of view a must for developing model transforma-

tion in practice. The implication of this limitation is that most attribute correspondences cannot be detected as well as configurations, such as reusing existing objects for aggregation or determining the matching strategy Sequence or Set cannot be considered during the generalization process. The only possibility for the user is to adapt and extend the generated ATL code, which is more difficult compared to providing such configurations in our proposed template language. Second, Kessentini et al. [KSB08] interpret exogenous transformations as an optimization problem. Therefore, Kessentini et al. propose an adapted version of a particle swarm optimization algorithm to find an optimal solution for the transformation problem, which is described by multiple source and target model pairs. However, as it is the case with most artificial intelligence approaches, only an approximation of the optimal solution can be found. This may be enough for some scenarios (e.g., searching for model elements in a model repository where the user has to select one of the best matches). For other scenarios (e.g., model exchange between different modeling tools), carefully-engineered model transformations are necessary [BM07]. Such scenarios are not supported by Kessentini et al., because the transformation logic is only implicitly available in the trained optimization algorithm, which is not adaptable.

Finally, a complementary approach for generating model transformations automatically is metamodel matching. Two dedicated approaches [FV07, FHLN08] have been proposed for computing correspondences between metamodels that are input for generating model transformations. We have experimented with technologies for ontology matching by transforming metamodels into ontologies [KKK+06]. However, we have experienced [KKK+07] that in a setting where *(i)* metamodels use different terminology for naming metamodel elements and *(ii)* the structures of metamodels are very heterogeneous, it is sometimes impossible for the matching algorithms to find the correct correspondences. However, we have to mention that a hybrid approach (i.e., combining a matching approach with a by-example approach) seems to be very promising for gaining the benefits of both approaches. We consider this topic as the subject to future work.

4.3 Limitations and Future Work

Although a wide range of endogenous and exogenous transformation scenarios can be addressed by our MTBD approach, there are still some limitations to be addressed in the future. These potential directions for future work can be grouped into four distinct categories. Firstly, we may extend the expressive power of the transformation specifications to also support even more sophisticated transformation scenarios. Secondly, we may improve our approach regarding the degree of automation. Thirdly, we may provide more sophisticated means for maintaining transformation specifications, and, fourthly, especially in the domain of endogenous transformations, we may further elaborate on reasoning about specified composite operations. In the following, we briefly discuss these three directions.

Expressive power of transformation specifications. One limitation of the expressive power of our transformation specifications concerns our iteration mechanism for endogenous model transformations. As already mentioned above, iterations are attached to precondition templates. Although this is an intuitive way of addressing several transformation scenarios, there are still

some scenarios, in which this technique is insufficient. In particular, sometimes it is not only necessary to process all model elements equally by repeating all diff elements that affect these model elements; instead, diff elements that are *independent* from the multiply bound template should be repeated. For instance, in the domain of state machines, when we want to add a state to composite state C1 for each existing state in composite state C2. Furthermore, due to our current realization of iterations, we may only repeat diff elements for input *model elements*. However, there are scenarios in which we might want to iterate over *simple-typed feature values*. For instance, we may intend to create a new transition for each distinct word in another transition's name. As simple-typed values are not represented by a dedicated template, we may not iterate over them. Such scenarios may only be addressed by enabling users to configure iterations directly to the diff elements for each result of an OCL query.

Another issue in our approach to be addressed in the future concerns more complex user-defined selection mechanisms (i.e., queries) for selecting model elements to be transformed. Currently, our template binding mechanism does not support, for instance in Ecore models, to collect recursively all structural features of all direct or indirect superclasses and process them in some way. Model elements may only be matched according to the containment structure of the example source model. Consequently, we may not collect model elements across the whole input model irrespective of their structure; for instance, whether these model elements are contained by the superclass or super superclass (and so on) of an input class. Therefore, we suggest to introduce a special type of template, which we call *selector templates*. Such templates are bound to the model elements that are returned by a user-defined OCL query in the context of model elements that are already bound to usual templates. This, however, has not been implemented yet in our MTBD tool.

Another potential limitation of our MTBD approach for exogenous transformations, concerns the operators and matching strategies. We support four operators that may be used to annotate templates; namely, *reuseObject*, *MapTypes*, as well as two different matching strategies *Set* and *Sequence*. These operators, however, address only the most common exogenous transformation scenarios. Therefore, further scenarios have to be considered to evaluate whether further operators or annotations are required.

Another interesting direction for future work in the area of endogenous transformation specification concerns the composition of existing operation specifications. Currently, a composite operation only exists on its own having no relationships to other composite operations. In practice, however, it would be beneficial to allow users to compose larger composite operations from existing composite operations; either by combining them or by putting them into a sequential chain.

Automation the operation specification process. Besides elaborating on the expressive power of our transformation specifications, there is also potential for improvement left regarding the automation of the specification process. Currently, several configuration steps have to be manually applied by the user. However, by employing more sophisticated heuristics (e.g., from the domain of ontology matching [ES07, RB01]), we may automate some configurations or, at least, compute recommendations for configurations. For instance, if in the source example model, a value of a feature is "name" and, in the target example model, there is the value "getName",

more sophisticated string matching techniques might allow for automatically detecting an adequate computation to get the string "getName" from the input value "name" (e.g., `"get" + value.firstToUpper()`, whereas `value` refers to "name").

Several configurations (e.g., iterations and user inputs for endogenous transformations, or operators and mapping strategies for exogenous transformations) could be derived automatically, if the user provides more than one demonstration. However, it is doubtful whether the specification of multiple demonstrations is more efficient from the user's point of view than having user-friendly ways for fine-tuning the transformation specification derived from only one demonstration. One way of tackling the tedious specification of multiple demonstrations is to purposefully generate an alternative source model from the original source model and apply the current transformation specification to the alternative source model. Subsequently, we may ask the user whether the result is correct for the alternative source model. If not, the user may correct the result, and based on these corrections, we could try to derive reconfigurations automatically for the current transformation specification. For such derivations of configurations, we may reuse several existing techniques from machine learning (e.g., [DMDH04]) or evolutionary computational techniques (e.g., [KSB10]). Another way for obtaining multiple demonstrations would be to observe the user's actions continuously after a specific transformation has been applied. If these actions mostly concern similar corrections or reworks, we may also derive specific reconfigurations. Especially for endogenous transformation, this is also somehow related to the research direction of *LiveMTBD* by Sun et al. [SGW+11b]. Similar to the techniques for automating the configurations of transformation specifications, we may test the generated transformation specification automatically. In particular, by applying the inferred transformation specification to the source model, the obtained target model may be compared to the example target model created during the specification process. If any differences are found in the comparison, either the transformation specification or the original target model is obviously wrong. In this sense, MTBE and MTBD inherently implement the idea of test-driven development [Bec03]. To summarize, an interesting challenge to be addressed in future work is to suggest corrections automatically to the transformation specifications based on the detected differences between the example target model and the actual result after the transformation has been applied.

Maintenance of transformation specifications. Our implementation of the presented MTBD approaches currently have some restrictions regarding the maintenance of a transformation specification after the demonstration has been performed. After testing the transformation specifications, users might experience errors in their specifications, which they aim to fix. However, fixing and improving existing transformation specification is currently hampered by the following limitations.

On the one hand, it is currently impossible to modify the actions that have been automatically derived from the user's demonstration. However, users may have made mistakes during the demonstration or they recognize that they want to replace the action for removing and adding a model element with a corresponding move of a model element. On the other hand, the derived pre- and postconditions may only be fine-tuned in terms of adding, removing, or modifying them; it is not possible to merge two equivalently modified model elements in the example model, which are, as a consequence, represented by two distinct templates, into one template.

To this end, an interesting research direction is to elaborate means for testing, maintaining, and refactoring transformation specifications. Although this topic, in general, is not particularly tied to MTBD, we believe that the idea behind demonstration-based approaches might also be applicable to those tasks in the domain of model transformations.

Reasoning about transformation specifications. Orthogonal to the future research directions mentioned above, another potential topic for future work is reasoning about transformation specifications. Thereby, we especially refer to analysing endogenous transformation specifications as, for instance, inferring whether two composite operations can be composed into one "composite composite operation", whether they can be put into a sequential chain, or whether they are interfering with each other. For such analysis, it would be beneficial to translate operation specifications into a representation based on a theoretical foundation such as graph transformation systems. Besides others, this would allow for reusing theoretically well-grounded techniques. For instance, the critical pair analysis has been successfully applied in [MTR05]. A first work regarding the translation of operation specifications into graph transformation systems has been accomplished in the MSc thesis by Gabmeyer [Gab11].

Listing 4.3: Generated ATL Code

```
1  module CD2ER;
2  create OUT : ER from IN : CD;
3
4  rule GenerateERModel {
5    from cdmodel : CD!SimpleCDModel
6    to ermodel : ER!ERModel (
7      entities <- cdmodel.classes,
8      attributes <- cdmodel.classes ->
9        collect(e| e.properties) -> flatten(),
10     relationships <- cdmodel.classes -> collect(x| x.references) ->
11       flatten() -> collect(x| thisModule.GenerateRelationship(
12       x, x.opposite)))
13 }
14 ------------------------------------------------------------------------
15 rule GenerateEntity {
16   from class : CD!Class
17   to entity : ER!Entity (
18     name <- class.name,
19     attributes <- class.properties)
20 }
21 ------------------------------------------------------------------------
22 helper def : mapTypes(x : CD!Types) : ER!Types =
23     Map{(#String,'varchar'), ...)}.get(x);
24
25 helper def : seenERTypes : Set(ER!Type) = Set{};
26
27 rule GenerateAttribute {
28   from property : CD!Property
29   to attribute : ER!Attribute (
30     name <- property.name,
31     type <-
32       if(thisModule.seenERTypes -> exists(e|
33                 e.name = thisModule.mapTypes(property.type)))
34       then thisModule.seenERTypes -> any(e|
35                 e.name = thisModule.mapTypes(property.type))
36       else thisModule.CreateType(property.type) endif)
37 }
38
39 lazy rule CreateType {
40   from cdType : CD!Types
41   to erType : ER!Type (
42     name <- thisModule.mapTypes(cdType))
43   do{thisModule.seenERTypes <- thisModule.seenERTypes ->
44         including(erType);}
45 }
46 ------------------------------------------------------------------------
47 unique lazy rule GenerateRelationship {
48   from reference1 : CD!Reference,
49      reference2 : CD!Reference (reference1.opposite = reference2)
50   to relationship1 : ER!Relationship (
51       roles <- Set{role1, role2}),
52     role1 : ER!Role(
53       name <- reference1.name,
54       refersTo <- reference1.target,
55       cardinality <- cardinality1),
56     role2 : ER!Role(...),
57     cardinality1 : ER!Cardinality(
58       upper <- reference1.upperBound,
59       lower <- reference1.lowerBound,
60     cardinality2 : ER!Cardinality(...)
61 }
```

CHAPTER 5

Operation Detection

Having introduced our MTBD approach called EMO for specifying composite operations in the previous chapter, which is a prerequisite for a composite operation-aware model versioning system, we now proceed with presenting the steps of the merge process (cf. Figure 1.2 on page 8). In this chapter, we discuss our approach for detecting operations that have been applied between two versions of a model.

The operation detection consists of four steps, which are depicted in Figure 5.1. In particular, the first step is the *UUID-based Matching*, which takes one origin model V_o and two modifications of it, V_{r1} and V_{r2}, as input and computes two distinct match models, $M_{V_o,V_{r1}}$ and $M_{V_o,V_{r2}}$. To overcome the drawbacks of UUID-based matching, we subsequently improve these two match models by additionally applying language-specific match rules to those model elements that could not be matched based on their UUIDs, which is referred to as *Rule-based Matching* in Figure 5.1. These two matching techniques are discussed in Section 5.1. As match models only indicate which model element of one model version has a corresponding model element in another model version, we further have to reveal the actual operations that have been applied between V_o and V_{r1}, as well as V_o and V_{r2}, before we may proceed with detecting

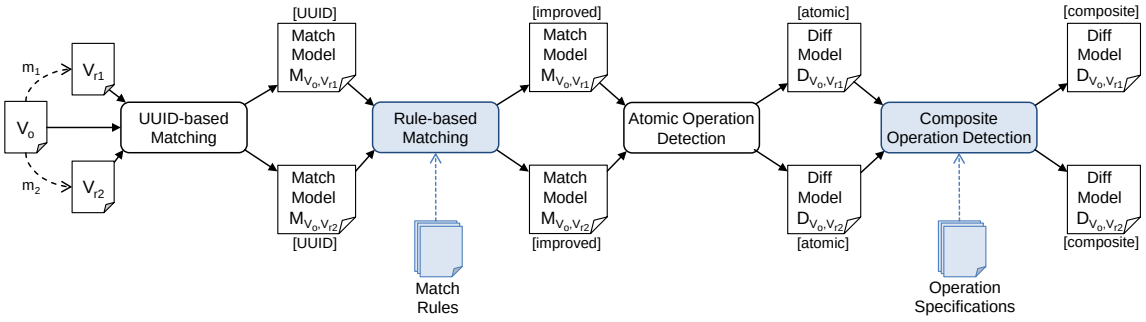


Figure 5.1: Operation Detection Steps in the Adaptable Merge Process

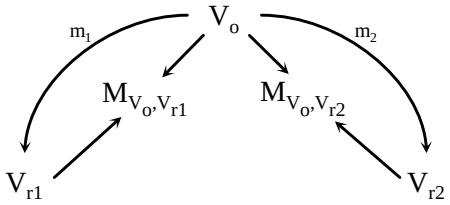


Figure 5.2: Model Versions and Match Models

conflicts among them. Therefore, the next step is the *Atomic Operation Detection*, which reconstructs all applied atomic operations based on the input match models and saves the detected operations into two distinct diff models, $D_{V_o,V_{r1}}$ [atomic] and $D_{V_o,V_{r2}}$ [atomic]. This step, which is agnostic to the respective modeling language of the input models and which does not incorporates additional language-specific knowledge, is discussed in Section 5.2. Subsequently, in the next step, namely the *composite operation detection*, we aim at also detecting applications of language-specific composite operations; therefore, we certainly require the operation specifications that have been specified using EMO (cf. Section 4.1). Our approach for reconstructing applications of these operation specifications between two model versions is presented in Section 5.3. All applications of composite operations that have been detected in this step are saved to the diff models, $D_{V_o,V_{r1}}$ [composite] and $D_{V_o,V_{r2}}$ [composite], and handed over to the succeeding conflict and warning detection steps (cf. Chapter 6).

5.1 Model Matching

The first step in our operation detection process is model matching—as in most of the existing state-based model comparison approaches [BP08, CW98, Men02]. The goal of matching two models is to produce a mapping of each model element in the original model V_o to the corresponding model element in the revised versions, V_{r1} and V_{r2}. The relationships among match models and the model versions are illustrated in Figure 5.2. In particular, the original version V_o is separately matched first with V_{r1} and, subsequently, with V_{r2}. This leads to two distinct match models $M_{V_o,V_{r1}}$ and $M_{V_o,V_{r2}}$. Obviously, $M_{V_o,V_{r1}}$ describes the correspondences between V_o and V_{r1}, and the match model $M_{V_o,V_{r2}}$ contains the correspondences between V_o and V_{r2}.

Corresponding model elements do not necessarily have to be equal entirely; they might differ regarding some attribute or reference values. For determining whether two model elements correspond to each other, a match function is required. How this match function is implemented strongly varies among different model matching approaches. Therefore, we first review the state of the art in the domain of model matching in Section 5.1.1 and highlight the diverse ways to realize such match functions. Subsequently, we discuss the match functions that are applied in AMOR for computing these match models and justify why we choose to apply these functions in Section 5.1.2. Once the correspondences have been determined, regardless of which match function has been applied, the correspondences have to be represented to be processed in succeeding steps. To this end, we introduce a dedicated metamodel to express these correspondences in terms of a match model in Section 5.1.3.

5.1.1 State of the Art in Model Matching

The problem of matching two model elements is to find the *identity* of the model elements to be matched. Once the identity is explicit, model elements with equal identities may be matched. Thereby, the identity is computed by a match function. The characteristics of a model element that are incorporated to compute the identity, however, varies among approaches, scenarios, and objectives of a match function.

The predecessors of solutions for model matching are approaches to program element matching in the research domain of multi-version program analysis [KN06], as well as schema matching in the data base research domain [RB01], and ontology matching in the knowledge representation research domain [WVV+01]. Therefore, we first highlight some remarkable categorizations of matching techniques from these three research domains and, subsequently, proceed with surveying recent approaches to model matching.

Program element matching. Finding correspondences among program elements, that is, entities within the specification of software, is a fundamental building block of multi-version program analysis and other software evolution research such as profile propagation, regression testing, and software version merging [KN06]. Kim and Notkin [KN06] classified existing approaches to this problem according to the matching technique they apply. In the following, we outline some of the identified matching techniques. *Entity name matching* is perhaps the simplest technique, which treats program elements as immutable entities with a fixed name and matches these elements by name (e.g., file name, function name). *String matching techniques* are applied by approaches that compare programs by solely comparing their textual representation (i.e., source code) as discussed in Section 2.1. As also already discussed in Section 2.1, program elements may be matched by analysing their graph-based representation in terms of the *abstract syntax tree* instead of their textual representation. There are also approaches that focus on matching the *control flow graph* of programs instead of their static structure. Interestingly, as stated by Kim and Notkin, another technique exists, which is denoted as *origin analysis*. This technique establishes matches by tracing back to a program entity's origin; thereby also information on refactoring applications, such as splits, merges, renames, and moves, are incorporated.

Schema matching and ontology matching. The problem of matching database schema gained much attention among researchers for addressing various research topics such as schema integration, data extraction for data warehouses and e-commerce, as well as semantic query processing. To reconcile the structure and terminology used in the emerged approaches from these research topics, Rahm and Bernstein [RB01] proposed a remarkable classification of existing approaches. On the most upper layer, Rahm and Bernstein distinguish between *individual matcher approaches* and *combining matchers*. Individual matchers are further classified according to the following largely orthogonal criteria. First of all, they consider whether matching approaches also incorporate instance data (i.e., data contents) or only the schema for deriving correspondences among schema elements. Further, they distinguish between approaches that perform the match only on single schema elements (i.e., they operate on *element level*) or on combinations of multiple elements to also regard complex schema structures (i.e., *structure level*). Another distinction is made upon approaches that uses either *linguistic-based matching* (e.g.,

based on names or descriptions) or *constraint-based matching* (e.g., unique key properties or data types). Matching approaches may also be characterized according to the match cardinality; that is, whether they return one-to-one correspondences or also one-to-n or even n-to-m correspondences. Finally, there are approaches that not only take a schema as input, but also exploit auxiliary information (e.g., dictionaries, global schemata, previous matching decisions, or user input). On the other side, among combining matchers, Rahm and Bernstein identified *hybrid matchers* that directly combine several matching approaches to determine match candidates based on multiple criteria or information sources. They also identified *composite matchers* that combine the results of several *independently executed* matchers. The composition of matchers is either done automatically or manually by the user.

With the rise of the semantic web [BLH01], the problem of integrating, aligning, and synchronizing different *ontologies* into one reconciled knowledge representation induced an active research area. Therefore, several ontology matching approaches have been proposed (cf. [WVV$^+$01] for a survey). As argued by Shvaiko and Euzenat [SE05], schema matching and ontology matching are largely the same problem because schemata and ontologies both provide a vocabulary of terms that describes a domain of interest and both constrain the meaning of terms used in the vocabulary [SE05]. Consequently, they differ regarding the input, that is, how the schemata or ontologies to be matched are expressed. The same is also true for conceptual models in general, as well as for XML Schema [W3C09] and XML documents. The survey by Rahm and Bernstein [RB01] focuses on schema matching approaches, whereas the survey by Wache et al. [WVV$^+$01] concentrates on ontology matching. Thus, Shvaiko and Euzenat extended the previously mentioned categorization of matching approaches in [SE05] in order to addresses both schema and ontology matching approaches explicitly. In another dimension, Shvaiko and Euzenat added a dimension concerning *syntactic*, *external*, and *semantic* matchers. Syntactic matchers operate only on the input schemata/ontologies following some clearly stated algorithm. In contrast, external matchers also incorporate external resources of a domain and common knowledge in order to interpret the input schemata/ontologies such as lexicons, existing mappings, upper level ontologies, repositories of structures. Finally, semantic techniques exploit formal semantics (e.g., model-theoretic semantics) to interpret the input and justify their results. Furthermore, they distinguish between *exact* matchers, which they guarantee a discovery of all the possible mappings and *approximate* algorithms, which tend to be incomplete and only provide one or more possible mappings.

Model matching. The aforementioned categorizations and terminologies also can be used for characterizing model matching approaches. However, the distinction between schema-only and instance-based approaches only applies to approaches specifically tailored to match *metamodels*, because models on the $M1$ level in the metamodeling stack (cf. Section 3.4.2) have no instances to be used for matching. Furthermore, in the context of model matching, the only constraint-based similarity measure that can be used across all meta levels is the type information (i.e., the respective metaclass) of a model element. Besides applying the categorization coming from schema and ontology matching, Kolovos et al. [KDRPP09] further proposed a categorization specifically dedicated to model matching approaches. In particular, they distinguish between static identity-based matching, signature-based matching, similarity-based matching, and cus-

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

In the following, we discuss existing approaches in the domain of model matching. Obviously, many existing *model versioning* approaches also address the topic of model matching. These approaches, however, have already been discussed in Section 2.1.2. Therefore, in the following, we only discuss the model matching approaches that have not been considered yet and highlight only those approaches for model versioning that offer interesting matching capabilities.

One of the first model matching approaches has been proposed alongside their model comparison algorithm by Alanen and Porres [Por05]. Although their approach only supports UML models and, thereby, they easily could have incorporated language-specific match rules, the proposed match function relies on static identifiers only. Also, specifically tailored for a specific modeling language is *UMLDiff* [XS05], which is, however, not based on static identifiers. Instead, *UMLDiff* computes similarity metrics based on a model element's name and structure. In terms of the aforementioned categorizations, UMLDiff applies string-based matching at the element level as well as graph-based matching at the structure level and internally combines the obtained similarity measures; thus, UMLDiff is a hybrid matching approach. The same is true for the approach by Nejati et al. [NSC+07], which is specifically tailored for matching UML state machines. Their matching approach uses static similarity measures such as typographic, linguistic, and depth properties of model elements, but also behavioural similarity measures. Also specifically tailored to UML models is *ADAMS* [DLFST09], which uses a hybrid matcher that first applies a static identity-based matcher and matches all remaining (not matched) model elements using a simple static signature-based approach based on model element names. In contrast to language-specific matching approaches, also several generic approaches have been proposed such as *DSMDiff* [LGJ07] and *EMF Compare* [BP08]. DSMDiff first compares elements based on a computed signature (incorporating the element name and type) and, subsequently, considers the relationship among model elements previously matched by signature. Largely similar to DSMDiff, EMF Compare computes four different metrics and combines them to obtain a final similarity measure. In particular, EMF Compare regards the name of an element, its content, its type and the relations to other elements. EMF Compare also offers a static identity-based comparison mode, which works similarly to the approach by Alanen and Porres [Por05]. However, EMF Compare only allows for either similarity-based or static-identity based matching. Both strategies cannot be combined (cf. Section 3.4.3 for more information on EMF Compare). The similarity-based matching approach applied in EMF Compare heavily exploits the tree-

based containment structure when comparing models. Rivera and Vallecillo [RV08] argue that this leads to issues concerning the detection of, for instance, elements that have been moved to new container elements. Therefore, Rivera and Vallecillo [RV08] propose to compare model elements independently of their depth in the containment tree. Besides this difference, the exploited information on model elements for matching is largely similar to DSMDiff and EMF Compare. DSMDiff and EMF Compare aim at obtaining an optimal result, whereas no language-specific information or configuration is necessary; in contrast, the goal of *SiDiff* [SG08] is to provide an adaptable model comparison framework, which may be fine-tuned for specific modeling languages by configuring the actual characteristics of model elements to be considered in the comparison process and attaching weights to these characteristics. DSMDiff, EMF Compare, and SiDiff are hybrid matching approaches. On the contrary, Barret et al. recently presented *Mirador* [BBC10], which is a composite matching approach. That is, several matching strategies are independently applied and presented in a consolidated view of all match results. Using this view, users may interactively refine the computed match by attaching weights and manually discarding or adding matches. Thereby, the goal is to offer a wide assortment of model comparison algorithms and matching strategies under control of the user. Yet another approach is taken by Kolovos with the *Epsilon Comparison Language* (ECL) [Kol09] (cf. Section 3.4.4 for more information on ECL). Instead of providing a set of predefined and potentially configurable matching strategies, ECL is a hybrid rule-based language, which enables users to implement comparison algorithms at a high level of abstraction and execute them for identifying matches. Although it indeed requires some dedicated knowledge to create language-specific match rules with ECL, it facilitates highly specialized and sophisticated matching algorithms, which may also incorporate external knowledge such as lexicons, and thesauri.

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

5.1.2 Combining UUID-based and Rule-based Matching

Although UUID-based matching is probably the most efficient and straightforward technique for model matching, there are mainly four issues mitigating its advantages. First, the prerequisite for UUID-based matching is obviously that the editor attaches a UUID to each created model element and does not change it through the whole lifecycle of the model element. Although XMI [OMG07], OMG's standard for serializing models, foresees IDs by a dedicated attribute, the use of IDs is still optional. Consequently, we may not presume that every modeling editor attaches IDs to each model element. Second, with UUID-based matching, we may only match a model with one of its predecessors. Two independently created models may obviously not be matched based on their UUIDs. The third issue of UUID-based matching appears when a user

deletes a model element and adds a new model element having largely the same characteristics as the previously deleted one. Reasons for that might be an improper implementation of the copy & paste feature in the used modeling editor (as it is sometimes the case with EMF editors) so that a pasted model element looses its original UUID or users might decide that it is more efficient to delete and re-create a new model element at a different location in a model rather than moving the original model element to a new location. In both cases, the new model element, which in fact should be matched with the original model element, cannot be matched based on UUIDs. Finally, another shortcoming of UUID-based matching is that it is possible to match two concurrently added model elements having entirely or at least largely the same characteristics.

As also stated by Kolovos et al. [KDRPP09], selecting the right matching approach involves deciding on a trade-off between the required accuracy and the effort necessary to accomplish the differencing. Thus, it is still tempting to take advantage of the efficient and straightforward way of matching model elements by UUID, but users should be enabled to invest more efforts, if required, to improve the matching result and overcome the shortcomings of UUID-based matching. Therefore, we propose a combination of UUID-based matching and language-specific matching. In the following, we first discuss the shortcomings of UUID-based matching and, subsequently, provide dedicated solutions to overcome them. The first issue concerns modeling editors that do not attach a UUID to each created model element. Although we refrain from putting any restrictions on the used modeling editor, we may still easily attach UUIDs independently from the editor for our model versioning scenario. In particular, at the beginning of the lifecycle of each model under version control, it first has to be checked into the versioning system. At this point, the model versioning system can easily attach UUIDs by itself, unless they have already been set by the editor. For persisting these UUIDs, we may use the attribute `XMI:ID`, which is designed to be independent from an annotated modeling language. As this attribute is specified by the OMG to be immutable, it can be presumed that modeling editors will not modify this UUID during the whole lifecycle of a model element. The second shortcoming concerns the inability of matching two independently evolved models. However, for our scenario, we only have to match revised models with their respective origin model; consequently, this issue does not concern us for our requirements. The third issue is that UUID-based approaches are incapable of handling scenarios, in which a model element loses its UUID caused by improperly implemented copy & paste actions, or when the user deletes and adds a new model element with the same characteristics. The same is true regarding the fourth mentioned issue; that is, both users concurrently added an entirely or at least largely equal model element. For such cases, we install a second matching phase, which exploits language-specific match rules as discussed in the following.

To combine the advantages of UUID-based and language-specific matching, we apply a two-phase matching process. First, we perform an proven UUID-based matching to obtain a base match, which is, subsequently, improved by applying language-specific match rules. Thereby, we only try to *re-match* model elements that could not be matched by their UUIDs. Consequently, the applications of the comparatively slow rule-based matching is kept at a minimum. Consider, for instance, the example depicted in Figure 5.3. In this example, the origin version V_o of a state machine contains a composite state Active having the UUID [s1], three single states with the UUIDs [s2], [s3], and [s4], as well as one transition with the UUID [t1]. In the revised

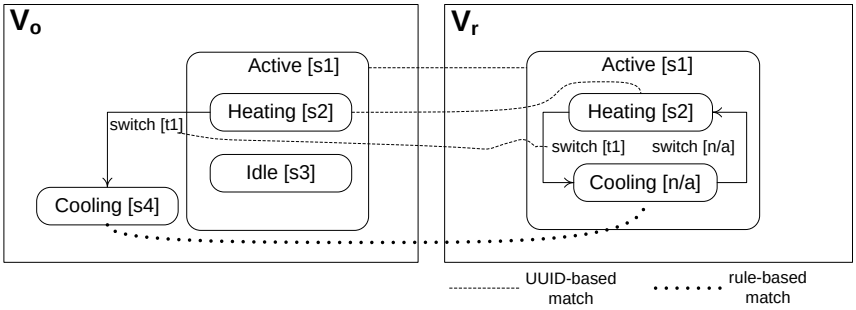


Figure 5.3: Example for Combining UUID-based and Rule-based Matching

Listing 5.1: Excerpt of Match Rules for State Machines

```
1  rule State2State
2    match o : Origin!State
3    with r : Revised!State {
4    compare : o.name = r.name
5  }
6  rule Transition2Transtion
7    match o : Origin!Transition
8    with r : Revised!Transition {
9    compare : o.name = r.name
10 }
```

version, the state Cooling, which originally had the UUID [s4], has been cut and pasted into the composite state Active. Unfortunately, the UUID of Cooling has been lost during this transaction. Additionally, the state Idle [s3] has been deleted and a new transition switch has been added without a UUID. For obtaining the match between these two models, we first compute an initial match based on UUIDs. The detected correspondences are illustrated by the dashed lines. Obviously, three model elements could not be matched based on their UUIDs. In particular, this is the state Cooling, which lost its UUID, the state Idle, which has been deleted, and the transition switch that has been added in the revised version. Next, we try to find additional matches of heretofore unmatched elements based on the rules depicted in Listing 5.1. Thereby, we obtain an additional match for the state Cooling [s4] in the original version and Cooling [n/a] in the revised version as both states have an equal name (cf. rule State2State in Listing 5.1). Thus, the unmatched state Cooling in the revised version does not match with the second unmatched state Idle in the origin version. Please note that although the added transition named switch would match with the equally named transition in V_o according to the match rule Transition2Transition in Listing 5.1, no further match is added because switch [t1] has already been matched based on its UUID and only heretofore unmatched elements are involved in the second matching phase.

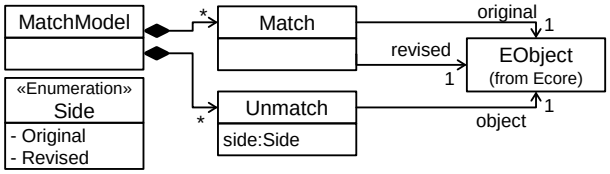


Figure 5.4: Match Metamodel

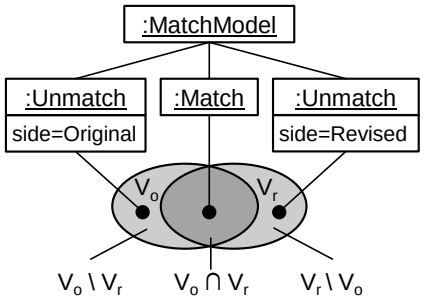


Figure 5.5: Conceptual Representation of a Match Model

5.1.3 Match Metamodel

Having obtained the model element correspondences between the original version V_o and a revised version V_r of a model, they have to be represented in some way for their further usage. Therefore, we introduce the match metamodel depicted in Figure 5.4. Please note that this match metamodel is largely equivalent to the one used in EMF Compare [BP08]. Basically, a match model is a so-called *weaving model* [FBJ+05], which adds additional information to two existing models by introducing new model elements that refer to the model elements in the original and the revised model. In particular, a match model comprises an instance of the class MatchModel, which contains, for each pair of matching model elements, an instance of the class Match. This instance refers to the corresponding model element in the original version through the reference original and the revised version through the reference revised. If a model element, either in the original model and in the revised model, could not be matched, an instance of the class Unmatch is created, which refers to the unmatched model element in the respective model. The attribute side indicates whether the unmatched model element resides in the original or the revised model.

A match model groups the model elements in V_o and V_r into three distinct sets (cf. Figure 5.5). The first set constitutes all model elements that are contained in the original version, but not in the revised version (i.e., $V_o \setminus V_r$). The second set contains all model elements that are contained in both models (i.e., $V_o \cap V_r$) and the third set comprises all model elements that are contained in the revised model but not in the original model (i.e., $V_r \setminus V_o$). In EMF, attribute and reference values of a model element are possessed by the respective model element. Thus, they are considered as being a property of the model element rather than being treated as its own entity. Consequently, in the match model only corresponding *model elements* are linked by Match instances.

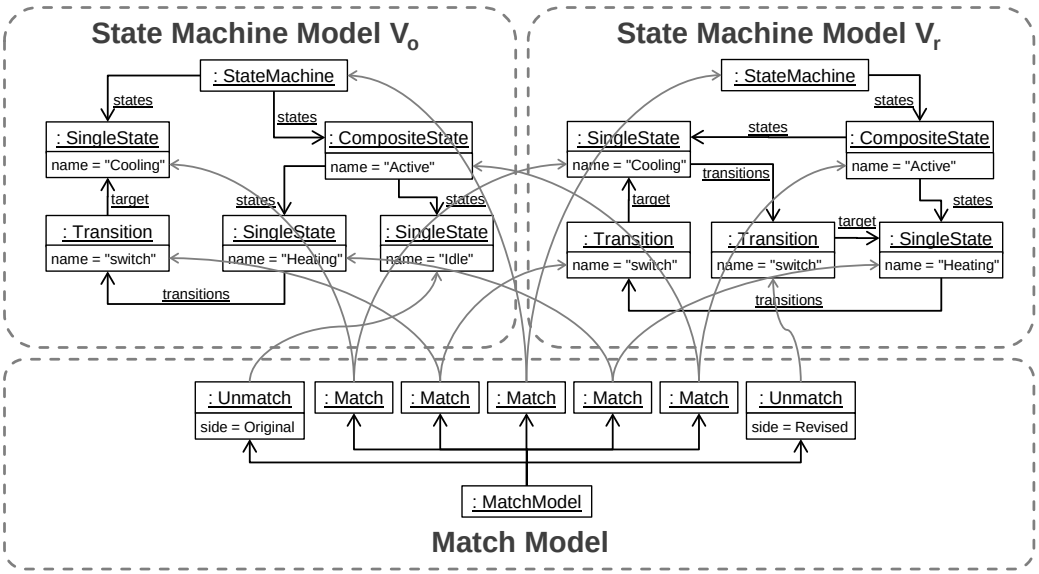


Figure 5.6: Example for a Match Model

To further clarify match models, in Figure 5.6, we depict the two versions of a state machine and the corresponding match model of our matching example (cf. Figure 5.3) in terms of object diagrams. As the state Idle has been deleted and the transition switch going from Cooling to Heating has been added in this example scenario, the corresponding match model contains two instances of Unmatch; one, for the deleted state on the side Original and one for the added transition on the side Revised. All other model elements have a corresponding model element in the opposite model. Thus, the match model further contains five instances of Match; each of these instances links the respective corresponding model elements in V_o and V_r.

5.2 Atomic Operation Detection

Having obtained the correspondences among model elements in an original model V_o and a revised model V_r, we may now proceed with deriving the atomic operations that have been applied by the user to V_o in order to create V_r. As already mentioned in the previous section, match models only indicate the corresponding model elements and those model elements that only exist either in V_o or in V_r. Corresponding model elements, however, might not be entirely equal as their attribute values or reference value might have been modified. Therefore, we further derive a *diff model* from the match model to also represent operations affecting attribute values and reference values before we may search for conflicts among concurrently performed operations. To put the detection of atomic operations in the context of a model versioning scenario, recall that we have two modifications, m_1 and m_2, and one match model comprising the correspondences for each side, $M_{V_o,V_{r1}}$ and $M_{V_o,V_{r2}}$. Therefore, we also have two *diff models* (cf. Figure 5.7). In particular, $D_{V_o,V_{r1}}$, which is computed from $M_{V_o,V_{r1}}$, represents the operations applied in m_1 and $D_{V_o,V_{r2}}$, derived from $M_{V_o,V_{r2}}$, represents the operations applied in m_2.

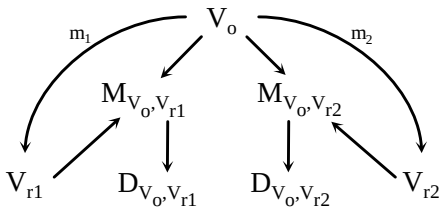


Figure 5.7: Model Versions, Match Models, and Diff Models

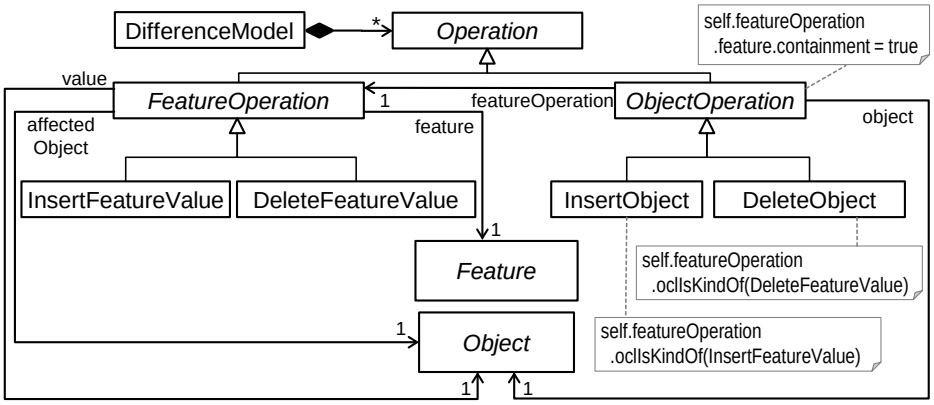


Figure 5.8: Kernel Difference Metamodel

In the remainder of this section, we first introduce *atomic kernel operations* in Section 5.2.1, which can be applied independently of the metamodeling framework. These kernel operations only constitute the basic types of modifications that are applicable to all kinds of graphs or models. For representing all facets of operations that can potentially be applied specifically to *EMF models*, we extend this set of kernel operations and outline how these applications of these operations can be reconstructed from two versions of an EMF model in Section 5.2.2. Finally, we review existing work in the area of detecting and representing applied operations in Section 5.2.3.

5.2.1 Atomic Kernel Operations

Difference models comprise the information that is not explicitly available in a match model, such as changed attribute values, changed reference values. In the following, we first present a *kernel difference metamodel*, which captures only the fundamental information on a model modification without taking the specialities of EMF into account. Therefore, this kernel only contains additions and deletions of model elements and modifications of feature values; EMF-specific facets such as multiplicities and ordered features are omitted in this kernel metamodel for now.

Difference models. To represent the fundamental operation types, the kernel difference model, depicted in Figure 5.8 contains a root class called DifferenceModel, which comprises arbitrarily many instances of Operation. We distinguish between two types of operations: FeatureOperation, which modifies the value of a feature, and ObjectOperation, which represent the insertion or deletion of a model element. Please note that model elements are referred to as *objects* in this metamodel for the sake of generalization.

Feature operations. Objects hold feature values according to the definition of their metaclass. Consequently, values can be inserted or deleted in the course of a modification. For expressing such operations, we use two concrete subclasses of FeatureOperation in the difference metamodel, namely InsertFeatureValue and DeleteFeatureValue. Feature operations refer to the object that has been changed using the reference affectedObject, to the affected feature in the modeling language's metamodel (reference affectedFeature), and to the inserted or deleted feature value (reference value). In case of a reference, this value is a model element and in case of an attribute, the value is a simple data type such as String, or Boolean, etc. However, we omitted to distinguish explicitly between model elements and simply typed data values in Figure 5.8 for the sake of readability. It is worth noting that, in case of a InsertFeatureValue, the reference value refers to the inserted value in the revised model (V_{r1} or V_{r2}) and, in case of a DeleteFeatureValue, it refers to the deleted value in the original model V_o.

Object operations. Besides inserting and deleting feature values to existing objects, users may also insert and delete entire objects (i.e., model elements). Therefore, the metamodel contains the two classes InsertObject and DeleteObject, which are subclasses of ObjectOperation. Except for root objects, objects are always contained by another object through a *containment feature*. Consequently, inserting and removing an object is realized by a feature operation affecting the respective containment feature. Thus, object operations are further specified by a reference to the respective instance of a FeatureOperation, which gives information on the inserted or deleted object (reference value), the container of the inserted or removed object (reference affectedObject), and the containment feature through which the object is or originally was contained (reference affectedFeature). Certainly, as defined by the invariants in Figure 5.8, a valid instance of InsertObject must refer to an instance of InsertFeatureValue and a valid instance of DeleteObject must refer to an instance of DeleteFeatureValue, whereas the affected feature has to be a containment feature. To avoid the lengthy navigation through the referenced FeatureOperation, instances of ObjectOperation contain a reference called *object*, which directly refers to the inserted or deleted object.

5.2.2 Atomic EMF Operations

When recalling the Ecore metamodel (cf. Figure 3.9 on page 69), it is clear that the kernel difference model in Figure 5.8 does not cover the complete picture of all potentially applicable operations to EMF models. Several aspects, such as ordered features and multiplicities supported by the Ecore metamodel, are not represented in the kernel difference metamodel. Therefore, we

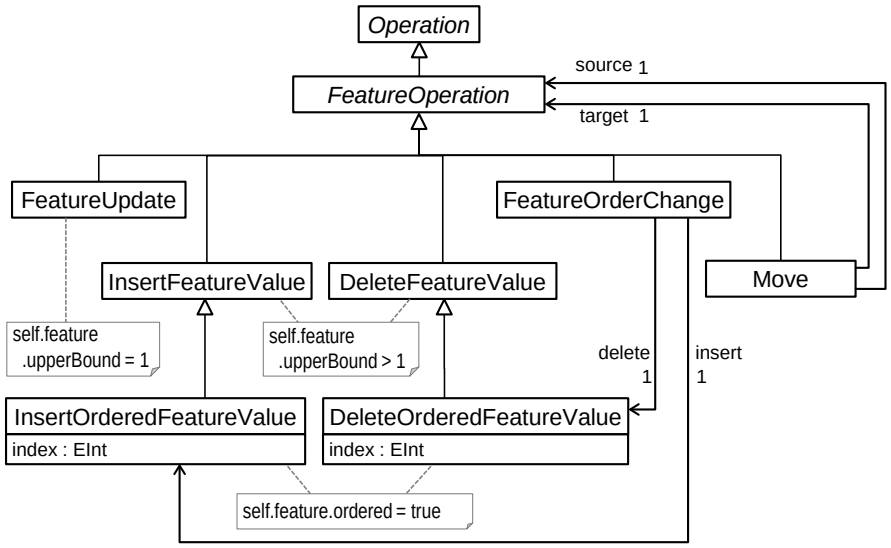


Figure 5.9: Extension of the Difference Metamodel for EMF-based Models

extend the kernel difference metamodel in order to allow for expressing all facets of operations in the context of EMF-based models. The extensions of the kernel metamodel is depicted in Figure 5.9.

Single-valued features. In the kernel difference metamodel, the multiplicity of features is not explicitly represented. However, when merging, it makes a difference whether a feature is single-valued (upperBound is equal to one) or multi-valued (upperBound is greater than one). Changing a single-valued feature value always overwrites the old value and, consequently, if a single-valued feature is changed on both sides in a versioning scenario, a conflict always has to be reported. This is not the case with multi-valued features. Hence, we introduce FeatureUpdate, which represents the operations applied to a single-valued attribute or reference in addition to InsertFeatureValue and DeleteFeatureValue for multi-valued features.

Ordered features. The Ecore metamodel allows to define *ordered* features. Ordered features pose an additional challenge when merging two versions of a model because the resulting order of feature values has to be regarded. If a feature is ordered, each model element in the set of values has an index. In the extended difference metamodel, this is reflected by the classes InsertOrderedFeatureValue and DeleteOrderedFeatureValue. Besides inserting and deleting values from ordered feature values, users may also modify *only the order* of feature values, whereas the set of values remain the same. Such an operation is realized by one instance of DeleteOrderedFeatureValue for detaching the object from its original index and one instance of InsertOrderedFeatureValue for inserting the same object at its new index again. In order to make such operations more explicit, we additionally introduce the class FeatureOrderChange in the extended metamodel (cf. Figure 5.9), which refers to the aforementioned instance of DeleteOrderedFeatureValue and InsertOrderedFeatureValue realizing the order change.

Moving model elements. If two feature operations affect *containment references* and insert and delete the *same object*, we can infer that this object is *moved* from one container to another. Thus, a Move is a derived operation consisting of two feature operations. That is, either one InsertFeatureValue and one DeleteFeatureValue, if both containment features are multi-valued. If only the source containment feature is multi-valued, the Move consists of one DeleteFeatureValue and one FeatureUpdate. If, however, only the target containment feature is multi-valued, it consists of one FeatureUpdate and one InsertFeatureValue. Finally, if the source and target containment features are single-valued, the Move is derived from two FeatureUpdates. The old container of the moved object is indicated by the affectedObject reference in the source FeatureChange and the new container is indicated by affectedObject of the target FeatureChange (cf. Figure 5.9).

To exemplify the difference metamodel, a concrete instance is depicted in Figure 5.10, which represents the differences of our example (cf. Figure 5.3). Please note that we depcited the links between the objects of the state machine versions in gray for the sake of readability. This figure depicts the two versions of the state machine, V_o and V_r, the difference model D_{V_o,V_r} expressing the differences between these two state machines, and an excerpt of the metamodel for state machines. Between the origin model and the revised model, the user deleted the state Idle, moved the state Cooling from the state machine root into the composite state Active, and inserted the transition switch to the transitions of Cooling. Consequently, the difference model contains an instance of DeleteObject, another instance of Move, as well as an instance of InsertObject. These three instances refer to the respective instances of InsertFeatureValue and DeleteFeatureValue that actually realize the object insertion, object move, and object deletion. These instances again refer to the inserted or deleted value (reference value), the affected object, in which the respective value has been inserted or deleted (reference affectedObject), as well as to the feature of the metamodel for state machines (i.e., the EReferences named states and transitions).

Detecting applied operations. After having defined all types of operations, we may now discuss how the operations that have been applied between an original model and a revision of it are detected. As already mentioned, the atomic operation detection step takes a previously computed match model as input. Starting from a match model, the detection of applied operations is largely straight-forward. In particular, the atomic operation detection component first iterates through all Match instances of this match model and performs a fine-grained feature-wise comparison of the two corresponding model elements. Thereby, the feature values of each feature of both corresponding model elements are checked for equality. If a feature value of one model element differs from the respective value of the other model element, an instance of the corresponding operation type from the difference metamodel is created, accordingly linked to the model elements, and, finally, added to the difference model. Subsequently, for all Unmatch instances, depending on its value at the attribute side, an InsertObject or DeleteObject instance is created.

In AMOR, the detection of atomic operations is realized using EMF Compare. In particular, we only apply the differencing component of EMF Compare to obtain the differences based

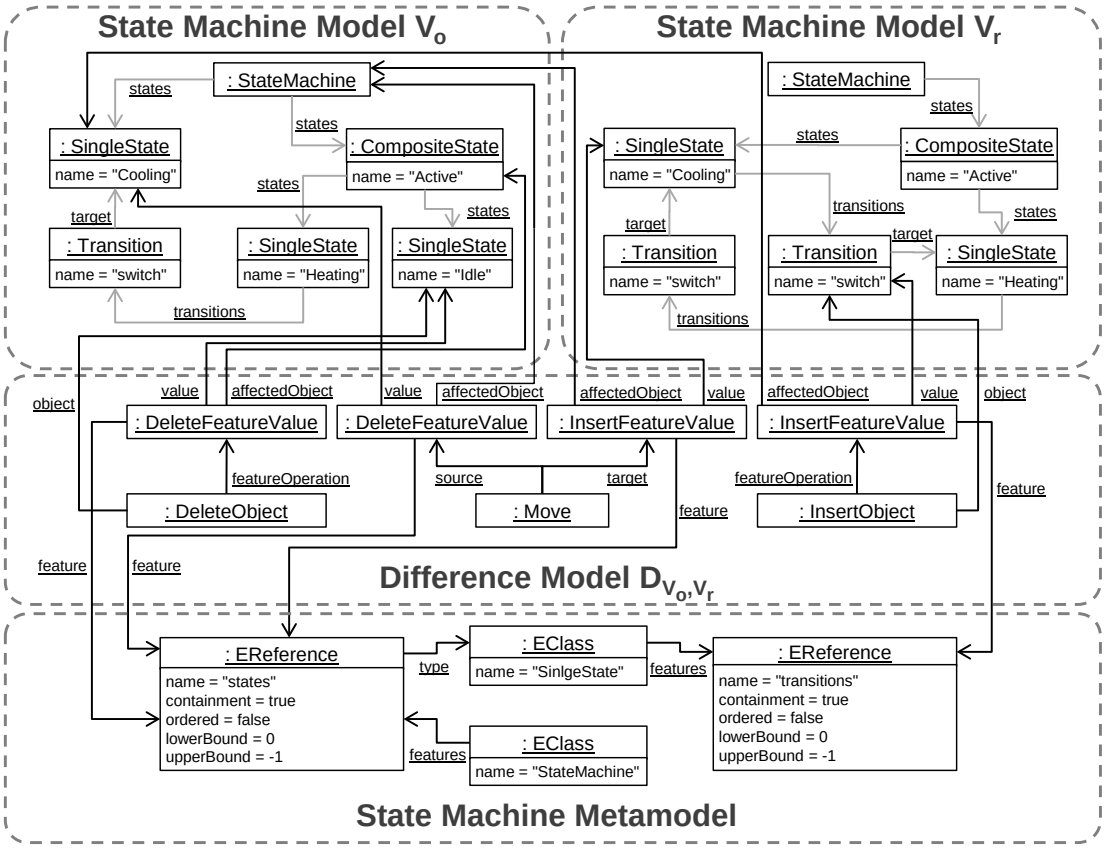


Figure 5.10: Example for a Difference Model

on the match model obtained from our own implementation of the two-phased model matching approach (cf. Section 5.1.2). These differences are then optimized and translated into our model-based representation as depicted in Figure 5.10. The difference metamodel in EMF Compare is very similar to our extended difference model. In our extended difference metamodel additional information is explicitly represented in order to to allow for an efficient analysis when detecting conflicts.

5.2.3 Related Work

Existing work in the area of model differencing can be distinguished regarding their approach to matching model elements across two versions of a model. We considered these different approaches for establishing model element correspondences already in Section 5.1.1. Existing algorithms for computing the actual differences based on such model correspondences are largely similar from a conceptual point of view. That is, the algorithm performs a fine-grained comparison of two model elements that correspond to each other (as indicated by the applied match function). If two corresponding model elements differ in some way, a description of the differ-

ence is created and saved to some kind of diff model. If a model element has no corresponding model element on the opposite side, an element insertion or deletion is noted. However, existing approaches diverge in terms of how precisely the detected operations are represented. Thus, in the following, we focus on this aspect when surveying the related work.

For assessing different approaches for representing differences between two versions of a model, Cicchetti et al. [CDRP07] identified a number of properties a representation of operations should fulfill. Most importantly, they mention the properties indicating whether a representation is *model-based* (i.e., conforming to a dedicated difference metamodel), *transformative* (i.e., applicable to the compared models), and *metamodel independent* (i.e., agnostic of the metamodel the compared models conform to). Besides these properties, in our context, it is important how explicit the detected operations are represented, or whether important information (such as the index at which a value has been added to an ordered feature) is hidden in the context of a detected operation's representation.

In several research papers addressing the topic of model differences, such as [DLFST09, BBC10, MGH05], it is not explicitly mentioned how the detected differences are represented. Many others at least define the types of differences they aim to detect. For instance, *DSM-Diff* [LGJ07] marks model elements to be *added*, *deleted*, or *changed*. Alanen & Porres [Por05] explicitly represent, besides added and deleted model elements, *updates* of single-valued features, *insertions* and *deletions* of values in multi-valued features as well as *ordered* features. *SiDiff* [SG08] distinguishes among *structural differences*, *attribute differences*, *reference differences*, and *move differences*. Several language-specific approaches, in particular, Gerth et al. [GKLE10], *UMLDiff* [XS05], and Ohst et al. [OWK03], introduce operations that are tailored to the specific modeling language they support; thus, they use a metamodel dependent representation of applied operations. For instance, Gerth et al. defines the operations, such as *move activity*, *delete fragment*, etc., for state machines and UMLDiff presents a fine-grained definition of UML class diagram operations such as *new inheritance relationship* or *return type change*. Interestingly, Ohst et al. represent, besides intra-node and inter-node differences, also modifications of the UML class diagram *layout*.

All of the approaches mentioned above do not represent the detected differences in terms of a model that conforms to a dedicated difference metamodel; at least, it is not explicitly mentioned in their research papers. Nevertheless, the difference representations by Alanen & Porres and Gerth et al. are transformative; that is, detected differences can be applied to the compared models in order to create a merged version.

To the best of our knowledge, the only approaches that use a model-based representation of differences are EMF Compare [BP08], Herrmannsdoerfer & Koegel [HK10], and Cicchetti et al. [CDRP07]. All of these approaches are metamodel independent, whereas EMF Compare and Herrmannsdoerfer & Koegel use a generic metamodel and Cicchetti et al. propose to *generate* automatically a dedicated difference metamodel for specific modeling languages. Thereby, in the approach by Cicchetti et al., a dedicated metaclass for indicating the insertion, deletion, and changes for every metaclass in the respective modeling language's metamodel is generated. For instance, for UML class diagrams, difference metaclasses such as AddedClass and ChangedAttribute are generated, whereas Class and Attribute are metaclasses in the modeling language's metamodel. In contrast, EMF Compare and Herrmannsdoerfer & Koegel make use of the reflec-

tive power of EMF and refer to the modeling language's metaclasses to indicate, for instance, a modification to a specific feature of a model element. EMF Compare, as we do in our difference metamodel, refers to the affected model element by a generic reference to EObject, which is the abstract type of all objects within EMF. In contrast, Herrmannsdoerfer & Koegel foresee a more flexible model referencing technique to also enable persistent ID-based and arbitrarily other ways to express a reference to the affected model elements.

To summarize, several model differencing approaches and representations of applied operations have been proposed during the past years. As we use EMF Compare for obtaining the atomic operations, our model differencing approach is not new. Also for representing operations, we draw inspiration from existing definitions and operation/difference metamodels. Although these existing metamodels (e.g., from EMF Compare [BP08] or Hermannsdoerfer & Koegel [HK10]) are technically sound, we feel that our metamodel is more concise and, consequently, is more appropriate for presenting our proposed solutions for detecting composite operations and operation-based conflicts. Nevertheless, it is easy to translate our representation of applied operations to other metamodels, and vice versa, for achieving interoperability as they conceptually correspond to each other.

5.3 Composite Operation Detection

As illustrated in the versioning scenarios in Section 3.1.4 and Section 3.1.5, the knowledge on applications of composite operations between two versions of a model significantly helps in many scenarios to better respect the original intention of a developer, as well as to reveal additional issues when merging two concurrent modifications. The prerequisites for considering applications of composite operations is to detect them among a set of applied atomic operations.

Computing the information on applied composite operations is a very challenging task. One way to acquire the set of applied composite operations is to use *operation recording* [HK10, LvO92], that is, tracking the execution of operations directly within the modeling environment as they are performed by the user. Although this leads to a precise operation log [Men02], there are several drawbacks mitigating its advantages. Most importantly, operation-based approaches strongly depend on the modeling environment and only those operations are detectable that are supported by the modeling editor. Moreover, a set of manually applied atomic operations, having together the intent of a composite operation, which is indeed frequently happening in practice [MHPB09], cannot be identified by such approaches, because no explicit action has been issued in the modeling environment. Finally, in a usual setting, the evolution of models is stored in terms of revisions in traditional version control systems; consequently, the recorded operation logs are not available.

In the absence of an operation log, the applied operations have to be detected *a posteriori* using *state-based model comparison* approaches by either employing generic model comparison tools (e.g., [BP08, KWN05, LGJ07, SG08]) or language-specific comparison algorithms (e.g., [Kol09, XS05]). Whereas current generic approaches only support detecting atomic operations, some language-specific approaches also allow for detecting composite operations. However, the latter class of approaches is restricted to specific modeling languages. Moreover, the specification artifacts for automatically *executing composite operations* (cf. Section 4.1) in the

modeling editor and the implementation of the algorithms *for detecting* applications of them have to be manually kept consistent.

To address these mentioned problems, we propose to *reuse* existing specifications for executing composite operations also for detecting applications of them. In particular, we installed a dedicated adaptable step in the operation detection process (cf. Figure 5.1) after the step for detecting atomic operations. This step takes two inputs. The first input is a *difference model* (cf. $D_{V_o,V_{r1}}$ [atomic] and $D_{V_o,V_{r2}}$ [atomic] in Figure 5.1) containing the applied atomic operations. The second input is a set of composite operation specifications, which are automatically preprocessed to explicate their *diff pattern* representing the syntactical modifications of the composite operation for detecting applications of them. Now, the respective difference model is scanned for occurrences of those diff patterns. In order to consider also the semantic aspects of the composite operation, the respective parts of the considered models are checked whether they fulfill the composite operation's pre- and postconditions, respectively. The final output of this step is a difference model enriched with annotations for indicating applications of composite operations (cf. $D_{V_o,V_{r1}}$ [composite] or $D_{V_o,V_{r2}}$ [composite] in Figure 5.1).

The benefits of our approach are the following. Our approach does not rely on any editor-based operation tracking; thus, it is independent from the used modeling environment. As a further consequence, our approach is also capable of detecting applications of composite operations, even if they have been manually performed by applying their comprised atomic operations. Our approach is designed to be metamodel-agnostic. As the implementation of our approach is based on EMF [SBPM08], it can be used for any Ecore-based modeling language. No additional detection rules are required. In contrast, the set of detectable composite operations is automatically derived from existing operation specifications for executing the composite operations. Consequently, the set of detectable composite operations can be extended easily and is always kept consistent with the specification for executing them.

Although our composite operation detection approach is implemented for detecting applications of operation specifications created in EMO (cf. Section 4.1), we also show how operation specifications that are developed using any other declarative rule-based approach for endogenous model transformations (e.g., Henshin [ABJ+10], MT-Scribe [SWG09], etc.) can be supported. Therefore, in Section 5.3.1, we discuss the currently existing gap between specifications for executing composite operations automatically and rules for detecting applications of them. In this section, we further show how this gap can be bridged automatically. Subsequently, our approach for detecting applications of composite operations is presented in Section 5.3.2. In Section 5.3.3, we show how our approach can be applied iteratively to also allow the detection of composite operations that have been applied in a sequence. In Section 5.3.4, we elaborate how applications of composite operations are represented in the difference model for further processing. Before we conclude with a critical reflection and future work in Section 5.3.6, we survey related work for detecting applications of composite operations in Section 5.3.5.

Please note that in Section 7.2, we present a real-world case study concerning the extensive evolution of models coming from a subproject of Eclipse for evaluating the correctness and completeness of our implementation. Besides the correctness and completeness, in Section 7.2 we also investigate the scalability and performance of our implementation.

5.3.1 From Atomic Operations to Composite Operations

In this section, we describe the present gap between difference models containing atomic operations, as they are computed by currently existing model comparison frameworks, and operation specifications, which are created to allow for their (semi-)automatic execution in modeling environments. Subsequently, we outline our approach to bridge this gap.

5.3.1.1 Gap between Difference Models and Operation Specifications

Current state-of-the-art model comparison frameworks (cf. Section 5.2.3) produce difference models, called *diff models*, which comprise the atomic operations that have been applied between two versions of a model (cf. Section 5.2). Most of these model comparison approaches apply a two-phase comparison process: first, correspondences between model elements are established by applying *model matching* algorithms (cf. Section 5.1) for finding corresponding elements, and second, a *model diffing* phase, in which the actual differences between the two models from the previously established correspondences are computed. For instance, EMF Compare (cf. Section 3.4.3), which is one of the most prominent representatives of model comparison frameworks in the Eclipse ecosystem, is capable of detecting atomic operations; that is, additions, deletions, moves, and updates, that have been applied between an origin version and a revised version of a model. Please note that, for the sake of readability, we use a simplified representation of atomic operations in this section in comparison to the detailed representation introduced in Section 5.2.

An example for a diff model comprising solely atomic operations between two versions of a UML class diagram are depicted in the upper half of Figure 5.11. More specifically, this figure shows an origin model and a revised model in the concrete syntax as well as in the abstract syntax in terms of an object diagram. Between the origin model and the revised model, the refactoring *Extract Superclass* has been applied among other atomic operations. In the course of the applied refactoring, the new superclass Vehicle is introduced for the two existing classes. All common properties contained by the existing classes, that is, the property speed and horsePwr, are pulled up to the new superclass.

From these two model versions, all applied *atomic operations* can be derived using a state-based comparison. The obtained diff model is depicted in the middle of Figure 5.11 and comprises diff elements representing one addition of the new class, two updates for setting the new class as the superclass for both existing classes, two moves of the original properties of class Bike, and the deletion of the pulled up properties of class Car. Besides the refactoring, other atomic operations have been performed in this example: the class Car has been renamed to Automobile and the property fuel has been deleted. In the absence of a recorded operation log [Men02] that also directly tracks applied *composite* operations, the only way for users to reconstruct the information on the applied composite operation is to reason about the obtained atomic operations tediously in combination with the origin model and revised model.

However, as reported in several works [BKS+10, HBJ09, KHvW+10, Men08], working only on the atomic level of applied operations does not scale for several scenarios. To overcome this issue, a more concise view of model differences is required that aggregates the atomic operations into composite operation applications such that the common goal of the cohesive atomic operations is explicit. Unfortunately, current model comparison frameworks lack the

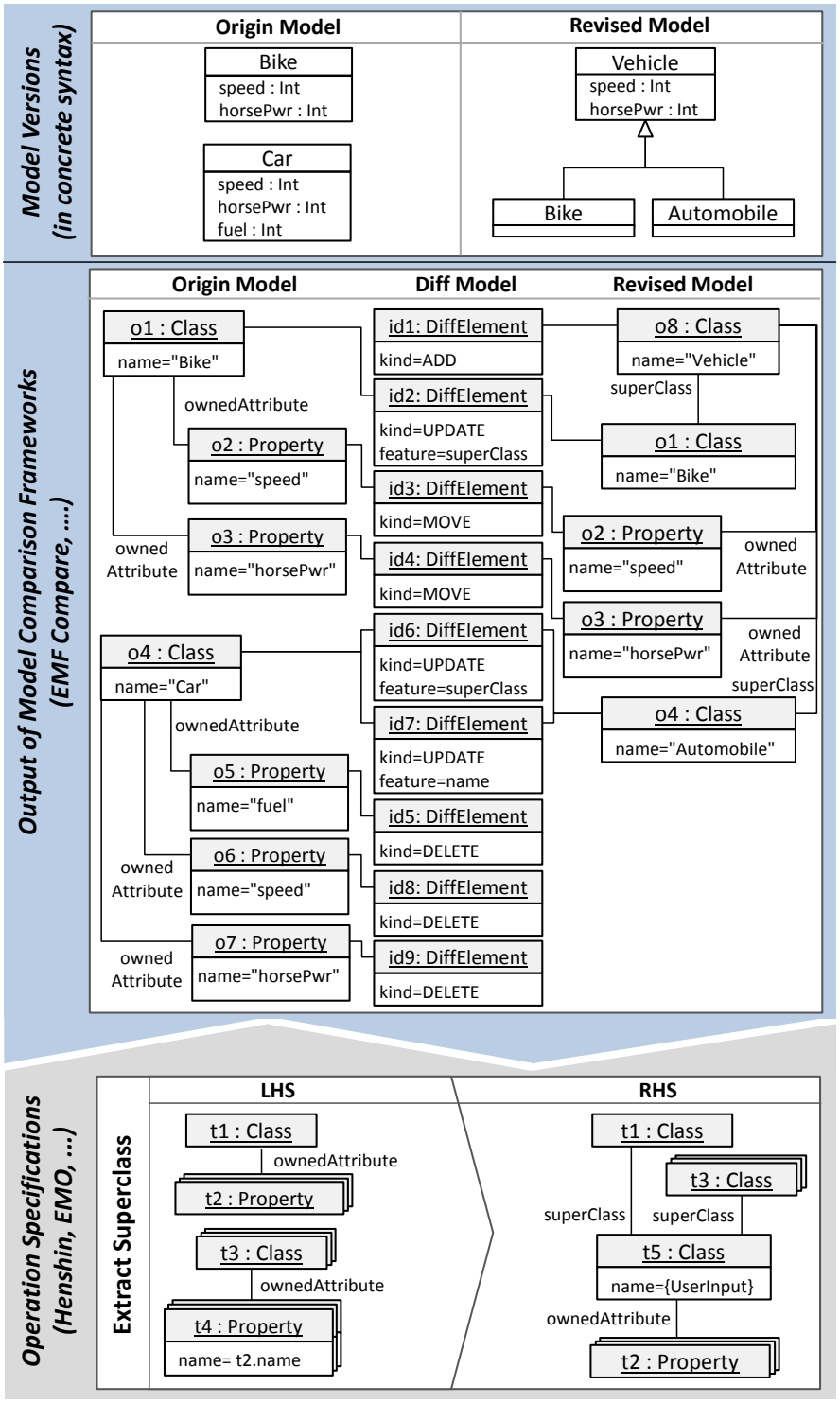


Figure 5.11: Gap between Atomic Diff Models and Composite Operation Specifications

means for detecting applications of composite operations in a generic way. Existing solutions [HGR10, KGFE08, XS06] only provide language-specific operation detection algorithms, which apply specifically implemented detection rules for a fixed set of composite operations. However, due to the plethora of existing modeling languages and composite operations that can be applied to their respective models, this is an unfavorable solution; because users have to tediously implement the dedicated operation detection algorithm for each language they use. A superior solution is to *reuse existing specifications of composite operations*, which presently only allow for their automatic execution, also for the *detection of their applications*. Thereby, no further implementation effort is necessary and the specification for executing the composite operation is automatically in sync with the specification artifacts for detecting applications of the respective operation.

When looking at current best practices for specifying executable composite operations, model transformations are the technique of choice (cf. [CH06] for an overview). In particular, composite operations are developed by specifying the operation's preconditions, its postconditions, and the actions that have to be executed for applying the operation. An example operation specification is depicted in the lower part of Figure 5.11 for the aforementioned refactoring *Extract Superclass* in graph transformation syntax [Hec06]. Thereby, the left-hand side (LHS) represents the precondition of the operation and the postcondition is specified in the right-hand side (RHS). Please note that graph transformations are a declarative specification approach; thus, the operation's actions are implicitly defined by the LHS and RHS. The precondition of the example operation states that only equally named properties may be pulled up to the new superclass. Therefore, the property denoted with t4 comprises the condition `name=t2.name`. Additionally, the operation should be applicable for more than one property and it should also be able to extract the superclass for more than one existing class. Therefore, the user configures so-called *iterations* (cf. Section 4.1.3; also referred to as *multi-objects* in graph transformation literature). Iterations indicate that more than one object may be matched with one element in the preconditions (e.g., t4) so that all matching objects are transformed equally when executing the composite operation. Due to the iterations in our example, the transformation rule is capable of pulling up multiple properties matching t2 and deleting all equally named objects matching t4 contained by multiple other classes corresponding to t3.

Current execution engines for model transformations, however, only provide the means for executing composite operation specifications, but they do not support detecting occurrences of an operation that have been applied between two versions of a model. Additionally, the operation specifications are not designed to be matched directly with diff models produced by current model comparison frameworks. Thus, we conclude that there is a gap between these two worlds: the specification for executing composite operations and the detection of applications of the composite operation.

In the following, we present a way to bridge this gap by an explicit integration layer allowing to detect applications of operation specifications in diff models in a generic way. By *generic*, we mean in this context that there is no dependency to the specific metamodels and no restriction regarding the specified composite operations. On the contrary, the approach should be applicable for all metamodels defined with Ecore and all composite operations that are defined with a declarative transformation language, such as current graph transformation languages.

5.3.1.2 Bridging the Gap between Difference Models and Operation Specifications

To bridge the gap between atomic operations detected by model comparison tools and operation specifications, we propose to generate an intermediate structure automatically from the operation specifications as illustrated in Figure 5.12. In particular, this intermediate structure extends the composite operation specifications by its comprised atomic operations. These atomic operations represent the operation's *diff pattern*, which can be matched with a diff model obtained from comparing two versions of a model. If a match of the diff pattern can be found within the diff model, we may proceed with evaluating the pre- and postconditions on the origin and revised models, respectively and, if these conditions are also fulfilled, an occurrence of the respective composite operation is reported.

For making the diff pattern of an operation specification explicit, we compute a diff model by applying existing model comparison tools to the LHS and RHS of the transformation rules. Other constructs of graph transformation rules, such as iterations or positive and negative application conditions, do not have to be considered as they can be checked in the subsequent evaluation of the pre- and postcondition. Of course, producing a diff pattern using existing model comparison tools requires that the LHS and the RHS of transformation rules be represented by "pure" models, i.e., instances of the modeling language metamodel, and not as models conforming to the transformation language metamodel as is often the case. Therefore, we apply a dedicated transformation to translate the LHS and RHS of the transformation rules to "pure" models (i.e., a direct instances of the modeling language's metamodel). This transformation is the inverse of the transformation used in [KMS+09] for generating a language-specific transformation language out of a modeling language. To be more precise, the LHS as well as the RHS of a transformation rule comprise so-called *templates* (cf. Section 4.1.4 for the definition of templates in EMO), which posses a *type*, a *variable*, *links* to other templates, *conditions* on attribute values in the LHS, *actions* for setting attribute values in the RHS, as well as a flag, which defines whether the template represents a multi-object or not. As shown in Figure 5.12, these templates are easily transformable to pure models by generating a pure object for each template. Thereby, the same types of templates have to be used also for the corresponding objects and the transformation specific modeling features are filtered out. Moreover, to each object an ID according to the template variable (e.g., t1) is assigned for preserving the trace between templates and generated objects. Having generated pure models from the LHS and the RHS, we may now apply existing model comparison tools to derive the explicit diff pattern (i.e., the differences between LHS and RHS) representing the minimal set of atomic operations the composite operation consists of.

The explicit diff pattern of operation specifications allows us to close the gap between operation specifications and diff models. Following this approach, we demonstrate the power of the main principle of model engineering, namely *everything is a model* [Béz05]. Please note, however, that the diff pattern abstracts away several transformation-specific information. Thus, the diff pattern allows for a fast preselection of *potentially applied operations*. The transformation-specific information, that is, pre- and postconditions as well as iterations, have to be subsequently checked. Therefore, we may reuse the corresponding techniques from the respective model transformation approach. The corresponding techniques that are used in EMO are presented in Section 4.1.4.

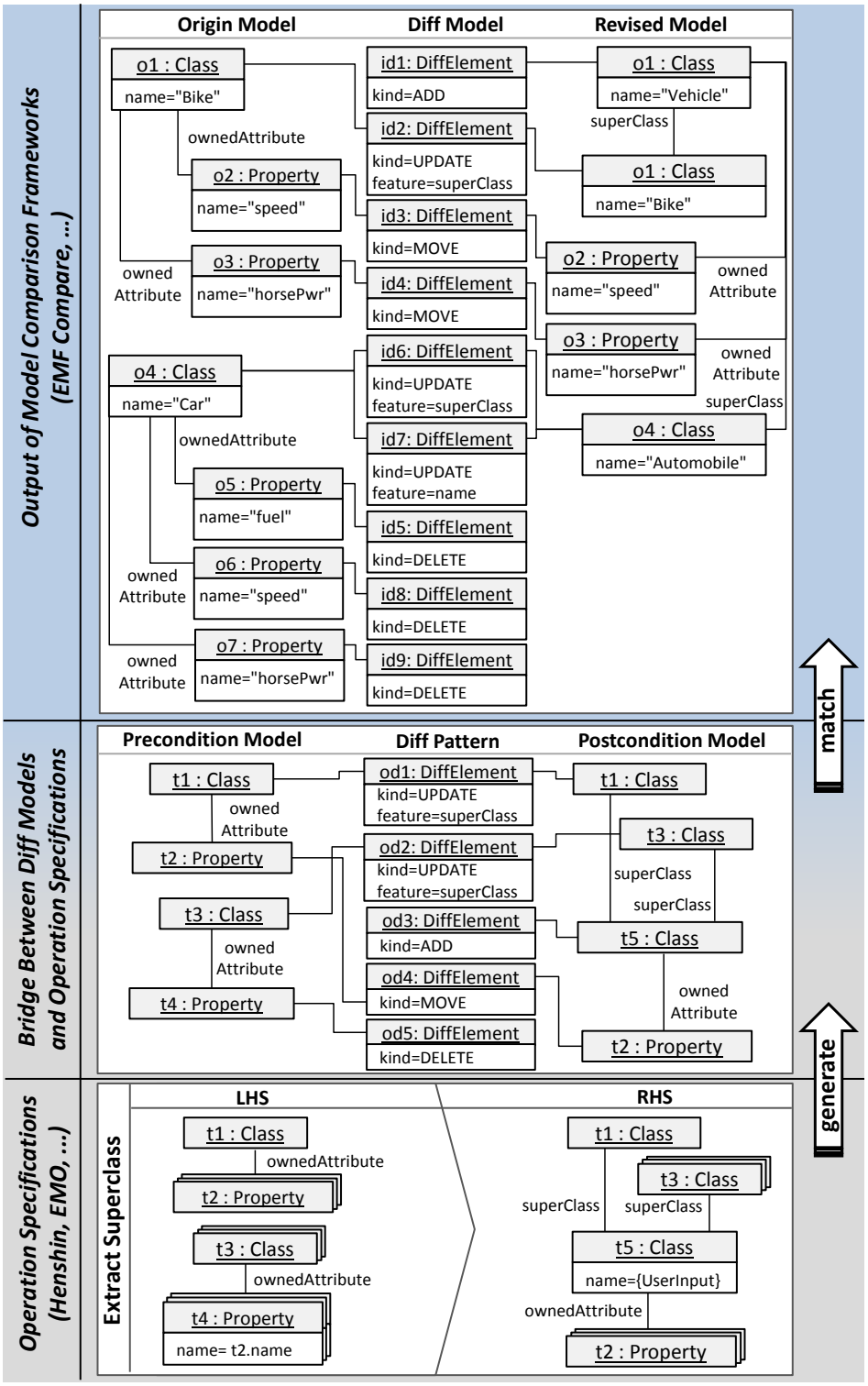


Figure 5.12: Bridge between Atomic Diff Models and Composite Operation Specifications

5.3.2 A Posteriori Composite Operation Detection Process

Having discussed how to generate the bridge between diff models and operation specifications in terms of diff patterns, we now proceed with describing our approach for detecting applications of composite operations among a set of atomic operations. More specifically, in this section, we first discuss the phases of the proposed detection process from a high-level view. Subsequently, we illustrate the process step-by-step using the example of the previous section. Finally, we give some details on the implementation of our approach.

Composite Operation Detection at a Glance

The composite operation detection process is depicted in Figure 5.13 in terms of a UML activity diagram. It has two inputs and consists of three phases. The first input is the diff model called input diff model containing the atomic operations that have been applied between two versions of a model. Secondly, our process takes an arbitrary number of operation specifications as input, which constitute the set of detectable composite operations. Please note that these operation specifications also comprise the diff patterns derived as described in the previous section. In the first phase of the process, the operation specifications' diff patterns are exploited for efficiently preselecting all composite operations that potentially have been applied between the two versions of a model. Therefore, the input diff model is searched for the diff patterns. As several transformation-specific features are abstracted in the generated diff patterns, these features have to be considered in the subsequent phases. Thus, in the second and third phase, for each potential composite operation occurrence, the pre- and postconditions of the composite operation are evaluated, respectively. If both are valid, an application of a composite operation is at hand and added to the output list of operation occurrences. In the following, the three phases are described in detail using the example introduced in Figure 5.12.

Phase 1: Diff Pattern Matching

The goal of this phase is to check whether the diff patterns of the given operation specifications are contained by the input diff model.

Diff model preprocessing. In a first step, the input diff model as well as all diff patterns are preprocessed and translated into so-called signatures (*input signature* for the diff model and *operation signatures* for the diff patterns). These signatures contain the relevant information of the diff elements in an easily processable format. In particular, the signature represents the operation kind and the metamodel types it affects. Thus, the format of a diff element in the signature is comparable to a common method signature in programming languages. For instance, if a UML property has been added, the corresponding signature has the form of `Add(UML:Property)`. Please note that in the implementation of our approach, more information such as the type of the new parent and sibling diff elements is encoded in the signature to increase the precision of the matching. In this section, however, we omit this additional information for the sake of readability.

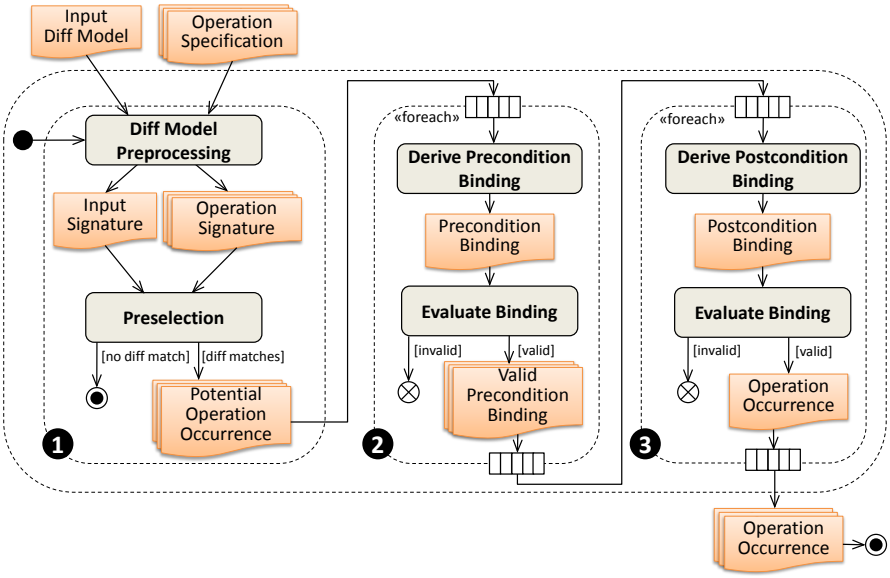


Figure 5.13: Process of Composite Operation Detection: (1) Diff Pattern Matching, (2) Precondition Matching, (3) Postcondition Matching

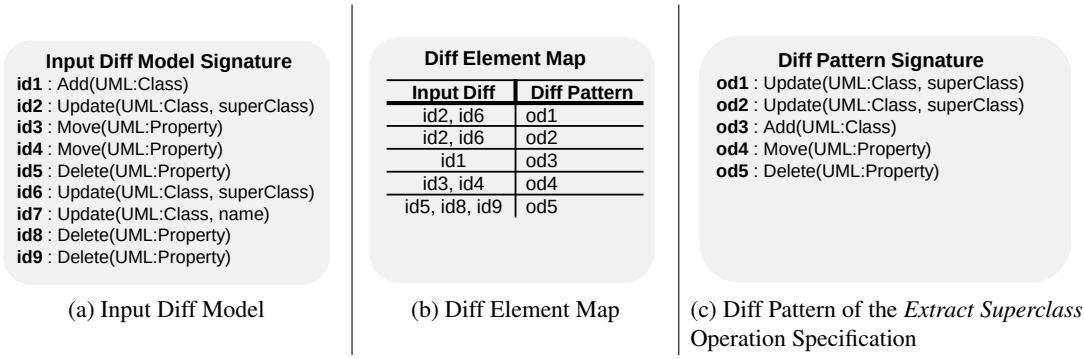
Preselection. In the next step, the preselection of potentially applied composite operations is accomplished based on the input signature and the operation signatures. The goal of this step is to check whether the signature of each operation specification is entirely contained in the input signature. If the operation signature is not contained, the respective operation specification may immediately be rejected as it definitely has not been applied according to its corresponding diff pattern. The used procedure for realizing the preselection is depicted in Algorithm 5.1. This algorithm initially adds all operation signatures to the preselection (cf. line 1 in Algorithm 5.1). Subsequently, it iterates over all operation signatures and checks for each operation signature element in the operation signature whether it is contained in the input signature (cf. line 4 in Algorithm 5.1). In case an operation signature element is not contained in the input signature, the respective operation specification cannot have been applied, as it is not entirely contained in the input signature; thus, the respective operation specification is rejected (i.e., removed from the preselection). All operation signatures that remain in the preselection, after the algorithm terminates, are entirely contained in the input signature. Consequently, they constitute *potential operation occurrences*. After the preselection of potentially occurred operations, a so-called *diff element map* is created. This diff element map saves for each diff element in a diff pattern the corresponding diff elements from the input diff model (i.e., a trace of input diff elements to matching diff elements in the diff pattern). This map is needed later and, therefore, handed over, alongside the preselection, to the next phases.

Example: Preselection. In the first phase of the composite operation detection process, the input diff model (cf. Figure 5.14a), as well as the diff pattern from the operation specification (cf. Figure 5.14c), are preprocessed and translated into the aforementioned signature format.

Input: inputSignature, operationSignatures
Output: preselection

```
   // Initially add all operation signatures to the preselection
1  preselection.addAll(operationSignatures)
2  for operationSignature ∈ operationSignatures do
3      for operationSignatureElem ∈ operationSignature do
4          if ¬ inputSignature.contains(operationSignatureElem) then
               // operationSignatureElem is not contained in
                  inputSignature. Thus, operationSignature is not entirely
                  contained in inputSignature → reject operationSignature
5              preselection.remove(operationSignature)
6              break
7          end
8      end
9  end
10 return preselection
```

Algorithm 5.1: Preselection based on Diff Patterns



(a) Input Diff Model

(b) Diff Element Map

(c) Diff Pattern of the *Extract Superclass* Operation Specification

Figure 5.14: Example for Preselection

Next, the preselection algorithm (cf. Algorithm 5.1) is applied to these signatures for checking whether the diff pattern of the operation specification is entirely contained in the input diff model, which is the case in our example. Furthermore, the diff element map is built, which explicitly indicates the matching diff elements among those two sets of diff elements (cf. Figure 5.14b). For instance, both diff elements in the diff pattern representing the update of the superclass of a class, namely od1 and od2, match with both corresponding diff elements, id2 and id6, in the diff model (cf. Figure 5.14b). Finally, a *potential operation occurrence*, which contains the operation specification as well as the diff element map, is created and handed over to the next phase.

Phase 2: Precondition Matching

In this phase, for each potential operation occurrence detected by the first phase, we check whether the preconditions are fulfilled. As already mentioned, the pre- and postconditions are organized in *templates*, which describe the role a model element plays in the composite operation (e.g., a property to be pulled up to the superclass). Each template contains relationships to other templates and conditions restricting a certain feature value of a model element (cf. Section 4.1.4 for more information on templates in EMO).

Derive precondition binding. The evaluation of the conditions might be rather time-consuming. Thus, we aim at checking only those model elements that have been modified according to the diff pattern. Therefore, we specifically compute a so-called *derived template binding*, which is as small as possible. A template binding maps the respective model elements (to be checked) to their corresponding templates (to be checked against). The derivation of this template binding is created based on the information in the diff element map, obtained from the previous phase, by exploiting the reference from diff elements to the model elements they affect. More precisely, we first determine the affected precondition template for each diff element in the diff pattern by navigating from the respective diff element via the aforementioned reference to the affected template (e.g., od1 $\rightarrow$ t1 in Figure 5.12). Based on this information, we can now create a binding of the affected template to each model element in the origin model that is affected by the corresponding input diff element as indicated by the diff element map.

Example: Derivation of precondition binding. The obtained template bindings in our example are depicted in the *derived precondition binding* box in Figure 5.15. For obtaining this binding, the following steps are performed. The diff element od1 in the diff pattern affects the template t1 (cf. Figure 5.12). According to the diff element map, od1 is mapped to the diff elements id2 and id6 from the input diff model. These two diff elements in turn affect the model elements o1 and o4 in the origin model, respectively. Thus, we create a binding of template t1 to the model elements o1 and o4. Next, we consider the diff pattern element od2, which leads to the same bindings as before (for od1), because it is mapped to the same input diff elements. The next diff pattern element od3 represents an addition of a model element; thus, it is skipped. The diff element od4 in the operation's diff pattern represents the move of template t2. The input diff elements that are mapped to od4 are id3 and id4, which affect the model elements o2 and o3, respectively. Consequently, we create a binding of template t2 to the model elements o2 and o3. The same procedure is repeated, until all diff elements in the diff pattern have been processed.

Evaluate precondition binding. The resulting *derived precondition binding* is evaluated using a condition evaluation engine. For this task, we may reuse evaluation engines that are shipped with the respective model transformation approaches for finding valid template bindings. However, there is one major requirement, which must be fulfilled by this engine in order to be usable for our needs: it must be capable of *detecting all valid bindings* among a set of model element candidates as specified in the derived precondition binding. In particular, it must return the set of all unique bindings among the candidates. Thereby, multiple bindings should only be allowed

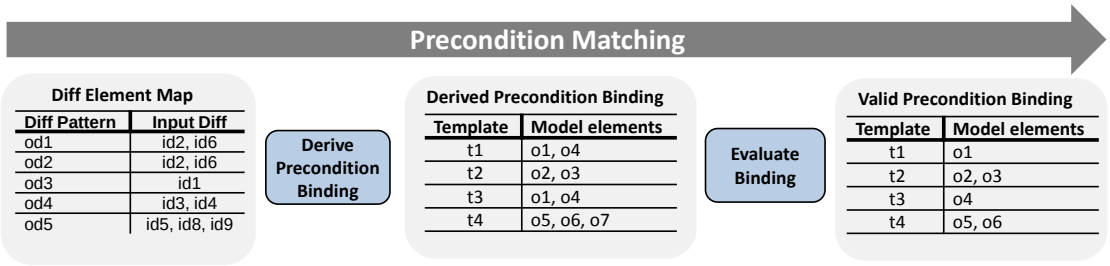


Figure 5.15: Precondition Matching and Evaluation

if iterations are attached to the respective template and model element candidates that are not part of any valid binding should be rejected. If one operation specification has been applied more than once, the evaluation engine must return a valid precondition binding for each potential application of the respective composite operation. The engine must also be capable of autonomously binding suitable model elements from the origin model to those templates that are not bound in the derived precondition binding. In Section 4.1.4, we have presented how these requirements are addressed in the template matching engine that is used in EMO.

After all potential operation occurrences have been checked, a list of valid precondition bindings is handed over to the next phase for checking also the postconditions. Obviously, if no valid bindings could be found, it is not necessary to continue to the next phase.

Example: Evaluation of the precondition binding. In our example, we have to evaluate the derived precondition binding depicted in Figure 5.15. Note that at this moment, the property called fuel with the object id o5 is bound to the template t4, as it is a valid model element according to the diff pattern; because this property has been removed. However, in the next step, the template binding evaluation engine rejects this object because the precondition `t4.name = t2.name` fails: there is no property bound to template t2 having the name fuel. For the objects o6 and o7, there are in fact two properties bound to t2 having the same name—o2 and o3, respectively. As a result, the valid precondition template binding depicted at the right side of Figure 5.15 is returned. Although there are two templates, namely t2 and t4, to which more than one model element is bound, this binding is still valid, because of the iterations attached to these templates.

Phase 3: Postcondition Matching

For each valid precondition binding computed in the previous phase, we check the postconditions. Therefore, we first derive a template binding of model elements from the *revised model* to the *postcondition templates*. Again, this is done using the aforementioned diff element map created during the diff pattern matching phase. First, the essential input diff elements are filtered for the current precondition binding to avoid unnecessary bindings to be evaluated. Therefore, only the diff elements are considered that directly affect the model elements bound in the precondition binding. All other diff elements are obviously obsolete now. For instance, the input diff element id5 is removed from the map, because the affected model element o5 (i.e., the property fuel)

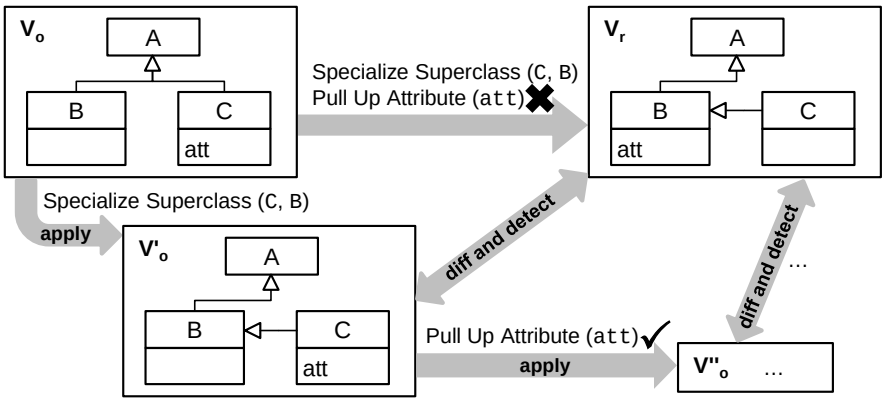


Figure 5.16: Iterative Composite Operation Detection

has been rejected during the precondition evaluation. Apart from the fact that the postcondition binding refers to model elements in the revised model, the derivation of the postcondition binding works analogously to the precondition binding derivation. This derived binding is evaluated again using the template evaluation engine. If a valid postcondition binding is found, an occurrence of a composite operation is at hand. Consequently, a *composite operation occurrence* is created and added to the final output diff model.

5.3.3 Iterative Composite Operation Detection

In several scenarios, multiple composite operations are sequentially applied to overlapping parts of the model. Consider the example depicted in Figure 5.16, in which the developer first applies the composite operation *Specialize Superclass* by changing the superclass of C to B. Subsequently, the same developer performs the composite operation *Pull Up Attribute* by moving the attribute att from class C to its new superclass B. When considering only the origin model V_o and the revised model V_r, our approach is only capable of detecting the first composite operation *Specialize Superclass* because the preconditions of the second operation *Pull Up Attribute* are not fulfilled as in the origin model V_o the class B is not a superclass of C.

To overcome this limitation, we use an iterative composite operation detection. Therefore, we *apply* all detected composite operations to the origin model V_o leading to a new origin model denoted with V_o' in Figure 5.16 and re-start the operation detection again to the new scenario $V_o' \rightarrow V_r$. Thereby, we first apply the model differencing algorithm to the new origin model V_o' and the revised model V_r and, subsequently, search again for occurrences of composite operations in the resulting diff model as presented above. This iterative process is repeated ($V_o'' \rightarrow V_r$, $V_o''' \rightarrow V_r, \ldots$) until no new applications of composite operations can be found. In our example (cf. Figure 5.16), this iterative procedure now enables also the detection of *Pull Up Attribute* because the preconditions of this composite operation, which restrict class B being a superclass of C, are fulfilled in the new origin model V_o'.

Please note that although this iterative approach allows for additional detections of sequentially applied composite operations in several scenarios, it significantly increases the required

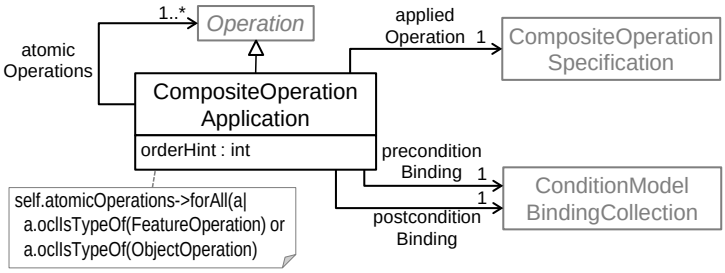


Figure 5.17: Composite Operation Application Metamodel

runtime and it does not work for *all* possible sequentially applied overlapping composite operations. We summarize all limitations of our approach in Section 5.3.6. The particular challenge of sequentially applied composite operations is also discussed in our case study presented in Section 7.2.

5.3.4 Representing Composite Operation Applications

Once all applications of composite operations are detected, they have to be represented accordingly in the difference report alongside the applied atomic operations. Thereby, we aim at fulfilling the following requirements. First of all, applications of composite operations should be represented in terms of models. Second, to avoid any information loss, the atomic operations constituting the composite operation application should not be lost; they should be grouped into its own container instead. Finally, the model-based representation of composite operation applications should comprise enough information to allow for executing them again to other versions of a model as this is required for merging concurrent versions of a model. Thus, the representation should give explicit information on *which operation specification* has been executed and *which model elements* have been bound to *which precondition and postcondition template* of the respective operation specification.

Therefore, we extend the difference metamodel presented in Section 5.2.2 by a dedicated class called CompositeOperationApplication as depicted in Figure 5.17. Please note that the classes depicted in gray in this figure are part of other metamodels and are only referenced from this metamodel for composite operation applications. In particular, this metamodel uses classes from the difference metamodel (cf. Figure 5.9), the operation specification metamodel (cf. Figure 4.6), and the template binding metamodel (cf. Figure 4.8).

The class CompositeOperationApplication is specified as a subclass of Operation from the difference metamodel. Thus, instances of CompositeOperationApplication can be added easily alongside other operations in a difference model. The atomic operations that constitute the composite operation application are indicated through the reference atomicOperations. As a composite operation may only consist of atomic operations and not of other composite operations, we specified a class invariant in Figure 5.17 to ensure that referenced operations must be either instances of FeatureOperation or ObjectOperation (i.e., they must be *atomic*). The class CompositeOperationApplication refers to the CompositeOperationSpecification through the reference appliedOperation to provide information on which composite operation has been

executed. Furthermore, it contains two references to ConditionModelBindingCollection: one for representing the template bindings of the precondition templates and one for the postcondition templates. Finally, the attribute orderHint indicates the iteration (cf. Section 5.3.3) in which the application of the composite operation has been detected. Consequently, this attribute tells us the order of the composite operation applications and, thereby, which detected applications can be reapplied only *after* other detected operation applications. For instance, applications with an order hint of *2* can only be applied after applications having an order hint of *1* already have been performed.

5.3.5 Related Work

Several approaches recently emerged to record directly or to detect a posteriori applied composite operations in different technical spaces. Most of them are designed for detecting refactorings in object-oriented programs, but there are also some dedicated approaches focusing on high-level specifications such as models and ontologies.

Object-oriented programming. The easiest way to capture applied refactorings is to track their execution directly in the development environment. In the context of versioning, such approaches are often referred to as *operation-based* versioning/merging [LvO92]. Refactoring tracking is for instance realized by [DMJN08, EA04, Rob07]. All these approaches highly depend on the used development environment which has to record the applied refactorings. Consequently, the set of detectable refactorings depends on the set of refactorings available in the development environment. Furthermore, performed manually refactorings are not detectable and refactorings which have been made obsolete by successive changes might be wrongly indicated.

State-based refactoring detection mechanisms aim to reveal refactorings a posteriori on the base of the two successively modified versions of a software artifact. For instance, Dig et al. [DCMJ06] propose an approach to detect applied refactorings in Java code. They first perform a fast syntactic analysis and, subsequently, a semantical analysis in which also operational aspects like method call graphs are considered. A similar approach is followed by [WD06]. After a preprocessing and a syntactical analysis have been conducted, conditions indicating the application of a refactoring are evaluated. Another heuristic-based approach is presented in [DDN00] in which a combination of various software measures as indicator for a certain refactoring is used. For instance, a decrease in a method's size, among other measures, is used to indicate that the refactoring *Split Method* has been applied.

Refactoring detection in code artifacts is in general more challenging than in model artifacts. In models, relationships between the model elements are usually explicitly available in the form of direct references represented by an address or an ID. In code, elements usually have no persistent address or ID and, therefore, have to be matched using name and content similarity measures. When detecting model refactorings, we face a multitude of different modeling languages. Consequently, there is a plethora of different refactorings as well as various forms and diverging implementations of the same refactoring. Hence, hard-coded detection strategies for a predefined set of refactorings are not a desirable solution for software models.

Model engineering. To the best of our knowledge, there exist three approaches for the a posteriori detection of composite operations in evolving software models. First, there is the approach by Xing and Stroulia [XS06] for detecting refactorings in evolving software models which is integrated in *UMLDiff*. Refactorings are expressed by change pattern queries used to search a difference model obtained by a state-based model comparison. However, UMLDiff only supports a fixed modeling language, namely a subset of structural UML diagrams. To add further composite operations, users have to develop new queries which seems to be more difficult than simply reusing existing specifications for executing the composite operations. Furthermore, only simple change patterns and no complex pre- and postconditions are regarded.

Second, the approach by Vermolen et al. [VWV11] copes with the detection of complex evolution steps between different versions of a metamodel to allow for a higher automation in model migration. For this, they use a diff model comprising primitive operations as input and calculate on this basis complex operations. The approach is tailored to the core of object-oriented metamodeling languages, but follows a similar methodology as UMLDiff. However, a specific feature is the detection of masked operations, i.e., operations which are hidden by other operations in a way that their effect is partially or also totally missing in the revised model, by defining additional detection rules. Nevertheless, the approach is again dedicated to one single modeling language and does not allow to reuse the operation specifications used for execution for the detection process.

Third, the work of Küster et al. [KGFE08] for calculating hierarchical change logs including compound changes in the absence of recorded change logs is widely related. The authors apply the concept of Single-Entry-Single-Exit fragments to calculate the hierarchical change logs after computing the correspondences between two process models. Thereby, several atomic changes are hidden behind one compound change. The difference between the work of Küster et al. and ours is twofold. First, we consider the detection of composite operations comprising changes cross-cutting the whole model, i.e., we have no restriction to a sub-part of a model, whereas Küster et al. mainly consider the insertion of a new sub-part into the model hierarchy. Second, our approach is language independent and extensible for arbitrary operations, thus we are not restricted to process models.

Ontology engineering. There is also widely related work in the field of ontology engineering. Hartung et al. [HGR10] present an approach for generating so called semantically-enriched evolution mappings between two versions of an ontology. Evolution mappings can be seen as diff models which comprise atomic as well as composite operations. The goal of Hartung et al. is to produce a minimal diff model by using a rule-based system for minimizing the atomic operations by first finding composite operations for a set of atomic operations which is subsequently eliminated. The approach is tailored to an ontology language comprising concepts, attributes, and relationships as well as to a small set of composite operations such as moving, splitting, and merging concepts by providing specific detection rules. Finally, they apply aggregation functions to further shrink the size of the diff model by combining composite operations, which is in our presented approach directly integrated in the composite operation detection by using iterations in the transformation rules.

In summary, our approach is the first which allows to reuse the operation specifications used for execution also for the detection process. All other approaches (irrespective of the field) are either change-tracking based or specifically developed for a dedicated language and pre-defined composite operations which have to be re-formulated as a form of detection rules. Instead, our approach does not rely on a change-log, but is generic in the sense that all Ecore-based models and available operation specifications defined as graph transformation rules are directly supported.

5.3.6 Limitations and Future Work

We made some assumptions to allow a precise detection of composite operations in evolving models. In this section, we discuss these assumptions and how the presented approach might be altered or extended to also work adequately if these assumptions are dropped.

Precise model matching. We assumed that the original model elements and their revised counterparts of two successive model versions may be precisely and correctly matched. This is usually only the case when model elements are annotated with persistent (natural or synthetic) identifiers. If such identifiers are missing, the model match depends on heuristics like name or content similarity measures only. Since heuristics inevitably lead to imprecision, a model element which, for instance, was intensely modified and moved might not be correctly matched and, consequently, a deletion and addition of this element instead of a move is reported. Obviously, this also affects the precision of the operation detection. For more information on model matching, we kindly refer to Section 5.1.

Overlapping atomic operations. When state-based comparison is applied, only the effective atomic operations are obtained. For instance, if a model element has been updated and subsequently deleted, only the deletion of the model element is detected at the end. On the one hand, this is a significant advantage, because relying on a state-based comparison, only truly effective composite operation occurrences are revealed. Operation recording approaches, in contrast, have to address the challenge to avoid wrong indications of composite operations, when the tracked application of a composite operation has been invalidated by subsequently applied operations. On the other hand, there might exist scenarios, in which a composite operation has been effectively performed, although subsequent operations hide essential operations of a composite operation's diff pattern. To enable our approach to also regard such scenarios, we propose to consider operation kinds that potentially hide other operations as "joker". For instance, a deletion of a model element can be considered to be an update and a move of the same model element in the diff pattern checking phase to this end. Analogous to the pre- and postcondition loosening discussed in the previous paragraph, this leads to a higher precision but to a lower recall. Again, this prioritization of measures depends on the application domain.

Preceeding and subsequent changes. In our approach, a composite operation occurrence is only reported if *all* pre- and postconditions are completely valid. With this strategy, a very high recall is obtained, as it can be seen in our case study (cf. Section 7.2). However, in some

scenarios, it might happen that atomic operations have to be performed first in order to obtain valid preconditions before a composite operation can be applied. Analogously, it is possible that subsequent operations lead to failing postconditions after a composite operation has been executed. Our implementation also supports loosening the strictness of condition evaluation in a way that a *partial* condition validity (to a certain threshold) can be checked. Of course, this increases the number of detected composite operations, but also leads to imprecision and, thus, decreases the recall. Moreover, it is hard to generically determine which conditions are essential to an application of a composite operation. It does not seem to be generally answerable whether, for instance, the composite operation *Extract Superclass* has been effectively applied, if the condition restricting the operation names to be equal fails. Thus, we propose to annotate truly essential conditions manually that have to be fulfilled in any case for successfully reporting a composite operation occurrence. It strongly depends on the application domain and the goals for employing the composite operation detection. If a model repository is mined to get an idea of how a model evolved, a high precision might be more important than a high recall. In contrast, users might prioritize recall over precision for automatic merging in model versioning.

Overlapping composite operation sequences. Finally, as also discussed in Section 7.2, the detection of overlapping composite operation sequences represents a major challenge for state-based approaches, because they lead to invalid pre- and/or postconditions and it is very likely that atomic operations are hidden by subsequent operations. To enable our approach to detect composite operation sequences more precisely, we propose, besides the iterative detection, to precalculate potentially combinable composite operation specifications on the basis of their pre- and postconditions. For instance, if the preconditions of composite operation *A* fit to postconditions of composite operation *B*, *A* potentially might be executed after *B*. If a valid combination is revealed, both operation specifications can be automatically merged to create a new *composite composite operation*. To merge operation specifications, the conditions as well as the examples have to be combined and a new composite diff pattern has to be computed. For limiting the search space of combinable composite operations, we may use the critical pair analysis comparable to how it has been done in [Men06]. However, so far, many kinds of composite operation sequences are currently unsupported by our approach posing a very interesting direction for future work.

CHAPTER 6

Conflict Detection

In the previous chapter, we discussed how operations, which have been applied between two versions of a model are obtained and explicitly represented without relying on editor-based operation recording. In this chapter, we proceed with presenting our approach to detect *conflicts* among operations. Besides detecting conflicts, we also show how additional merge issues can be identified, which potentially lead to flawed merged model. For such issues, we raise merge *warnings* to notify developers that the merge should be reviewed concerning potential flaws. For a more detailed definition of the terms *conflict* and *warning*, and how they are related to each other, we kindly refer to Section 3.2.

In Figure 6.1, we depict the particular five steps of the adaptable merge process of AMOR (cf. Figure 3.12) that realize the conflict and warning detection. The input of conflict and warning detection are two difference models, $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$, which describe the atomic and composite operations that have been applied between V_o and V_{r1} and V_o and V_{r1}, respectively. In the following, we briefly outline each step and refer to the section that presents the respective step of the conflict and warning detection process in more detail.

Atomic Operation Conflict Detection. The goal of the first step, the atomic operation conflict detection, is to find concurrent *atomic operations* that interfere with each other. The additional information on applied composite operations remains unconsidered in this step. However, the atomic operations that constitute the composite operation applications are regarded in this step. The output of the atomic conflict detection is a conflict model, which describes all revealed *atomic operation conflicts*. The techniques we apply for identifying such conflicts are presented in Section 6.1.

Composite Operation Conflict Detection. The next step is the composite operation conflict detection, which takes the knowledge on the ingredients of composite operations, such as pre- and postconditions, into account for revealing additional merge issues. This knowledge is specified by users in terms of operation specifications (cf. Section 4.1). As in the previous step, also this step adds detected merge issues to the conflict model, which is subsequently passed on to the

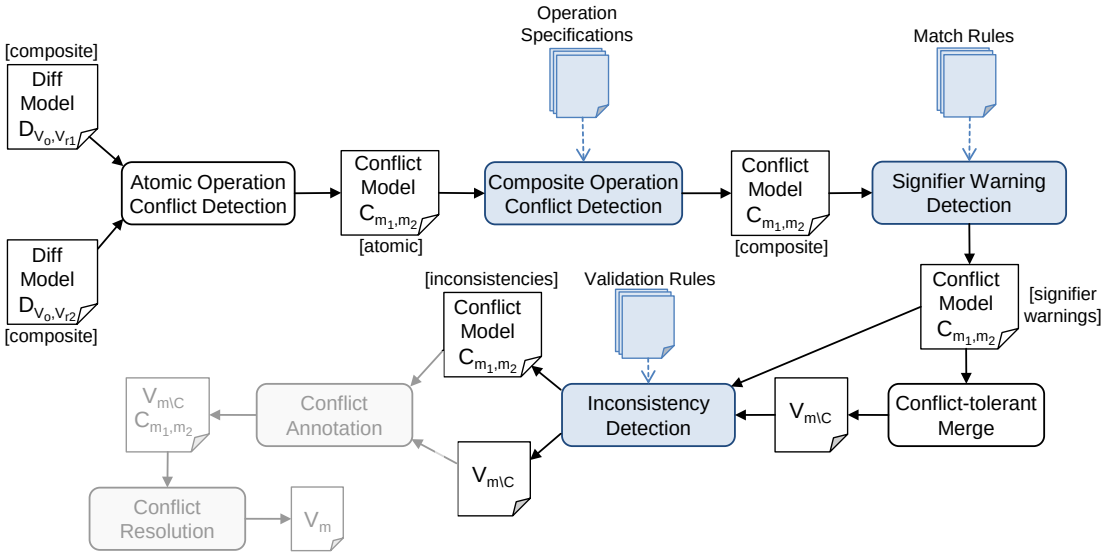


Figure 6.1: Conflict and Warning Detection Steps in the Adaptable Merge Process

next step. Our approach for detecting conflicts and warnings concerning composite operations is addressed in Section 6.2.

Signifier Warning Detection. Having obtained atomic and composite operation conflicts in the previous steps, the next step called signifier warning detection aims at detecting unexpectedly matching model elements and inadvertent concurrent changes to a model element's signifier based on user-specified language-specific match rules. If such merge issues are detected, corresponding warnings are added to conflict report and handed over to the subsequent step. The detection of warnings concerning model element's signifiers is introduced in Section 6.3.

Inconsistency Detection. Even if the origin model V_o and both revised models, V_{r1} and V_{r2}, conform to the modeling languages validation rules, the merged model obtained from these revisions may still comprise inconsistencies. Thus, the goal of this step is to detect such inconsistencies. Model inconsistency detection is a very active research area on its own. Recently, several remarkable approaches emerged to address the challenge of efficiently and reliably validating models. Consequently, instead of reinventing the wheel, we rather use one of those existing approaches for detecting inconsistencies and integrate it into the merge process. However, before we may search for language-specific model inconsistencies caused by the concurrently applied operations, we first have to obtain a merged model to be evaluated against user-specified validation rules. As conflicts might have been detected in previous steps, we apply a conflict-tolerant merge, which is capable of producing a merged model, called $V_{m\setminus C}$, irrespectively of occurred conflicts. As this merge strategy (cf. [Wie11] for more details), as well as the actual approach for detecting inconsistencies, is not the particular focus of this thesis, we only provide a brief overview on these steps and discuss their integration into the merge process in Section 6.4. The

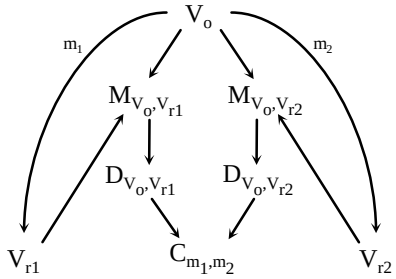


Figure 6.2: Model Versions, Match Models, Diff Models, and the Conflict Model

output of this component is the final conflict model, which ultimately contains all detected conflicts and raised warnings by the previous steps, as well as the inconsistencies identified in this step. Moreover, also the tentatively merged model $V_{m\backslash C}$, which builds the basis for the conflict resolution, is passed on to the subsequent steps.

To assist the user in resolving all occurred conflicts, the tentatively merged model is annotated with detailed information on the raised conflicts and warnings by placing dedicated stereotypes on top of the model's concrete syntax using EMF Profiles [LWWC11]. Finally, the annotated model is presented to the user, who may resolve all raised issues. Please note that there is also ongoing work on providing automatic conflict resolution recommendations in [Bro11]. However, conflict annotations and conflict resolution is beyond the focus of this thesis and, therefore, is only briefly outlined at the end of in Section 6.4 to round up the presentation of the adaptable merge process. For more information on how these two steps are realized in AMOR, we kindly refer to [Bro11, Wie11].

After we present our approach for detecting conflicts, warnings, and model inconsistencies in Sections 6.1 to 6.4, we conclude with a critical discussion of our approach and reveal current limitations and interesting directions for future work in Section 6.5. Please note that the related work in the realm of conflict detection in model versioning has already been surveyed and discussed in Section 2.1.2.

Furthermore, we present an evaluation of our approach for detecting conflict in Section 7.1. Due to the lack of techniques for evaluating and comparing existing approaches, we designed and realized a dedicated benchmark, which allows to assess implementations of conflict detection approaches automatically. We applied this benchmark to our approach, as well as to currently existing state-of-the-art approaches in the EMF ecosystem, to allow for a detailed comparison of their abilities for detecting certain conflict types.

6.1 Atomic Operation Conflict Detection

Operation-based conflicts denote two operations that are either parallel dependent or not commutative (cf. Section 3.2). Consequently, in the atomic conflict detection process, we search for combinations of two *atomic* operations in the difference models $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$ and check whether these operations are parallel dependent or not commutative. If this is the case for

two operations, a conflict is raised. Therefore, we create a description of the conflicting atomic operations and add it to the conflict model C_{m_1,m_2} (cf. Figure 6.2).

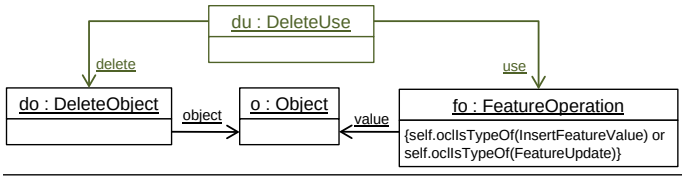
In the remainder of this section, we first introduce all types of atomic conflicts, which may occur when two developers concurrently modify EMF models, and show how these conflict types are detected. Subsequently, we discuss the technical realization of the atomic conflict detection and introduce our conflict metamodel for describing occurred atomic conflicts, which is the output of the atomic conflict detection process.

6.1.1 Atomic Conflicts Types

We define types of atomic operation conflicts in terms of *conflict patterns*, which are depicted as UML object diagrams (cf. Figures 6.3-6.11). On the one hand, these patterns serve as a clear specification of the respective conflict type and, on the other hand, they can be used for detecting conflicts of these types. More precisely, if such a conflict pattern matches with corresponding parts in the difference models, $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$, a conflict of the respective type occurred. As many conflict type cannot be specified by simple object patterns solely, we denote further constraints for more precisely defining these patterns in curly brackets in the respective objects of the patterns using OCL (e.g., {`self.oclIsTypeOf(...)`}). Alongside the conflict patterns, we further introduce a conflict metamodel to represent detected conflicts (cf. Figure 6.12), which is discussed in more detail at the end of this section. This metamodel contains for each conflict type a dedicated metaclass, which is refined by additional OCL invariants stated in the aforementioned conflict patterns. The instance of the conflict metaclass created for each detected conflict is depicted in green in the respective conflict patterns.

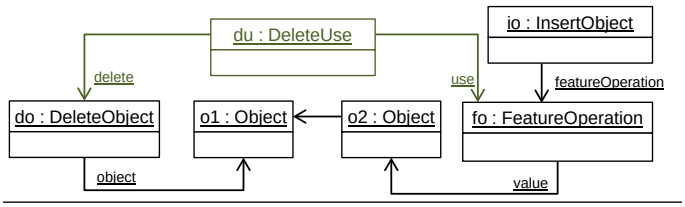
Delete-use conflict. The first conflicting combination of two operations in EMF models concerns the deletion of an object and concurrently linking to exactly the same object by setting a reference. We call such a conflict *delete-use* conflict because the *deleted* object is concurrently *used* as a new reference value. As defined in the conflict pattern depicted in Figure 6.3, a *delete-use* conflict occurs if an object o has been deleted and the same object has been concurrently inserted as target value in a multi-valued reference or set as target value in a single-valued one. For the model-based representation of such conflicts, we introduce the class DeleteUse in the conflict metamodel (cf. Figure 6.12). This class refers to two conflicting operation elements, namely a DeleteObject by the reference delete and an FeatureOperation by the reference use. Of course, only feature operations of the type InsertFeatureValue or FeatureUpdate are valid because no conflict should be raised if a DeleteFeatureValue is concurrently applied.

A special kind of a *delete-use* conflict occurs when an *inserted* object *uses* a deleted object. More precisely, a user inserts an object, which comprises a reference to another object that has been concurrently deleted. When objects have been inserted, the difference model contains only an operation representing the insertion; it does not contain further elements indicating the operations that have been applied to the inserted object. Thus, we require the additional conflict pattern depicted in Figure 6.4 for detecting such cases. This pattern matches if an object o1 has been deleted (through the DeleteObject instance do) and a feature operation fo exists that realizes the insertion of an object o2 having a reference to the deleted object o1. In the



context DeleteUse
inv: self.delete.object = self.use.value **and**
(self.use.oclIsTypeOf(InsertFeatureValue) **or**
self.use.oclIsTypeOf(FeatureUpdate))

Figure 6.3: Delete-Use Conflict



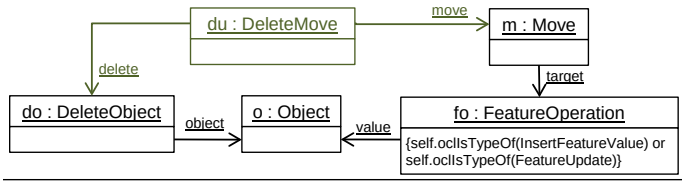
context DeleteUse
inv: InsertObject.allInstances.featureChange->includes(self.use) **and**
self.use.value.eCrossReferences->includes(self.delete.object)

Figure 6.4: Delete-Use Conflict: Through Addition

OCL invariant, we make use of the EMF-specific method called eCrossReferences returning all objects that are referenced through a non-containment reference. We do not have to consider containment references in this pattern, because if the deleted object has been added to a containment reference (i.e., it has been *moved* to the inserted object), the next conflict pattern called *delete-move* matches.

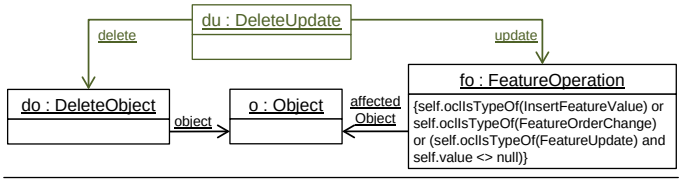
Delete-move conflict. Another special kind of a *delete-use* conflict is a *delete-move* conflict occurring if the feature operation representing the *use* in a *delete-use* conflict (fo in Figure 6.3) is part of a Move (cf. reference target in Figure 5.9). As a result, moving an object and concurrently deleting the same object is indicated as *delete-move* conflict. Therefore, we introduce the class DeleteMove in the conflict metamodel as a subclass of DeleteUse.

Delete-update conflict. A *delete-update* conflict occurs, if a model element is updated (i.e., a modification of the model element's reference or attribute values) and concurrently deleted. However, we refrain from raising a conflict, if the feature update is not a DeleteFeatureValue because in this case both operations may easily be merged without omitting the effect of one of the involved operations. Correspondingly, a FeatureUpdate setting a single-valued feature to *null* should also not cause a conflict when applied to an object deletion in parallel. Consequently, as illustrated in the conflict pattern and the OCL invariant depicted in Figure 6.6, a *delete-update* conflict occurs if an object o has been deleted and the same object as been updated by either an InsertFeatureValue or a FeatureOrderChange, or, in case of a single-valued feature,



context DeleteMove
inv: self.delete.object = self.move.target.value **and**
(self.move.target.oclIsTypeOf(InsertFeatureValue) **or**
self.move.target.oclIsTypeOf(FeatureUpdate))

Figure 6.5: Delete-Move Conflict



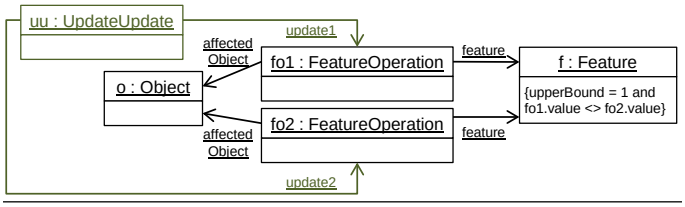
context DeleteUpdate
inv: self.delete.object = self.update.affectedObject **and**
(self.update.oclIsTypeOf(InsertFeatureValue) **or**
self.use.oclIsTypeOf(FeatureOrderChange) **or**
(self.oclIsTypeOf(FeatureUpdate) **and** self.update.value <> null))

Figure 6.6: Delete-Update Conflict

a FeatureUpdate as long as the updated value is not *null*. Please note that this pattern also raises a conflict, if an object is moved to another container object that has been concurrently deleted, because the target container is updated by the *target feature operation* of the *move*.

Update-update conflict. As already mentioned, features in EMF models may be single-valued or multi-valued. If they are single-valued, setting a new feature value will overwrite the old one. If now a single-valued feature is concurrently modified in EMF models obviously a conflict occurs, because the merged model may not contain both values of both users at the same time. Therefore, we raise a conflict, which is referred to as *update-update* conflict. Consequently, as illustrated in the conflict pattern in Figure 6.7, an *update-update* conflict is raised, if an object o has been concurrently updated at the same feature f by two instances of FeatureOperation, fo1 and fo2, unless both operations set the same new value such that `fo1.value = fo2.value`.

Update-update conflict: ordered features. We may also encounter conflicts between concurrent operations, if the updated feature is defined to be *multi-valued* and *ordered* in the modeling language's metamodel. Basically, when merging concurrent operations of an ordered feature, an adequate conflict detection strategy depends on whether it is the absolute index that conveys the real-world meaning of the object's position, or whether the meaning is conveyed by the object's predecessor and successor. This is obviously specific to the respective modeling language.



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

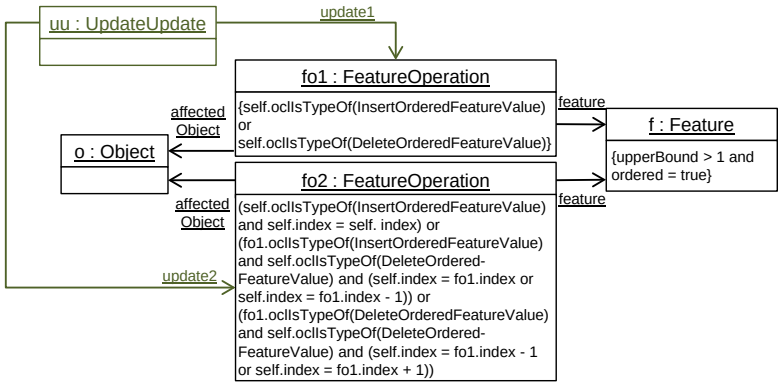
Figure 6.7: Update-Update Conflict

Based on our experiences, we argue that in most of the modeling languages, the meaning of an object's position in ordered features is based on its predecessors and successors. For instance, messages in UML Sequence Diagrams are normally not characterized by their absolute index, but as message before and after other messages. However, there might also be modeling languages, in which an object's index matters more than its predecessors and successors.

If the meaning of an object's position is conveyed by its index, merging concurrent operations to the order is particularly hard to automate. For instance, assume that we have three objects {A, C, E} in an ordered feature in the origin model. Now, developer 1 inserts a new object B at index 2 (i.e., after A) leading to the new ordered set of objects {A, B, C, E} and developer 2 inserts a new object D at index 3 (i.e., after C) so that the ordered set {A, C, D, E} is obtained. Assuming that the index is of major importance in the respective language feature, we have to respect that by retaining the intended indices while merging. Thus, we first insert the object B at index 2 as intended by developer 1. Now, however, the question arises whether to insert object D, added by developer 2, at the index 3 or whether we shift the index because of the previous insertion of B. If we choose to insert it at index 3, as it was originally intended by developer 2, we respect the index, however, we change the originally intended predecessor and successor of D, because we thereby obtain the ordered set {A, B, D, C, E}. On the contrary, if we decide to shift the index by *one* to the right instead, the predecessor and successor would remain as originally intended, however, we would change its index from 3 to 4 with the thereby obtained result {A, B, C, D, E}. Moreover, the index of E has been changed by both users from 3 to 4 and, caused by the merge, this object now resides on index 5. Recalling that the index has an impact on the meaning, this merge has potentially affected the meaning of E in an unintended way.

Listing 6.1: OCL Pattern for Conservative Update-update Conflict Detection

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




```
context UpdateUpdate
inv:  self.update1.affectedObject = self.update2.affectedObject and
      self.update1.feature = self.update2.feature and
      self.update1.feature.upperBound > 1 and
      self.update1.feature.ordered = true and
      (self.update1.oclIsTypeOf(InsertOrderedFeatureValue) or
      self.update1.oclIsTypeOf(DeleteOrderedFeatureValue)) and (
      (self.update2.oclIsTypeOf(InsertOrderedFeatureValue) and
      self.update2.index = self.update1.index) or
      (self.update1.oclIsTypeOf(InsertOrderedFeatureValue) and
      self.update2.oclIsTypeOf(DeleteOrderedFeatureValue) and
      (self.update2.index = self.update1.index or
      self.update2.index = self.update1.index - 1)) or
      (self.update1.oclIsTypeOf(DeleteOrderedFeatureValue) and
      self.update2.oclIsTypeOf(DeleteOrderedFeatureValue) and
      (self.update2.index = self.update1.index - 1 or
       self.update2.index = self.update1.index + 1)))
```

Figure 6.8: Update-Update Conflict: Ordered Features

Thus, we conclude that, if an object's index conveys its meaning in the respective modeling language, it is safer to report a conflict for *any* concurrent modification of the ordered feature of the same object. If, however, the meaning of an object's position is constituted by its predecessor and successor, we may apply a more liberal approach. Therefore, we decided to introduce an adaptation point in the implementation of our conflict detection approach. This adaptation point enables users to configure the conflict detection to either raise an *update-update* conflict for *any* concurrent modification of ordered features or to apply a more liberal conflict detection approach. For realizing the former strategy, the constraint in the feature object f, representing the concurrently modified feature in Figure 6.7, is changed such that the pattern matches for any concurrent modification, fo1 and fo2, of an ordered feature f in the same object o (cf. Listing 6.1 for the respective OCL pattern). If the user, on the other hand, prefers the more liberal approach, which is also the default configuration in our implementation, we employ a dedicated conflict rule as discussed in the following.

Our liberal approach of merging ordered features builds upon the principle that the meaning of an object's position is constituted by its predecessor and its successor. Thus, we aim at raising a conflict if and only if the final value order in the concurrently modified feature cannot be clearly determined or if one operation contradictorily affects the predecessor or the

successor of a concurrently inserted, deleted, or reordered object. In EMF, the order of values in ordered features is, however, represented by absolute indices that are assigned to each value in the ordered list and not by linked lists. Consequently, if a new value is inserted into an ordered list, the indices of all subsequent values are increased by one. To enable directly working with EMF's representation of ordered lists, we also have to detect conflicts regarding contradicting predecessors and successors based on their indices. Therefore, we introduce a dedicated conflict pattern, which is depicted in Figure 6.8. This conflict pattern comprises two feature operations, fo1 and fo2, which both modify the same object o at the same ordered feature f. The object fo1 is an instance of either InsertOrderedFeatureValue or DeleteOrderedFeatureValue. As further specified in the constraint for the object fo2, this conflict pattern matches if one of three particular scenarios occur: *(i)* both operations are instances of InsertOrderedFeatureValue and the *same* index is concurrently modified; thus, the final order cannot be clearly determined. *(ii)* fo1 is an instance of InsertOrderedFeatureValue, fo2 is an instance of DeleteOrderedFeatureValue, and the predecessor (`insertion.index−1=deletion.index`) or successor (`insertion.index=deletion.index`) of the inserted value is modified by the deletion. *(iii)* Both, fo1 and fo2 are instances of DeleteOrderedFeatureValue and the predecessor (`deletion1.index=deletion2.index−1`) or successor (`deletion1.index = deletion2.index+1`) is concurrently affected when both deletions are merged. Please note that, thereby, deleting the same index will *not* cause a conflict.

To summarize, an update-update conflict is raised for ordered features if the order of two inserted objects cannot be clearly determined or if the predecessor or the successor of a value in the merged model would differ from the predecessors and successors in the respective revised models. Please note that feature order operations are realized by a deletion and subsequent insertion of the same object at a different index; consequently, the conflict pattern in Figure 6.8 also addresses conflicting feature order operations.

Example for concurrent modifications of an ordered feature. In Figure 6.9, we show four exemplary scenarios for concurrent operations, which are applied to ordered features. In the first scenario (cf. Figure 6.9a), both users insert a new object to the beginning of the ordered feature so that we may not automatically decide whether to place the one or the other inserted object at index 1. As both operations insert an object at index 1, the conflict pattern (cf. Figure 6.8) matches and a conflict is reported. In the second scenario (cf. Figure 6.9b), developer 1 deletes the first object A and developer 2 inserts a new object B at index 2 (i.e., after A). Following our principle that the predecessor and successor of an ordered object conveys the real-world meaning, we conclude that developer 2 intended the inserted object B to be placed right after A; this object, however, has been concurrently deleted. Therefore, according to the conflict pattern, a conflict is reported, because object A has been deleted from index 1 (i.e., fo2 in the conflict pattern) and object B has been inserted at index 2 (i.e., fo1 in the conflict pattern) so that the constraint `fo2.index=fo1.index−1` is fulfilled. The third scenario (cf. Figure 6.9c) illustrates the reason for checking the predecessor and successor index only if at least one operation is a deletion (i.e., an instance of DeleteOrderedFeatureValue). In this scenario, both users concurrently insert an object at index 2 and index 3, respectively. We may easily merge this scenario in a unique way without affecting the intended predecessors and successors of the inserted objects

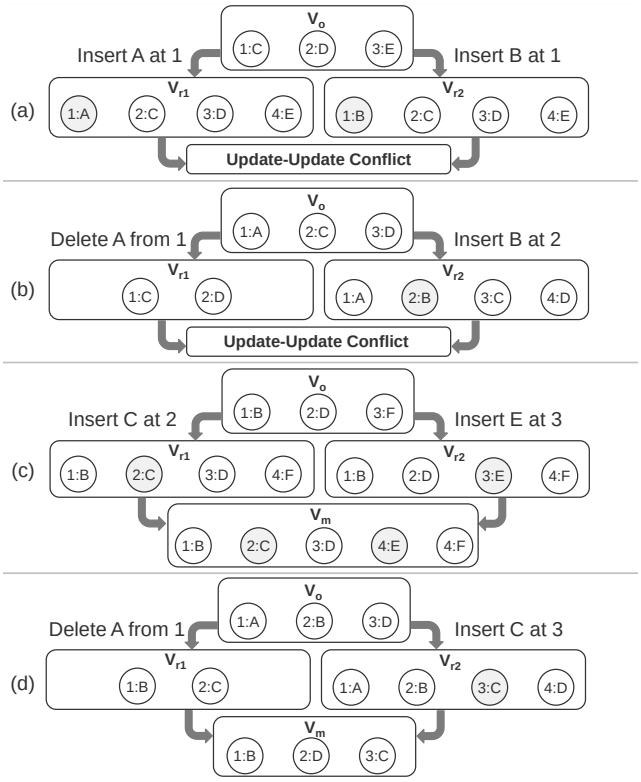
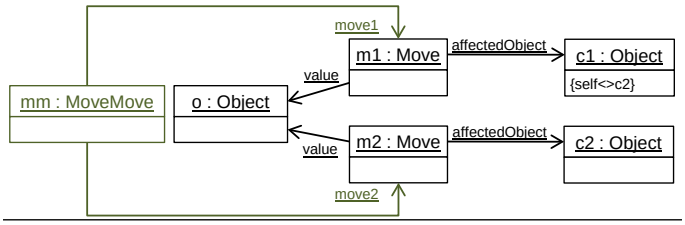


Figure 6.9: Examples for Concurrent Operations Applied to an Ordered Feature

as depicted in V_m of Figure 6.9c. None of both feature operations is a deletion; thus, no conflict is reported because `fo1.index≠fo2.index`. Finally, in our last scenario (cf. Figure 6.9d), developer 1 deletes object A from index 1 and developer 2 inserts a new object C between the objects B and D (i.e., at index 3). Consequently, the conflict pattern does not match and we may safely produce the merged model depicted in V_m of Figure 6.9d.

Move-move conflict. Next, we introduce a special case of an *update-update* conflict, which is related to concurrent updates of containment references of *different objects*, but using the *same object* as value. In particular, such a conflict—denoted as *move-move* conflict—occurs if the same object has been concurrently *moved* to different container objects. This is still an *update-update* conflict because Move is an operation type consisting of two feature updates (cf. Figure 5.9). However, in contrast to the common *update-update* conflicts as defined in Figure 6.7, *move-move* conflicts are not caused by concurrent feature updates of the *same object* but of *different objects*. In particular, a *move-move* conflict occurs if the *same object* o has been concurrently moved to *different* container objects c1 and c2 (cf. Figure 6.10). This pattern basically ensures that every object in an EMF model has at most one container. As depicted in the conflict metamodel in Figure 6.12, the class MoveMove is a subclass of UpdateUpdate and additionally references two Move elements.



context MoveMove
inv: self.move1.value = self.move2.value **and**
self.move1.affectedObject <> self.move2.affectedObject

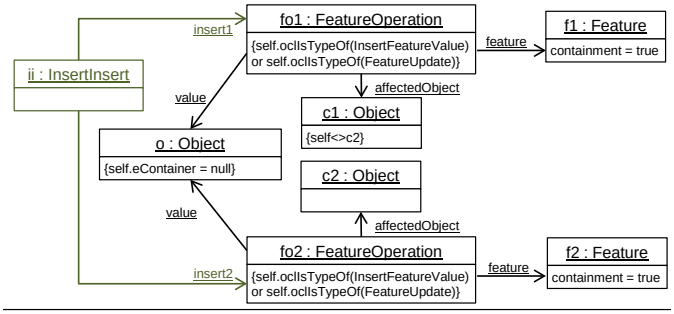
Figure 6.10: Move-Move Conflict: Non-Unique Container

However, because of the specific restriction of EMF specifying that every EMF models must be a spanning containment tree, we also have to avoid cyclic containment relationships. Basically, a containment cycle occurs if developer 1 moves an object to another container object and developer 2 concurrently moves the same container object (or a parent of it) to the object developer 1 moved (or a child of it).

Insert-insert conflict. Finally, we have to regard one special case concerning the containment relationships in EMF. As already mentioned, every object must have at most one container. When considering a scenario, in which one object does not have a container in the origin model and both users concurrently set different containers for this object then no move-move conflict is reported. Therefore, we introduce an *insert-insert* conflict pattern for addressing such a scenario in Figure 6.11. This conflict is raised if the same object o is concurrently inserted or set as feature value of a containment reference f1 and f2 in two different objects c1 and c2. Accordingly, we also introduce the class InsertInsert in the conflict metamodel, which references two FeatureOperations causing the conflict.

Completeness of the conflict patterns. We developed the aforementioned conflict patterns for finding atomic operation conflicts, on the one hand, by reviewing existing literature in the realm of conflict detection for models (e.g., [AP03, KHWH10, SZN04, Wes10]) and, on the other hand, by identifying all possible operation types for EMF-based models in the first step and setting up an *operation matrix* representing the cross product of all operation types leading to a list of all possible combinations of operations in the second step. Going through all possible combinations of operations, we decided for each combination whether it should be reported as conflict or not. Finally, we implemented the resulting list of conflict types in AMOR and conducted several case studies in collaboration with our industry partner SparxSystems[1] (the vendor of the UML tool called Enterprise Architect) to evaluate whether the list of identified conflict types covers a wide range of conflicts occurring in modeling practice.

[1] http://www.sparxsystems.at



context InsertInsert
inv: (self.insert1.oclIsTypeOf(InsertFeatureValue) **or**
self.insert1.oclIsTypeOf(FeatureUpdate)) **and**
(self.insert2.oclIsTypeOf(InsertFeatureValue) **or**
self.insert2.oclIsTypeOf(FeatureUpdate)) **and**
self.insert1.value = self.insert2.value **and**
self.insert1.value.eContainer = null **and**
self.insert1.affectedObject <> self.insert2.affectedObject **and**
self.insert1.feature.containment **and**
self.insert2.feature.containment

Figure 6.11: Insert-Insert Conflict

6.1.2 Technical Realization

Having set up the conflict detection rules discussed above, realizing the conflict detection is largely straightforward. Generally speaking, for all operation combinations of both difference models it has to be checked whether one of the aforementioned conflict patterns matches to indicate a conflict. However, for the sake of efficiency, we refrain from checking the complete crossproduct of all operation combinations among all operations of both difference models. In contrast, both difference models are translated in a first step into an optimized view grouping all operations according to their type into potentially conflicting combinations. Secondly, all combinations are filtered out if they do not spatially affect overlapping parts of the original model. Finally, all remaining combinations are checked in detail by evaluating the previously presented rules.

Conflict model. If one of the conflict patterns presented in the previous section has been matched within two concurrent difference models, a conflict description is created and added to a conflict report. A conflict report is a model-based representation of all conflicting operations in two difference models. Therefore, we use the conflict metamodel depicted in Figure 6.12, which may also serve as a summary of all supported atomic conflict types. Basically, an instance of ConflictReport contains for each occurred conflict an instance of the specific conflict type (e.g., DeleteUse) referring to the two operations (depicted in gray in Figure 6.12) causing the conflict. Thus, the conflict report explicitly indicates the occurred conflicts by providing, for each conflict, the required information on its type and the involved operations in the difference models.

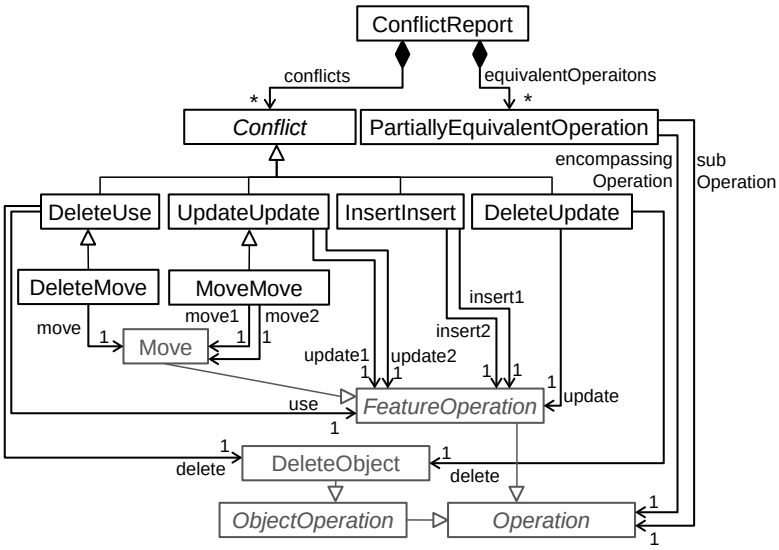


Figure 6.12: Conflict Metamodel

Equivalent operations. Besides the list of conflicts, the conflict report also comprises information on (partially) equivalent operations. By equivalent operations, we refer to operations that are indeed spatially overlapping, but which ultimately have the same affect and, thus, should not be marked as conflicting. For instance, if two developers update the same feature value of the same model element to an equal new value, no conflict should be reported as both operations have an equal effect. In the case of concurrent deletions, these equivalent operations may be only *partially* equivalent. More precisely, two operations are partially equivalent, if one operation deletes a subset of the model elements that have been also removed by a concurrent larger deletion. For instance, developer 1 deletes state *Connecting* from a state machine and developer 2 concurrently removes state *Active*, which contains *Connecting*. Hence, the operation by developer 1 is a partially equivalent sub operation of the deletion of developer 2. This information is important for correctly creating a merged model, because in case of equivalent operations, only one of the partially equivalent operations should be applied to the merged model. Therefore, equivalent operations between the two difference models are described in terms of instances of PartiallyEquivalentOperation in the conflict report, whereas the reference subOperation indicates the operation that partially equivalent to the larger operation referenced through encompassingOperation. Thus, when merging partially equivalent operations, we only apply the encompassing operation and omit the sub operation.

6.2 Composite Operation Conflict Detection

An application of a composite operation is more than its comprised atomic operations, as the comprised atomic operations have been applied all at once to fulfill a common goal reflecting the underlying intention of the developer, who performed the composite operation. The inten-

tion of the developer is only fulfilled, if the entire composite operation is successfully applied to all originally selected model elements. If another developer concurrently changes or adds model elements, the effect of the composite operation might be mitigated. Therefore, considering only atomic operations when detecting conflicts is not enough for several merge scenarios. In this section, we show how such scenarios can be detected. Therefore, we first revisit the categorization of merge issues in the context of composite operations in Section 6.2.1. Subsequently, we give an overview of the applied detection process in Section 6.2.2 and, finally, in Section 6.2.3 and Section 6.2.4, we show how the scenarios presented in Section 3.1.4 and Section 3.1.5 are detected by our approach, respectively.

6.2.1 Categorization of Composite Operation Merge Issues Revisited

Before proceeding with presenting our approach to detecting merge issues that involve applications of composite operations, we briefly recall the categorization of merge issues in this context. As introduced in Section 3.2, we distinguish between two categories of composite operation merge issues.

Composite Operation Conflicts. An *operation-based conflict* denotes two operations that are either parallel dependent or not commutative. Thus, both operations cannot be applied together without nullifying one operation; in other words, overlapping operations halt the merge unless at least one of the overlapping operations is omitted. Besides operation-based conflicts between atomic operations as discussed in Section 5.2, operation-based conflicts may also occur due to violated preconditions of a composite operation. More precisely, if the preconditions of a composite operation that has been performed by one developer are violated after the concurrent operations of another developer have been applied, a *composite operation conflict* occurs. For an example of such a conflict, we kindly refer to Section 3.1.5.

Composite Operation Match Warnings. The effect of a composite operation might be mitigated because the concurrent operations of another developer are not considered in the original application of the composite operation. As already mentioned, composite operation specifications comprise detailed preconditions and the application of a composite operation affects model elements that *match* the preconditions. If concurrent operations applied by another developer modify the model such that this match is influenced, we may either face an operation-based conflict, or we may encounter valid preconditions and an *increase of the match size*. That is, the composite operation application is still valid, however, more model elements than the originally involved ones match the preconditions after the concurrent operations have been applied. Therefore, developers are notified in terms of a *warning* that these additionally matching model elements should also be incorporated in the composite operation application. An example that illustrates such a scenario is presented in Section 3.1.4.

6.2.2 Detection Process at a Glance

The goal of the process for detecting merge issues in the context of composite operations is to identify those applications of composite operations, which are interfered or at least affected by

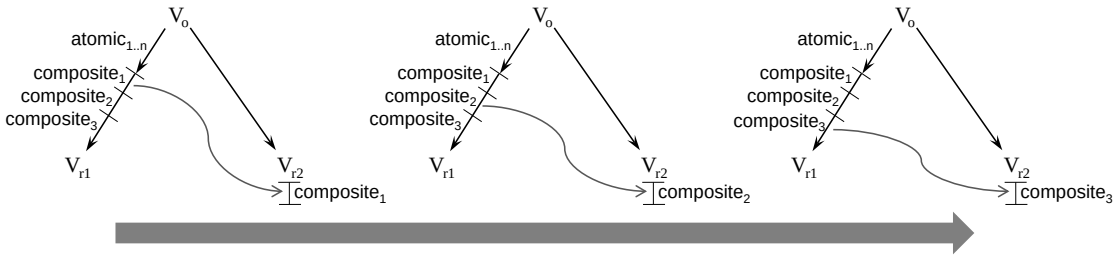


Figure 6.13: One-by-One Evaluation of each Composite Operation Application

concurrent operations. Therefore, each application of a composite operation on one side in a merge scenario (e.g., V_{r1}), is separately checked for conflicts and affected precondition matches with the operations applied on the respective opposite side (e.g., V_{r2}, or vice versa). As we have to evaluate the preconditions of composite operation applications, we may not only consider the concurrently applied operations, but also the opposite *state* of a model resulting from the concurrent operations. Thus, we extract each application of a composite operation from the difference model on one side and try to apply the same composite operations to the corresponding model elements on the opposite revised model as depicted in Figure 6.13. In this figure, three composite operations denoted with composite$_1$, composite$_2$, and composite$_3$ have been applied between V_o and V_{r1} alongside some other atomic operations atomic$_{1..n}$. Next, we iterate through all of these composite operation applications one-by-one, and separately check whether they can be applied to the revised opposite model V_{r2}. Of course, in case composite operations have been applied also on the opposite side V_{r2}, we subsequently have to repeat the same process the other way round by checking the composite operation applications originally applied in V_{r2} with the opposite model V_{r1}. The concrete steps that are necessary for checking the applications of composite operations with the opposite revised model are depicted in Figure 6.14 and discussed in the following.

Extract composite operation applications. In a first step, all composite operations are extracted from the difference model. This is easily realized by selecting all instances of type CompositeOperationApplication (cf. Section 5.3.4) from the difference model. For each extracted composite operation application, the following steps are repeated.

Besides detecting conflicts, we also aim at detecting an increased match size at the opposite side concerning the composite operation's precondition templates. Therefore, we have to compare the *original match size* with the *opposite match size*. As discussed in Section 5.3.4, an instance of a CompositeOperationApplication represents the precondition binding in terms of an instance of a ConditionModelBindingCollection, which indicates the model elements that have been affected by the composite operation application. This binding, however, only refers to the model elements that actually have been transformed. Nevertheless, the user, who applied the composite operation, might have purposefully *deselected originally matching model elements* before applying the composite operation; thus, the actual match might have been larger before the user originally applied the operation. To avoid reporting an increased match size when the

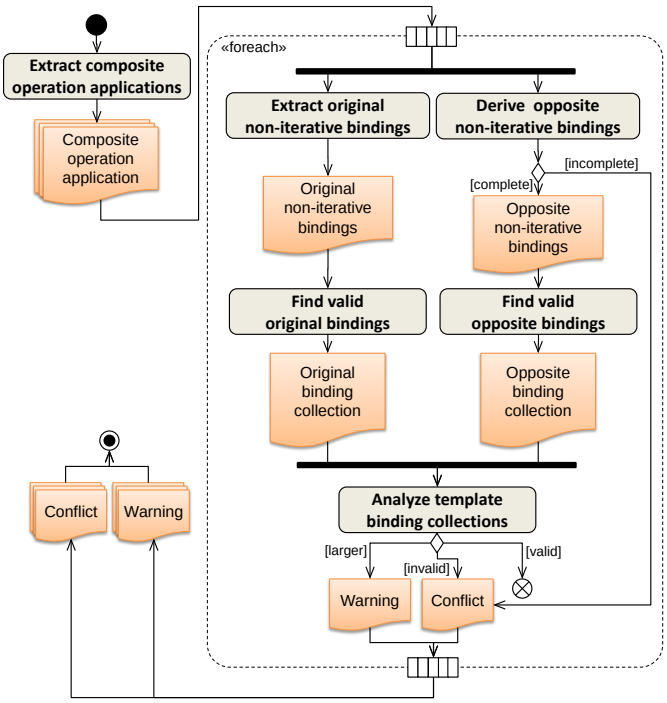


Figure 6.14: Process of Composite Operation Merge Issue Detection

original match size has already been larger and has been purposefully decreased by the user before the operation has been originally applied, we have to take *all* originally matching model elements into account for this comparison. Thus, in the following two steps, we first compute the original match size (i.e., the original precondition binding) and, subsequently, we derive the precondition binding for the opposite side. In the last step, we compare these two bindings to detect conflicts and to potentially raise an increased match size warning.

Extract original non-iterative bindings. To obtain all originally matching model elements, we first extract all *non-iterative bindings* from the precondition bindings, which are saved in the instance of the CompositeOperationApplication. By non-iterative bindings, we refer to those bindings of model elements bound to *non-iterative templates*. Therefore, we remove all template bindings of iterative templates so that the resulting original non-iterative bindings (cf. Figure 6.14) contain only the fixed bindings of model elements in the origin model to non-iterative precondition templates. Please note that also child templates of iterative templates are not included in the fixed bindings as they depend on their iterative parent templates.

Find valid original bindings. For finding all originally matching model elements, we apply the template matching engine, which has been discussed in Section 4.1.4. Thereby, we use the original non-iterative bindings extracted before as input binding for the template matching process. In this way, the template matching engine obtains *all valid bindings* for iterative templates

based on the original non-iterative bindings. As all non-iterative templates are already bound in the original non-iterative bindings (i.e., they are fixed), the resulting original binding collection (cf. Figure 6.14) is always *unambiguous* and only extends the input binding by valid bindings for iterative templates.

Having obtained all matching model elements for the original composite operation application, we may proceed with evaluating the same composite operation in the context of the opposite side. Therefore, in the following two steps, we first derive the non-iterative bindings for the opposite revised model and evaluate them.

Derive opposite non-iterative bindings. The extracted composite operation application has been originally applied to the origin model V_o. Thus, the precondition template binding, which is contained by the extracted composite operation application, refers to the respective model elements in the origin model V_o. As we aim at checking whether the same composite operation is also applicable to the corresponding model elements in the opposite revised model (either V_{r1} or V_{r2}), a new template binding for the respective revised model has to be derived first. For this task, the match model from the respective side ($M_{V_o,V_{r1}}$ or $M_{V_o,V_{r2}}$) is used. More precisely, a copy of the precondition template binding is created first. Subsequently, for each model element contained in the origin model (i.e., V_o) that is referenced from the precondition template binding, the corresponding model element in the revised model is obtained using the match model and put in the place of of the corresponding original model element. Thus, the resulting binding ultimately refers only to model elements of the opposite revised model in the place of the model elements from V_o. Thereby, as in the step for extracting the original non-iterative bindings, we extract only *non-iterative template bindings*. If for a bound model element of the original model no corresponding model element in the revised model could be found based on the match model, it obviously has been deleted in the opposite revised model, which causes the derived binding to be incomplete. Thus, we may immediately stop to conduct further analysis and directly report a composite operation conflict (cf. Figure 6.14). If, however, every model element from the original model has a corresponding model element in the revised model according to the match model, the resulting opposite non-iterative bindings (cf. Figure 6.14) are complete. That is, they contain all fixed template bindings, which map model elements in the *revised model* to the non-iterative precondition templates of the composite operation.

Find valid opposite bindings. After the fixed precondition template binding has been derived for the opposite revised model, we again use the template matching engine to obtain *all valid template bindings* based on the specified opposite non-iterative bindings. Thus, the resulting opposite binding collection (cf. Figure 6.14) contains all matching model elements in the opposite revised model. If the template matching engine could not find a valid template binding at all, the opposite binding collection is empty; nevertheless, it includes an evaluation report, which gives information on the failed conditions and on the model elements that violate the respective conditions.

Analyze template binding collections. In this step, the original binding collection and the opposite binding collection are analyzed for detecting conflicts and warnings. First of all, it is checked whether the opposite binding collection is valid. That is, all templates are bound to *at least one* matching model element. If this is not the case, a *composite operation conflict* is raised, because the preconditions of the composite operations are not fulfilled after the opposite operations have been applied; thus, the composite operation is not applicable anymore. If, on the contrary, the opposite binding collection is valid, we may still encounter a composite operation conflict, unless *every model element* that has been bound in the original application of the composite operation is also bound in the opposite binding collection. In other words, a composite operation conflict is also raised, if the opposite binding collection indeed contains at least one valid binding for every template (i.e., the binding collection is valid), but at least one model element that has originally been bound to an *iterative template* does not fulfill the preconditions anymore (after the opposite operations have been applied). Consequently, the composite operation would be only partially applicable to the opposite revised model. Therefore, we compare the set of model elements that are bound to iterative templates in the original binding collection with the set of model elements that are bound in the opposite binding collection. If at least one model element binding is missing, a conflict is raised.

Unless a composite operation conflict has been reported, we proceed with examining whether the opposite binding collection binds *more model elements* than in the original composite operation application. In this case, a warning is issued to notify the developers that, when merging the application of the composite operation, it potentially should also incorporate additional model elements (i.e., more model elements than in the original application). Therefore, we check whether the opposite binding collection binds model elements that are *not bound* in the original binding collection. If there additional model elements, a composite operation match warning is raised. Please note that the following two steps are not explicitly represented in Figure 6.14 to avoid crowding the figure.

Deriving the conflicting operation. If a composite operation conflict has been detected, it is helpful for resolving this conflict to know the opposite operation that actually causes the preconditions of the composite operations to be violated. Deriving the conflicting operation, however, is a difficult problem, because the potentially complex preconditions are evaluated against *the revised state* of a model. Hence, the only information that we have at the time of the evaluation is that a particular condition is violated by a specific model element. Thus, for deriving the causing operation, we first require a deep understanding of the respective condition to infer the actual *reason* for the condition to be violated by a specific model element. If we know the reason, we may search for the opposite operation that modified the model element such that the condition is violated. A universal solution to this problem is hard to find and goes beyond the focus of this thesis. However, we implemented a solution for basic scenarios. This solution is capable of detecting the causing operation if the violated condition is a *feature condition* (cf. Section 4.1.4) or if a *model element is missing* in order to create a valid binding.

As *feature conditions* restrict the value of a model element for a specific feature, we may, in case a feature condition is violated, search for an opposite operation that modifies exactly the same feature at the invalid model element. If such an operation can be found, we may assume

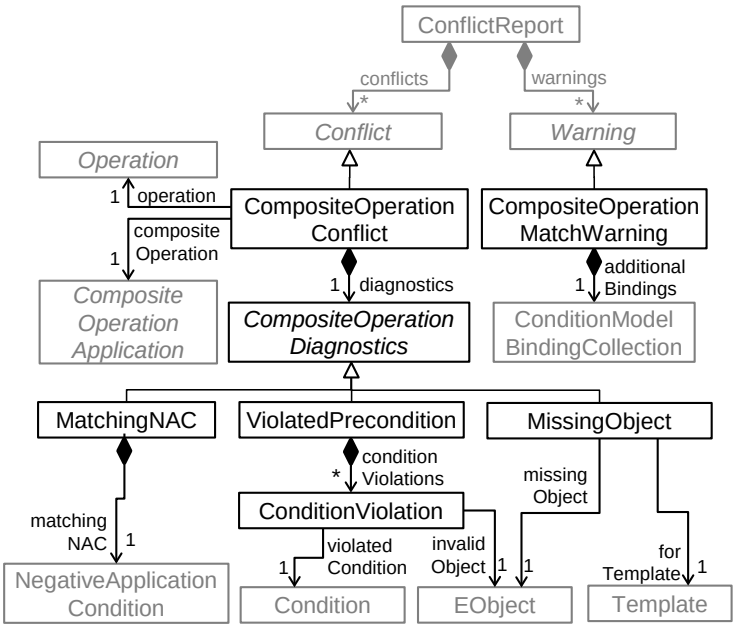


Figure 6.15: Metamodel for Composite Operation Merge Issues

that this is the operation that causes the respective condition to be violated. In case, we cannot find an operation that directly modified the respective feature of the invalid model element, we check whether the condition refers to another template and search for an operation, which modified the model element that is bound to the referenced template. For instance, if we have the condition `name = #{Transition_1}.name` in the template Transition_2 and no operation directly modified the name of the model element bound to Transition_2, we search for an operation that modified the name of the model element bound to Transition_1. Our implementation of this approach, however, covers only for basic conditions without logical conjunctions.

If we encounter a *missing model element* that is necessary for the preconditions to be valid, we simply search for an operation that deleted the previously matching model element and, in case we found such an operation, assume that this is the operation causing the conflict with the current composite operation application.

Representing composite operation conflicts and warnings. As already mentioned (cf. Figure 6.1), raised composite operation conflicts and warnings are added to the conflict report. Therefore, we extend the metamodel for conflicts as depicted in Figure 6.15. Please note that classes from already presented metamodels are painted in gray in this figure. For representing composite operation conflicts and warnings, we added the classes CompositeOperationConflict and CompositeOperationMatchWarning, respectively.

Instances of CompositeOperationConflict are further described by the abstract class CompositeOperationDiagnostics. Instances of this class provide further information on the actual reason for the raised composite operation conflict. In particular, a reason for such a conflict

is either a matching negative application condition, a violated precondition, or a missing object. Therefore, the metamodel contains dedicated classes to allow for describing each of these reasons. Instances of the class MatchingNAC refer to the operation specification's negative application condition, which could be matched with the opposite revised model and, consequently, prohibits the execution of the composite operation after the concurrent operations have been applied. The class ViolatedPrecondition represents scenarios in which a model element from the opposite revised model violates the precondition. Therefore, for each violated precondition, an instance of ConditionViolation is created, which refers to the invalid model element and the condition that is violated. If the application of a composite operation is interfered by the concurrent deletion of an involved model element, an instance of MissingObject is created, which refers, on the one hand, to the missing model element and, on the other hand, to the template to which it originally was bound.

Warnings for indicating an increased match size of composite operations are represented by the class CompositeOperationMatchWarning. Instances of this class refer to an instance of ConditionModelBindingCollection, which contains all additionally matching model elements that potentially need to be involved when merging the application of the composite operation with the concurrently applied operation from the opposite revised model.

6.2.3 Composite Operation Conflict Detection in Action

To exemplify the process for detecting composite operation conflicts, we show how the composite operation conflict occurring in the scenario presented in Section 3.1.5 is detected. The model versions of this model versioning scenario, the template bindings of each step in the detection process, and an excerpt of the object representation of the detected conflict is depicted in Figure 6.16. Please note that we omit the steps for computing the original template bindings, as these steps are only relevant for raising warnings concerning the match size and not for detecting composite operation conflicts. In the following, we go through the detection process for this scenario.

Derive opposite non-iterative bindings. The complete original binding is depicted in the upper part of Figure 6.16. In the first step, we derive the opposite non-iterative bindings. Therefore, a copy of the original binding is created. In this copy, each reference to the original model V_o is replaced by a reference to the original model element's corresponding model element in V_{r2} according to the match model, whereas only bindings to non-iterative templates are retained. The resulting opposite non-iterative bindings are depicted in the right part of Figure 6.16 and only contains bindings to the states Idle and DialTone, as well as to the transitions lift and abort (i.e., the transition, which was renamed in V_{r2} from hangup). The state Dialing and the transition named hangup (connecting Dialing and Idle) are not bound to a template because the template SingleState_2 is iterative and the template Transition_2 is a child of the iterative template SingleState_2.

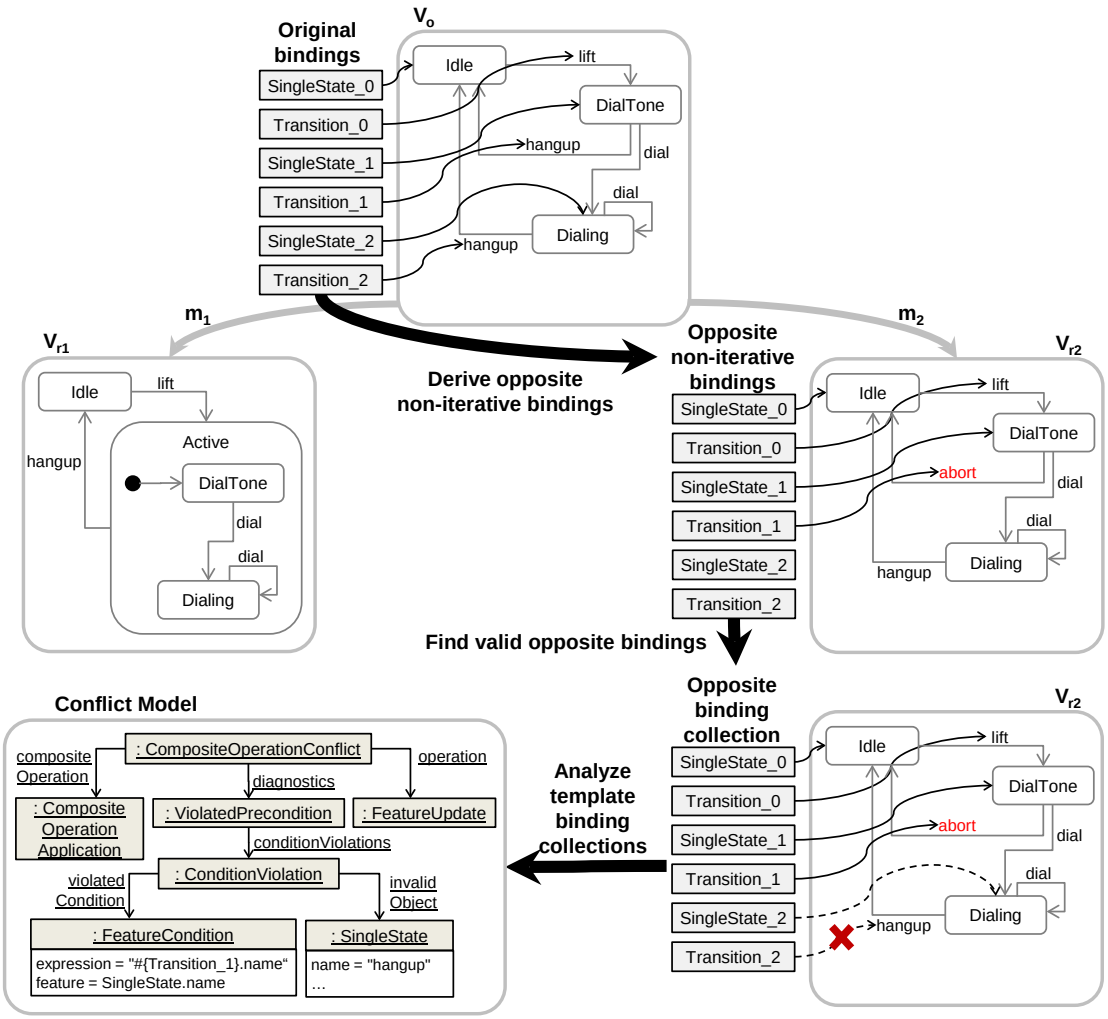


Figure 6.16: Example for Detecting Composite Operation Conflicts

Find valid opposite bindings. Next, the template matching engine is used to find all valid opposite bindings based on the opposite non-iterative bindings obtained in the last step. In this scenario, however, the engine cannot find a valid binding for the templates SingleState_2 and Transition_2 because the condition of Transition_2, which restricts the value of feature name to be equal to the value of the same feature in the model element bound to Transition_1, is violated for the transition hangup contained by the state Dialing (cf. opposite binding collection in Figure 6.16). Consequently, also the state Dialing does not match with the template SingleState_2, because this templates dictates matching model elements to contain a transition that matches to Transition_2, which is not the case in this model. Therefore, the template matching engine returns an empty opposite binding collection and provides further information on the violated conditions and invalid model elements in terms of an evaluation report.

Analyze template binding collections. In this step, the opposite binding collection is analyzed. As this binding collection is empty (i.e., no valid binding could be found for every template), a composite operation conflict is raised. Therefore, the opposite operation that has been applied between V_o and V_{r2} causing the conflict is derived and an instance of the class CompositeOperationConflict for describing the conflict is created. These two steps are described in the following two paragraphs in more detail.

Deriving the conflicting operation. The goal of this step is to derive the opposite operation that causes the preconditions of the composite operation to be violated. This is done by analyzing the evaluation result coming from the template matching engine. In our scenario, the template matching engine reports that the feature condition `name = #{Transition_1}.name` in the template Transition_2 is violated with the transition hangup contained by the state Dialing. Thus, we first search for an opposite operation that directly affects the feature name of the invalid transition. As no operation can be found that affects this feature, the condition is further analyzed to find out that the condition queries the template Transition_1 at the feature name. Hence, an operation is searched that affects the value of the feature name in the model element currently bound to Transition_1. Fortunately, there is such an operation; thus, it is concluded that this operation causes the composite operation conflict.

Representing the composite operation conflict. Finally, an instance of CompositeOperationConflict is created and added to the conflict model in order to describe the merge issue for the subsequent steps in the merge process. In the lower left part of Figure 6.16 (cf. Conflict Model), the representation of the conflict is depicted in terms of an object diagram. As the conflict was caused by a violated precondition, the diagnostics of the conflict is described by an instance of the class ViolatedPrecondition, which contains an instance of ConditionViolation for indicating the violated condition and the model element that fails to fulfill this condition. Please note that there might be more than one violations for one conflict, which is, however, not the case in this scenario.

6.2.4 Composite Operation Warnings in Action

In this section, we show how the example presented in Section 3.1.4 is solved by the proposed approach. Therefore, in Figure 6.17, we show the template bindings of each step in the detection process and an excerpt of the object representation of the raised warning. For the sake of readability, we omitted to depict the revised model V_{r1}, because it is equivalent to V_{r1} in Figure 6.16.

Extract original non-iterative bindings. To obtain all valid template bindings of the applied composite operation in the original model, the original non-iterative bindings are extracted. This is easily done by creating a copy of the original bindings and remove all bindings to iterative templates, which are, in this scenario, the templates SingleState_2 and Transition_2 (cf. Figure 6.17).

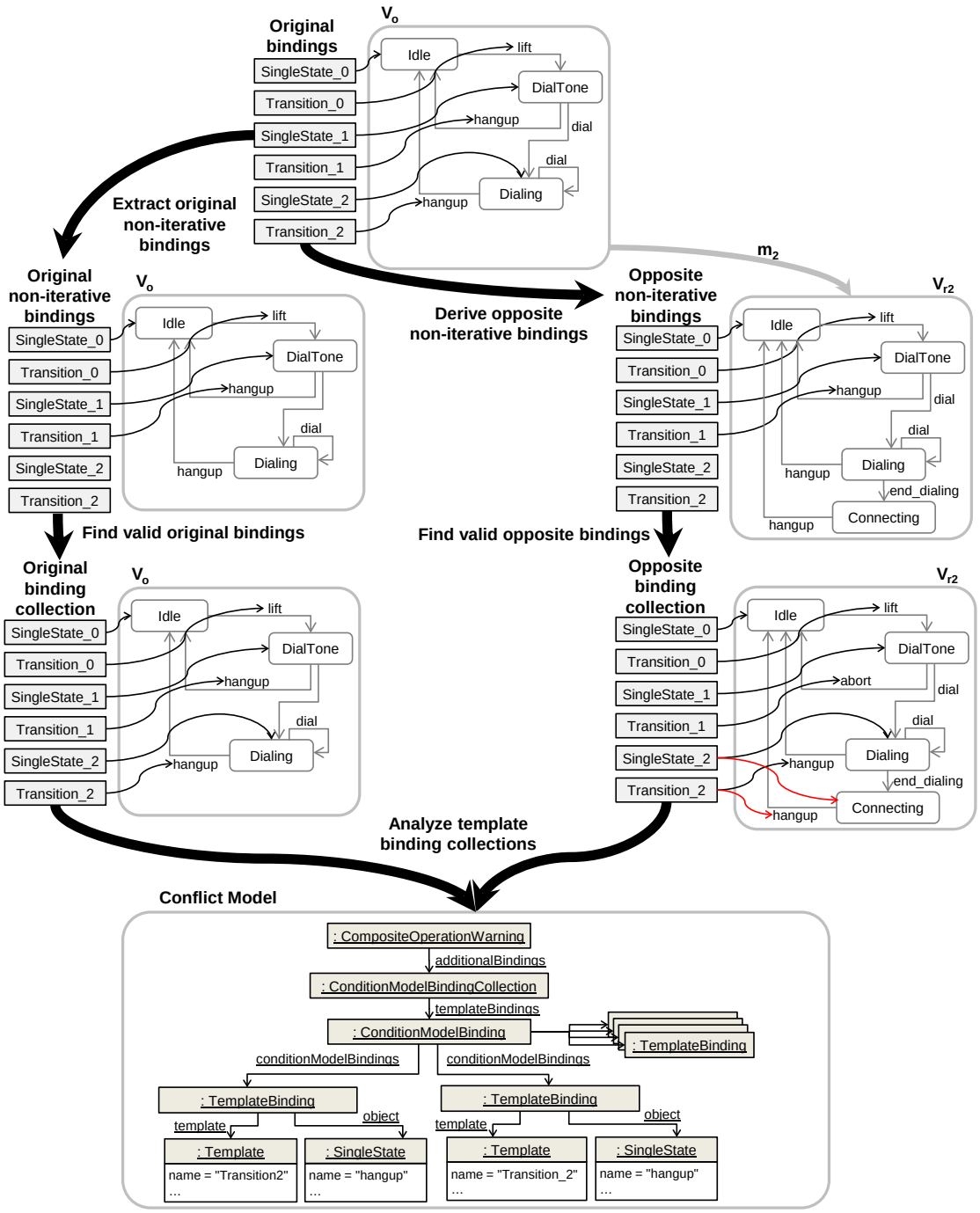


Figure 6.17: Example for Detecting Composite Operation Warnings

Find valid original bindings. In this step, the template matching engine is used for finding all valid bindings in the original model based on the previously created original non-iterative bindings. However, as the user did not manually deselect matching model elements, the resulting binding is equivalent to the original binding. Nevertheless, it is important to be aware of manually deselected bindings to avoid unnecessary composite operation match warnings for model elements that have been matching in the original model already and that have been purposefully deselected by the user.

Derive opposite non-iterative bindings. In the next step, the original binding is rewritten for the opposite revised model V_{r2}, whereas retaining only bindings to non-iterative templates. As already mentioned, the templates SingleState_2 and Transition_2 are omitted in the opposite non-iterative bindings (cf. Figure 6.17), as SingleState_2 is iterative and Transition_2 is a child of this iterative template.

Find valid opposite bindings. Having derived the opposite non-iterative bindings, we may apply the template matching engine to obtain all valid bindings in the opposite revised model V_{r2}. As depicted in the opposite binding collection in Figure 6.17, we thereby obtain additional valid bindings. More precisely, the template SingleState_2 is additionally bound to the newly created state Connecting and the template Transition_2 is bound to the newly created transition hangup contained by the state Connecting.

Analyze template binding collections. In the first step, the validity of the opposite binding collection is checked. As the binding collection is valid (i.e., every template is bound to at least one model element), we may proceed with comparing the original binding collections with the opposite binding collection. Thereby, it is checked whether the original binding contains bindings to model elements that are not bound in the opposite binding collection. In our scenario, this is not the case; thus, no composite operation conflict occurred. Next, the binding collections are compared the other way round; it is checked, whether there are any model elements bound in the opposite binding collection that are not bound in the original binding collection. As already mentioned, in this scenario this is the case for the state Connecting and its contained transition hangup. Consequently, the developers have to be notified that there are two model elements that potentially should be incorporated in the application of the composite operation when creating the merged model. Thus, a dedicated object representing this merge issue is added to the conflict model as described in the following.

Representing the composite operation warning. For describing an increased composite operation match size, an instance of the class CompositeOperationMatchWarning is created and added to the conflict model (cf. Conflict Model in Figure 6.17). The additional model elements that might be incorporated in the application of the composite operation are represented in terms of an instance of the class ConditionModelBindingCollection, which contains one instance of ConditionModelBinding. Please recall that one condition model binding always contains one complete intrinsically valid set of distinct one-to-one relationships between templates and model elements, whereas each template of the condition model is bound to exactly one model element

and one model element is bound only once (cf. Section 4.1.4). Thus, in our scenario, this binding collection contains, as expected, the new bindings for the state Connecting and its contained transition hangup alongside the one-to-one bindings of the remaining templates. Based on this model, we may provide all required information to the developers to decide whether they prefer to also incorporate these two model elements when reapplying the composite operation for obtaining the merged model.

6.3 Signifier Warning Detection

Having obtained atomic and composite operation conflicts in the previous steps, in the next step called signifier warning detection, we aim at detecting *unexpectedly matching model elements* and inadvertent *concurrent changes to a model element's signifier*. For detecting both of these types of merge issues, we apply user-specified language-specific *signifier specifications*. As already mentioned, with the term signifier, we refer to a combination of specific features of a model element type, which convey the *superior meaning* of its instances (e.g., the name, the return type, and the parameters of a UML operation). The importance of considering these signifiers in the merge process is illustrated in the versioning scenarios presented in Section 3.1.1, Section 3.1.2, and Section 3.1.3. Before we present our approach for detecting merge issues in the context of signifiers, we first briefly recall the categorization of signifier-based merge issues (cf. Section 3.2).

Unexpected signifier match. An unexpected signifier match indicates scenarios in which two model elements, which have either been added or modified, have the same signifier eventually; that is, they share the same meaningful characteristics. If these two model elements are completely equal as in the scenario in Section 3.1.1, we may safely suggest to remove one of those added or modified model elements to avoid redundancies in the merged model. If, however, the model elements indeed have the same signifier, but are not entirely equal, a decision of the developer is needed to verify if both model elements should be retained or how they should be joined (cf. Section 3.1.2 for an example).

Concurrent signifier change. Besides concurrent operations, which are directly conflicting, we may also encounter two operations, which are not overlapping in any kind, but which both modify the same model element such that its superior meaning is contradictorily affected. In such scenarios, which are referred to as *concurrent signifier change*, it is likely that the model element's meaning is obfuscated and, therefore, developers should be warned and review the merged model. An example for such a scenario is given in Section 3.1.3.

Current model versioning systems do not regard signifiers during the merge process (cf. Section 2.1.2). To address this deficiency, we present a dedicated detection technique for finding signifier-related merge issues. More precisely, we introduce an adaptation point allowing users to plug in language-specific signifier specifications for the modeling languages they use. Thus, the input of our approach for detecting signifier merge issues is a set of user-specified *signifier specifications*, the origin model V_o and two revisions of it V_{r1} and V_{r2} as well as two difference

models, $D_{V_o,V_{r1}}$ and $D_{V_o,V_{r2}}$, which contain the applied operations for these two revisions. We elaborate on how signifier specifications are represented in Section 6.3.1. Nevertheless, we first outline the three steps used for detecting signifier merge issues and point to the section, in which we present the respective step in more detail.

Signifier specification preprocessing. Before searching for *unexpected signifier matches* and *concurrent signifier changes*, we preprocess the user-specified signifier specifications. The goal of this preprocessing is to make the metamodel features that are used for computing the signifier of a model element more explicit. The thereby obtained explicit representation of the signifiers' *visited* metamodel features, called match triggers, allows us to prune the set of model elements to be considered. That is, only the signifiers of those model elements need to be matched that have been modified in one of the revisions such that their signifier potentially changed. This preprocessing is discussed in more detail in Section 6.3.2.

Unexpected signifier match detection. For detecting unexpected signifier matches, we have to compare the model elements from V_{r1} and V_{r2} with each other. Thereby, we only have to consider model elements of the same type, which have been inserted or modified in the course of the revision according to the match triggers. If a model elements in V_{r1} matches with a model element in V_{r2} and this match is not reflected in the match model; that is, they do not correspond to the same origin model element, we raise an *unexpected signifier match* warning. This technique is more precisely presented in Section 6.3.3. In this section, we also show how the merge issues in the merge scenario presented in Section 3.1.1 and Section 3.1.2 are accordingly detected.

Concurrent signifier change detection. The detection of concurrent signifier changes works slightly different from the detection of unexpected signifier matches. More precisely, for detecting concurrent signifier changes, we have to consider the matches among model elements from the origin model V_o and model elements from the revised models V_{r1} and V_{r2}. Therefore, we iterate through the match models $M_{V_o,V_{r1}}$ and $M_{V_o,V_{r2}}$ indicating the correspondences obtained from the model matching phase (cf. Section 5.1). For each correspondence, we examine based on the match triggers whether the model element in the revised model has been modified such that its signifier might have changed. If this is the case, we compare the model element from the V_o with the corresponding model element in the respective revised model, V_{r1} or V_{r2}, using the signifier specifications. If this comparison reveals a change of the signifier, we check whether the corresponding model element's signifier in the opposite revised model has also changed; if this is the case, a *concurrent signifier change* warning is raised. In Section 6.3.4, we introduce the applied process in more detail and illustrate how the merge issue in the merge scenario presented in Section 3.1.3 is revealed using the presented approach.

6.3.1 Signifier Specifications

The user-specified *signifier specifications* are realized using the same technology that is also applied for adapting the model matching phase: *match rules* that are expressed using the Ep-

silon Comparison Language [Kol09] (ECL). This technology has been already introduced in Section 3.4.4. In the remainder of this section, we discuss our rationale behind using ECL for specifying signifiers and provide two examples for signifier specifications. In particular, in Listing 6.2, we show an excerpt of the signifier specification for UML class diagrams, and in Listing 6.3, we depict an excerpt of the signifier specification for Ecore models. Please note that these two exemplary signifier specifications will also be used in the subsequent sections for presenting the signifier-related merge issue detection. In this section, we further show how signifier specifications are structured, point out how the match model obtained in the preceding matching phase (cf. Section 5.1) can be accessed from within ECL rules, and how match rules for model matching might differ from ECL rules for specifying the signifier of model elements.

Listing 6.2: Excerpt of the Signifier Specification for UML Class Diagrams

```
1  pre {
2    var matchModel =
3      new Native("org.modelversioning.signifiers.MatchModelECLBridge");
4    matchModel.load();
5  }
6
7  operation Element isMatched(other : Element) : Boolean {
8    return matchModel.isMatched(self, other);
9  }
10
11 rule Class2Class
12   match l : Left!Class
13   with r : Right!Class {
14   compare :
15     l.name = r.name and
16     l.package.isMatched(r.package)
17 }
18
19 rule Generalization2Generalization
20   match l : Left!Generalization
21   with r : Right!Generalization {
22   compare :
23     l.specific.isMatched(r.specific) and
24     l.general.isMatched(r.general)
25 }
```

Requirements for signifier specifications. As already mentioned, with the term *signifier*, we refer to a combination of specific features of a model element type, which convey the superior meaning of a model element. Therefore, a signifier specification for a model element type should provide the information on *which metamodel features* have to be considered for computing a model element's signifier and in which way the values of these features have to be *combined*. Furthermore, this specification should also be *directly applicable* to examine whether two model elements have matching signifiers. It also should be possible to *access the match model* from within a signifier specification for examining whether a model element has been matched during

Listing 6.3: Excerpt of the Signifier Specification for Ecore Models

```
1  pre {
2    var matchModel =
3      new Native("org.modelversioning.signifiers.MatchModelECLBridge");
4    matchModel.load();
5  }
6
7  operation EModelElement isMatched(other : EModelElement) : Boolean {
8    return matchModel.isMatched(self, other);
9  }
10
11 rule EClass2EClass
12   match l : Left!EClass
13   with r : Right!EClass {
14   compare :
15     l.name = r.name and
16     l.ePackage.isMatched(r.ePackage)
17 }
18
19 rule EReference2EReference
20   match l : Left!EReference
21   with r : Right!EReference {
22   compare :
23     l.name = r.name and
24     l.eContainingClass.isMatched(r.eContainingClass) and
25     l.eType.isMatched(r.eType)
26 }
```

the preceding match phase in the merge process. Finally, the signifier specification should be *programmatically analyzable* for collecting all metamodel features that are *visited* in model elements when comparing them. All these requirements are fulfilled with ECL. Thus, ECL match rules are a perfect fit for these requirements.

Structure of signifier specifications. A signifier specification for a specific modeling language is a set of ECL rules, which always compare model elements *of the same type* with each other (e.g., Class2Class in line 11 of Listing 6.2). A rule in a signifier specification typically contains expressions comparing the primary real-world identifiers, such as the name of a model element. For instance, the name of a UML class conveys its real-world identity. Thus, in line 15 of Listing 6.2, the name of classes is checked for equality. In several cases, there might be multiple features that convey the real-world identity *in combination*. Therefore, we may easily combine multiple expressions, each comparing one feature, in terms of conjunctions in an ECL rule. The features to be taken into account for computing the signifier of model elements are usually dictated by the *uniqueness constraints* of the modeling language's metamodel. Such uniqueness constraints indicate that there must not be more than one model element having the same value at the unique features within one model. When features are defined to be unique, it is most likely that these features convey the real-world identity of a model element and, there-

fore, should be considered in the signifier specification. Hence, we may automatically generate an initial signifier specification for a modeling language by creating a rule for each metamodel class, that compares its features that are defined to be unique. To summarize, using ECL, we may easily represent the signifiers of model element types in the form of ECL rules and use these rules to check whether two model elements share the same signifier. If the rule indicates a match, both model elements share a common signifier; if two model elements cannot be matched according to such a rule, the compared model elements have a different signifiers.

Accessing the match model. As can bee seen from the two example signifier specifications, there is a pre block and an operation block at the beginning of both signifier specifications (cf. Listing 6.2 and Listing 6.3 line 1 to 9). These two blocks enable developers of signifier specifications to query the match model for determining whether two model elements have been matched in the preceding match phase. Therefore, we exploit the functionality of ECL to instantiate Java objects using the built-in Native data-type and calling their methods. More precisely, from within a signifier specification, we instantiate a class called MatchModelECLBridge, which is allows accessing a singleton class containing all currently loaded match models in the current merge process. Using the operation isMatched() (cf. Listing 6.2 at line 7), we query the match model for finding out whether two model elements are corresponding to each other according to the match model. Please note that there is a major difference between this custom operation isMatched() and ECL's built-in function matches(). Using the operation isMatched(), we may query the match model computed in the preceding match phase of the merge process; with the function matches(), the comparison is delegated to another match rule for the model element within the same ECL file. For example, `l.package.isMatched(r.package)` in line 16 of Listing 6.2 returns true if the match model contains a match for a classes. On the contrary, the expression `l.package.matches(r.package)` returns true if there is a match rule for packages in the ECL file that evaluates to true for the `l.package` and `r.package`.

In our example signifier specifications, we used the operation isMatched(). The reason for using this operation rather then the function matches() is that we aim at checking whether, for instance, a class has the same name and is contained by the *same package*. Thereby, the signifier of the package is irrelevant. It is only relevant whether both classes are located in the same package according to the match model. In other words, the computation of the signifier for classes is *isolated* from the computation of the signifier for packages. For instance, assume that the signifier of a package is defined to be a package's name. Developer 1 changes the name of the package and developer 2 concurrently changes the name of the class contained in the concurrently renamed package. If we would use the function matches() for comparing the container package of the class, we would call the signifier specification for packages for evaluating the signifier of classes. Consequently, we would obtain a concurrent signifier change because, on the one side, the name of the class has been changed and, on the other side, the name of the package has been changed, which is incorporated in the computation of the contained classes' signifiers. In most of the scenarios, this is not what we aim for.

Match rules versus signifier specifications. As already mentioned, we apply the same technology for specifying signifiers as we apply for matching models in the matching phase of the

merge process. Thus, the question arises whether match rules differ from signifier specifications at all. From a technical point of view, they do not. Both are ECL rules and, also if we did not explicitly mentioned it in Section 5.1, we may also query the match model from within ECL match rules applied in the model matching phase, as the rule-based matching is installed after the UUID-based matching. Thus, we already have a match model to be queried from within the match rules applied in the model matching phase. However, from a practical point of view, there are scenarios, in which users might prefer to apply different ECL rules for model matching as for signifier matching. For example, we might want to use fuzzy matching techniques or use thesauri for establishing correspondences among model elements in the matching phase. Obviously, such techniques are usually improper for specifying a model element's signifier, although it is possible from a technical point. Therefore, we allow users to configure different ECL files for model matching and for the signifier specifications. However, users may keep them synchronized as they wish.

6.3.2 Signifier Specification Preprocessing

As already outlined above, for detecting unexpected signifier matches and concurrent signifier changes, we have to compare the signifiers of model elements in all three considered versions of a model (V_o, V_{r1}, and V_{r2}) with each other. The computation of signifiers, however, might be rather time-consuming. To avoid having to compare *all* model elements with each other, we aim at keeping the amount of signifier comparisons at a minimum. We only have to compare the signifiers of model elements, if they have been added in the course of one revision or if they have been subjected to modification such that their signifiers might potentially have changed. Therefore, we apply a preprocessing step, which analyzes the signifier specifications to reveal and explicitly represent the metamodel features of a type making up the respective signifier. Being equipped with this information, we only have to compare those model elements that have been changed at those features.

For collecting the information on the metamodel features that are visited by the ECL rules, we traverse through the abstract syntax tree (AST) of each rule. If we reach a node in this tree that corresponds to a feature name of the current modeling language's metamodel, it is saved as visited feature for the metamodel type that is compared by the current rule. If we encounter the built-in function called matches(), we additionally have to keep track of the metamodel features that are visited by the rule that is indirectly invoked by this function call. Therefore, we read the argument that is passed to this function and obtain the argument's metamodel type. Then, we search for the corresponding rule that processes the obtained metamodel type and also save the features that are visited by the obtained rule.

The obtained visited features are saved in terms of another model. Therefore, we introduce the *match trigger metamodel* depicted in Figure 6.18. This metamodel contains a root class called MatchTriggerModel, which contains an instances of the class MatchTrigger for each rule in the signifier specification. Match triggers refer to the EClassifier from the current modeling language's metamodel that is processed by the current rule and the features that are visited by the current rule for computing the signifier of the respective EClassifier. As rules may depend on each other in terms of calls to the function matches(), a match trigger may also be triggered by other match triggers. For instance, if in the ECL rule for UML classes the function matches()

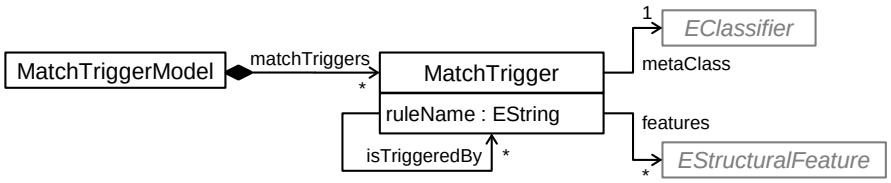


Figure 6.18: Match Trigger Metamodel

is called for the containing package, the match trigger for the UML class refers to the match trigger for UML packages through the reference isTriggeredBy. Thereby, we know that we have to run the signifier comparison for a UML class either if its name or the respective features of the containing package has been changed. Please note that, thereby, we only represent a conservative estimation of the actual impact an applied operation has on a model element's signifier, as we do not consider the full semantics (such as logical conjunctions) of an ECL rule.

The functionality of match triggers is further illustrated in terms of a small example, which is depicted in Figure 6.19. In this figure, we show the match trigger model for the ECL rule called EReference2EReference from the signifier specification in Listing 6.3. This match trigger model contains one instance of MatchTrigger, which refers to the metamodel class EReference and three features, most importantly for this scenario, the feature name. Thus, the corresponding rule compares instances of the class EReference and considers, besides two other features, the feature called name for computing the signifiers of instances of EReference. Next to the match trigger model, Figure 6.19 also shows a diff model and a revised model. The diff model contains one operation: the instance of a FeatureUpdate (cf. Section 5.2). This operation modifies the value of the object in the revised model at the feature name as indicated by the references feature and affectedObject[2]. Based on these three models, we can derive that according to the match trigger, we have to include the object in the revised model in the signifier matching process because a feature that is considered by the signifier specification has been modified; thus, it is most likely that the signifier of the object has changed.

6.3.3 Unexpected Signifier Match Detection

Unexpected signifier matches occur if a model element in one revised model (e.g., V_{r1}) unexpectedly matches with a model element in the opposite revised model (e.g., V_{r2}). By *unexpected*, we mean that the matching model elements *do not correspond to each other* according to the match models between the origin model V_o and one revised V_{r1}, as well as V_o and the opposite revised model V_{r2}; thus, these two model elements have the same signifiers, although they actually *have no common origin model element*. As a result, when naively merging the operations of both revisions, we end up having two model elements sharing the same real-world identity and the merged model seems to have redundant model elements. As an example for unexpected signifier matches, consider the merge scenarios presented in Section 3.1.1 and Section 3.1.2.

[2]Please note that an instance of FeatureUpdate refers as a matter of fact to a model element in the *origin model* via the reference affectedObject. However, for the sake of readability, we abstracted away the navigation to the corresponding model element in the revised model through the match model.

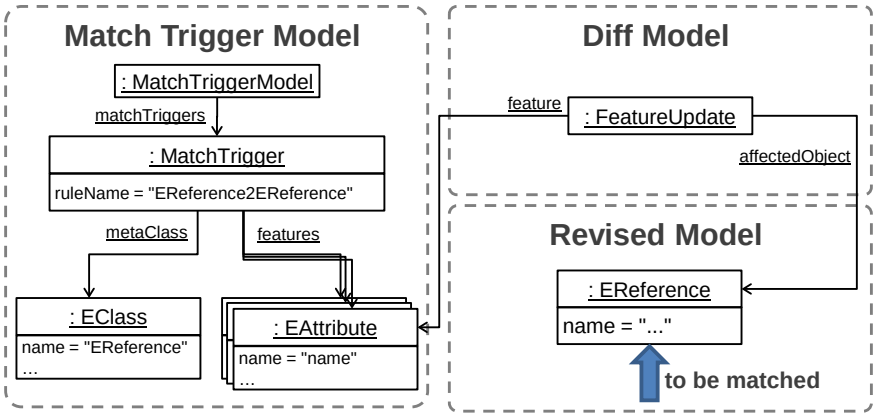


Figure 6.19: Example for Match Trigger

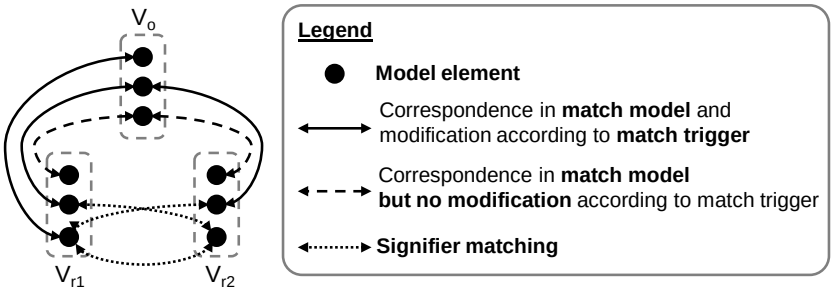


Figure 6.20: Selecting Model Elements for Signifier Matching

Selecting model elements for signifier matching. To avoid such an unfavorable redundancy, we compare the signifier of model elements in the revised model with each other. Thereby, we only have to match the signifier of all *inserted* model elements or model elements that have been *modified* such that their signifier might have changed. Obviously, we only have to match model elements having the same type with each other, whereas we may omit those model elements that correspond to each other according to the match model because they are expected to have the same signifier.

The selection of model elements to be compared for finding unexpected signifier matches is illustrated in Figure 6.20. In this figure, the three model versions V_o, V_{r1}, and V_{r2} are conceptually depicted, whereas the black dots represent model elements in those versions. The solid and dashed connections among model elements between the original model and the revised models indicate that the connected model elements correspond to each other according to the match model. More precisely, model elements that are connected with *solid lines* are not only marked as a *match* in the match model, they have been further *modified* such that the match trigger indicates the potential change of its signifier. Model elements connected with a *dashed line* are only marked as *match* in the match model and have *not been modified* or at least not in a way such that their signifier might have changed.

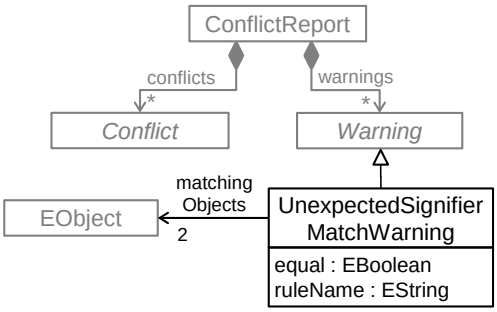


Figure 6.21: Metaclass for Unexpected Signifier Match Warnings

In this illustrative example, we have three model elements in each version. In the model version V_{r1}, all three model elements are marked as a match in the match model and two of these model elements have been further modified such that their signifier might have changed. In the opposite revised model, V_{r2}, only two model elements have corresponding model elements in the origin model. Thus, one model element has been deleted, and one model element has been inserted. One of the two model elements in V_{r2} that have corresponding model elements in the origin model has further been modified such that its signifier might have changed. Consequently, we have to perform three signifier comparisons among four model elements. Both modified model elements in V_{r1} have to be compared to the modified model element in V_{r2} and the inserted model element in V_{r2}. However, we may omit comparing the signifier of the model elements that have the same origin element according to the match model (the black dots in the middle of each version), because these two model elements are expected to have an equal signifier anyway; thus, a matching signifier would constitute no merge issue.

Equal versus similar. If we find an unexpected signifier match, it is helpful for the subsequent steps in the merge process to know whether the matching model elements are entirely equal; that is, all their feature values are the same, or whether they are only similar. By similar, we mean that their signifier matches, but at least one of their feature values is different. This information is important because if they are entirely equal, our versioning system may automatically suggest the developer, who is merging the model versions, to drop one of the two entirely equal and redundant model elements. If these model elements, on the contrary, are not entirely equal but only share a common signifier, our versioning system should indicate the need for a decision on how to resolve this merge issue.

Representing unexpected signifier matches. Unexpected signifier matches are saved in the conflict model in terms of a warning. Therefore, we extend the conflict metamodel by the class UnexpectedSignifierMatchWarning as depicted in Figure 6.21. This new class contains a reference to the unexpectedly matching model elements in the revised model versions and two attributes, equal and ruleName. The Boolean attribute equal indicates whether the matching model elements are entirely equal and the attribute ruleName saves the name of the ECL rule in the signifier specification that indicated the match.

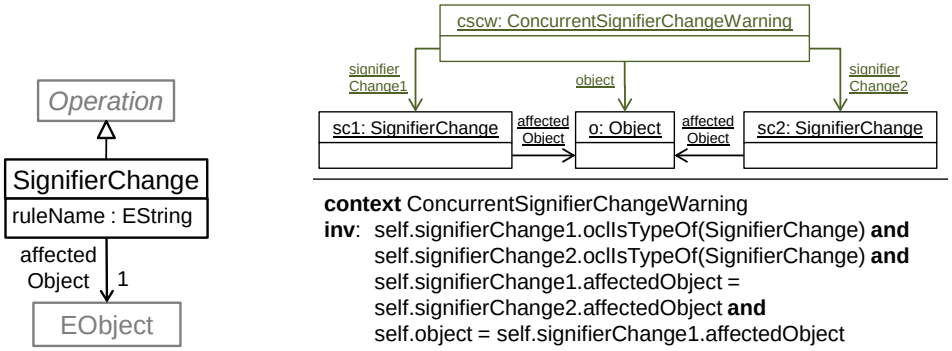
Revisiting the examples. In Section 3.1.1, we introduced a versioning scenario, in which both developers concurrently added a generalization relationship between the same classes. As these added generalization model elements have been newly inserted, they have no common origin model element in V_o and, consequently, no correspondence is noted in the match model. As a result, when naively merging both operations, we end up having two exactly equal generalization relationships. To overcome this unfavorable situation, we use the previously discussed signifier matching approach. As the generalization objects have been inserted, they will be matched using the signifier specification given in Listing 6.2. Fortunately, the rule Generalization2Generalization (cf. line 19) matches for both added generalization objects because both objects refer to the same objects through the reference specific and general. Thus, we proceed with comparing the matching objects in detail and figure out that not only the signifier matches, but the whole objects are entirely equal. Consequently, an *unexpected signifier match warning* is raised for indicating unexpectedly entirely equal model elements.

The versioning scenario in Section 3.1.2 is comparable yet different. Again, both developers concurrently added similar instances of EReference, which, however, are not entirely equal. Nevertheless, both added model elements share the same signifier. Again, both added model elements have no corresponding origin model element in V_o and, thus, when naively merging, the merged model redundantly contains two model elements that have the same real-world identity. This scenario may again be detected using the aforementioned approach. Therefore, the concurrently added references are matched using the signifier specification for Ecore models given in Listing 6.3. In our example, the rule EReference2EReference (cf. line 19) indicates a matching signifier for the inserted references, because both references connect the same classes and share the same name. When proceeding with comparing these matching objects in detail, we encounter a difference concerning the value of the references' lower bound. Therefore, an instance of UnexpectedSignifierMatchWarning having the attribute value equal = false is created for representing the detected merge issue in the conflict report.

6.3.4 Concurrent Signifier Change Warning Detection

Concurrent signifier changes denote scenarios, in which both developers concurrently modified the real-world identity of the same model element. In other words, the signifier of a model element is contradictorily affected in the course of both concurrent revisions. As a result, it is most likely that the model element's meaning is obfuscated when naively merging the concurrent operations to the model element. Therefore, the developers should be notified in order to review the merge issue. An example for such a merge issue is discussed in Section 3.1.3.

To detect such merge issues, we use a comparable approach as for detecting unexpected signifier matches. However, in contrast to comparing the model elements in the revised models *with each other*, we have to compare the model elements of the revised models (V_{r1} and V_{r2}) with the corresponding model elements *in the origin model* V_o and check whether the signifier has been concurrently affected in both revisions.



(a) Signifier Changes (b) Detecting Concurrent Signifier Changes

Figure 6.22: Concurrent Signifier Change Detection

Finding and representing signifier changes. We first iterate through both match models, $M_{V_o,V_{r1}}$ and $M_{V_o,V_{r2}}$, which contain all model element correspondences between the origin model and the respective revised model. For each correspondence, we check whether the corresponding model element in the revised model has been modified according to the match trigger derived from the signifier specification (cf. Section 6.3.2). If the match trigger indicates that the model element's signifier might have changed, we apply the signifier specification to compare the model element in the origin model V_o to its corresponding model element in the respective revised version (either V_{r1} or V_{r2}). If the signifier has changed, an annotation is added to the diff model (either $D_{V_o,V_{r1}}$ or $D_{V_o,V_{r2}}$) to represent the signifier change.

In order to be able to annotate the signifier change in the diff model, we extend the difference metamodel from Figure 5.9 by the class SignifierChange as depicted in Figure 6.22a. Instances of this class refer to the object in the origin model that has been modified in the course of the revision such that its signifier has been affected.

Detecting concurrent signifier changes. Having explicitly represented every signifier change, we proceed with finding *concurrent* signifier changes. This is largely straightforward. We only have to check whether the signifier of same model element in the origin model has been modified in both revisions. Therefore, we use the object pattern depicted in Figure 6.22b. This pattern matches whenever there is an object *o* in the origin model that is affected, as indicated by the reference affectedObject, by two instances of SignifierChange, *sc1* and *sc2*. If we obtain a match for this pattern, a concurrent signifier change warning is added to the conflict model.

Representing concurrent signifier changes. For describing concurrent signifier change warnings, we again extend the conflict metamodel by the new class ConcurrentSignifierChangeWarning (cf. Figure 6.23). As expected, this class refers to the model element that has been concurrently changed through the reference object. Furthermore, the name of the rule that computed the concurrently changed signifier is saved in terms of an attribute and both concurrent signifier changes in the diff model are referenced.

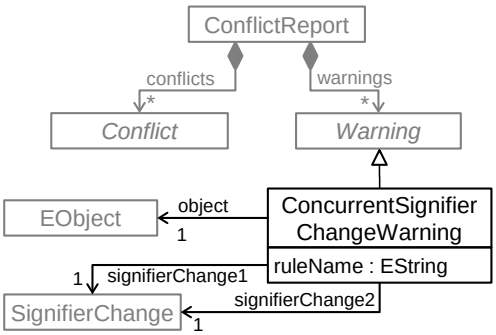


Figure 6.23: Metaclass for Concurrent Signifier Change Warnings

Revisiting the example. An example for a concurrent signifier change is provided in Section 3.1.3. In this scenario, one developer modified the source (i.e., the container) of a reference in an Ecore model, and another developer concurrently changed the target (i.e., the feature eType) of the same reference. To avoid an obfuscated reference, which is obtained when merging naively, we apply the detection approach discussed above. More precisely, for each model element it is checked whether it has been modified such that the signifier changed using the match triggers from the preprocessing step (cf. Section 6.3.2). In particular, we retrieve the reference that has been concurrently modified in the motivating example. Thus, we apply the respective rule from signifier specification (i.e., rule EReference2EReference in line 19 of Listing 6.3) to compare the signifiers of reference in the revised models, V_{r1} and V_{r2}, with the signifier of the corresponding reference in V_o.

In revision V_{r1}, developer 1 changed the containing class of the reference. As a result, the signifier rule EReference2EReference returns that the specified references could not be matched (cf. line 24 of Listing 6.3), because the feature eContainingClass is part of the signifier and is not equal in the compared references. In other words, the signifier of the reference in V_{r1} differs from the signifier of the corresponding reference in the origin model V_o. Hence, this signifier change is accordingly marked in the diff model $D_{V_o,V_{r1}}$.

In the opposite revision V_{r2}, the developer changed the target (i.e., the feature eType) of the same reference. Consequently, the rule EReference2EReference also reports a difference concerning the references' signifier, because the feature eType is incorporated for computing the signifier of a reference. Thus, a corresponding signifier change is added to the diff model $D_{V_o,V_{r2}}$.

By applying the object pattern depicted in Figure 6.22b, we easily find the concurrent signifier change and, consequently, add an instance of ConcurrentSignifierChangeWarning to alert the developers that the concurrently changed reference might be obfuscated, if both revisions are merged.

6.4 Inconsistency Detection

Inconsistency detection is a research domain on its own and a lot of well-elaborated approaches have been recently proposed to address this challenging task (e.g., [BMMM08,Egy11,MVDSD06, OMG10]). Thus, we refrain from reinventing the wheel and rather reuse existing works for detecting language-specific model inconsistencies. Therefore, we integrate an existing approach from the EMF ecosystem, the EMF Validation Framework, into our conflict detection process (cf. Figure 6.1). However, the EMF Validation Framework is only capable of working on states rather than applied operations. Hence, before we may search for inconsistencies caused by the concurrently applied operations, we first have to obtain a merged model, which can be evaluated against user-specified validation rules. Therefore, we installed the step called conflict-tolerant merge before we perform the inconsistency detection.

In the following, we briefly present the design rationale behind conflict-tolerant merging and the merge rules realizing this merge strategy. Subsequently, we elaborate on different inconsistencies and the used approach for detecting them. Finally, to round off the presentation of our merge process, we shortly discuss its final steps; that is, conflict annotation and conflict resolution.

Conflict-tolerant merge. As conflicts might have been detected in previous steps, this merge component has to be capable of producing a merged model irrespectively of any operation-based conflicts. The conflict-tolerant merge, however, is not the focus of this thesis; for more information on this topic we kindly refer to [Wie11]. Thus, we provide only a brief overview on this merge strategy in the following.

From several expert interviews and user studies (cf. [Wie11]), we learned that developers prefer to always have a current, merged model reflecting all opinions and goals of all involved developers. Furthermore, occurring conflicts should not be resolved by one developer alone, who is forced to decide whether to apply either the own or the conflicting operation for creating the merged model. On the contrary, developers prefer to work on a model that builds the *basis for discussing and resolving conflicts*. Conflicts are more than overlapping operations. They often reflect different viewpoints and opportunities to discuss them. Therefore, the overall goal of conflict-tolerant merging is to incorporate as many operations as possible that have been concurrently performed by two modelers into a new version of a model for providing the basis for discussion and conflict resolution [BLS+10b]. Thereby, this merge strategy implements the major principle: neither model elements involved in a conflict nor operations causing the conflict should get lost and, irrespectively of any occurring conflicts, the merge process is never interfered. Thus, developers are never forced to resolve conflicts and inconsistencies immediately.

For realizing this principle, we developed dedicated merge rules. In particular, in case of delete-use and delete-update conflicts, we omit the deletions and only apply the feature updates involved in these conflicts to avoid information loss in the merged model. However, in case of update-update and move-move conflicts, we are not able to apply both conflicting operations because of the restrictions of EMF models: when applying both operations involved in such conflicts, one operation would overwrite the other. For instance, having a single-valued reference, which has been concurrently modified in a contradictory way, we may not express both

operations in one model, because EMF is only capable of persisting one value for single-valued features. The same is true for composite operation conflicts. Applying both operations would potentially compromise one of the operations. However, reflecting all contradictory operations in the merged model is essential for the comprehension of all operations. The developer, who is responsible for resolving all conflicts, has to understand the concurrent evolution of the model in order to be able to construct a consolidated version. Therefore, in case of update-update conflicts, move-move conflicts, as well as composite operation conflicts, we omit both conflicting operations and *annotate* the merged model to support the user in understanding the evolution. We briefly discuss the techniques used for annotating a model at the end of this section.

Before annotating the model, however, we aim at detecting further inconsistencies coming from the validation rules of a modeling language. Therefore, after having created a merged model by the use of the conflict-tolerant merge strategy, we proceed to analyze the model in order to reveal inconsistencies. We may distinguish between two kinds of inconsistencies (also referred to as state-based conflicts) in the realm of EMF-based models.

Well-formedness of EMF models. Every EMF-based model must conform to general well-formedness rules regardless of the respective metamodel. These rules specify that every EMF model must be a spanning containment tree. That is, every model element must be reachable from the root element following a unique path only through containment references. Thus, every model element, except the root element, must have a unique container and no cyclic containment relationships are allowed. Assuming that both modified versions V_{r1} and V_{r2} are well-formed, the merged model obtained by the merge strategy discussed before is also well-formed because otherwise the rules for detecting move-move, delete-move, or delete-update conflicts would have prohibited producing a broken containment tree. Consequently, we do not have to consider containment violations anymore at this point.

Language-specific validation rules. Besides the well-formedness rules of EMF, every model must conform to its metamodel, as well as to additional validation rules, such as OCL constraints. Most of these rules coming solely from the metamodel cannot be violated in the merged model assuming that they have not been violated in each of the two concurrent versions V_{r1} and V_{r2}. Also inserting more than one value to a single-valued feature is avoided by raising an update-update conflict and dangling references are prohibited by delete-use conflicts. However, the merged version might still violate the lower or upper bounds of multi-valued features, uniqueness constraints, and arbitrary additional constraints, such as OCL constraints.

As already mentioned, we use the EMF Validation Framework for detecting inconsistencies, because it is the defacto standard for this task in the EMF ecosystem. Using this framework, each EMF-based model may be validated to detect violations of constraints arising directly from the metamodel and the constraints that have been specified additionally. The EMF Validation Framework supports constraints expressed in OCL or Java. Thereby, language designers may easily develop validation rules to detect inconsistent states as in our motivating example introduced in Section 3.1.6. Whenever a violation is detected, diagnostics are returned from the EMF

Validation Framework, which describe the severity of the constraint violation and provide an error message, as well as the model elements involved in the respective violation. This information on detected inconsistencies is finally saved to the conflict report and handed over to the next step, the conflict annotation.

Conflict annotation. To point the developers to all raised conflicts, inconsistencies, and warnings, we *annotate* conflicts directly in the merged model. Unfortunately, EMF does not provide a common annotation mechanism. Therefore, we ported the lightweight extension mechanism known from UML Profiles [FFVM04] to the realm of EMF models as presented in [LWWC11]. Thereby, every model may be annotated with stereotypes containing tagged values. If for instance, an update-update conflict appeared, a corresponding stereotype is applied to the object that was concurrently modified. This stereotype marks the object that is involved in the conflict and contains information on the contradictory updated values. Stereotype applications may be visualized on top of the abstract or the concrete syntax of a model to support the user in resolving all annotated conflicts directly in the model.

Conflict resolution. After we created a tentatively merged model and annotated it with information on occurred merge issues, it is presented to the user, who may resolve the issues. This can either be done manually or semi-automatically in terms of applying resolution recommendations as presented in [Bro11]. Irrespectively of whether the conflict resolution is performed manually or semi-automatically, developers may resolve merge issues on their own or in a team as presented in [BSW+09]. Anyway, conflict resolution is beyond the focus of this thesis. Thus, for more information on how these two steps are realized in AMOR, we kindly refer to the PhD theses by Brosch [Bro11] and Wieland [Wie11].

6.5 Limitations and Future Work

Although our approach for detecting merge issues among a set of concurrently applied atomic and composite operations is capable of revealing a wide range of conflicts and unfavorable scenarios, which cannot be addressed by currently existing approaches, there are still some open issues posing challenging topics for future research.

Composite operation conflicts. For detecting composite operation conflicts, we currently check whether the composite operations applied in one revision (V_{r1}) are also applicable to the opposite revised model V_{r2} (cf. Section 6.2). This is done by evaluating the validity of the composite operations' preconditions *one-by-one* in V_{r2}; the other atomic and composite operations applied in V_{r1} are not considered. As a result, we are not able to detect a violated precondition, if one application of a composite operation applied in V_{r1} is not be applicable to V_{r2} after other (atomic or composite) operations, originally performed in V_{r1}, have also been applied to V_{r2}. Although this is usually not the case, there is still a chance to encounter such a situation, especially if composite operations contain complex user-defined preconditions that check larger parts of the model. The easiest way of also being able to detect such scenarios would be a try-and-error brute force approach. In other words, we could check the composite operation applications

of V_{r1} *in every possible combination* together with the other applied operations from V_{r1} in the opposite model V_{r2}. This, however, seems to be an unfavorable solution. First ideas to address this challenge involve a detailed analysis and interpretation of the preconditions to reduce the combinations of operations that potentially affect the preconditions. Possibly, we may therefore reuse existing analysis techniques, such as *critical pair analysis* from graph transformation theory [Hec06]. However, this challenge is left for future work.

Deriving the causing opposite operation. Another open challenge is to provide a *complete* solution for deriving the opposite operation that truly causes a composite operation conflict, signifier-related warning, or inconsistency. For composite operation conflicts, we already took first steps towards identifying the causing opposite operation. However, as already discussed in Section 6.2, with our proposed technique, we are capable of detecting the causing operation only for a limited set of condition types. This open issue can be traced back to the challenge of computing the impact an *operation* has on the validity of a condition that restricts the *state* of a model. For model inconsistencies, which are violations of conditions that restrict a model's state, interesting work regarding operations and consistency has been accomplished, for instance, by Blanc et al. [BMMM08], Mens et al. [MVDSD06], and Egyed [Egy11]. However, we left the detection of opposite operations causing signifier-related warnings (cf. Section 6.3) and model inconsistencies (cf. Section 6.4) out of consideration.

Detecting semantic conflicts. Conflicts concerning the semantics of models denote an unexpected behaviour (or interpretation) of a merged model. For detecting such issues, a deep understanding of the semantics of the language is necessary and strongly relies on the runtime semantics of a modeling language [Men02]. At the moment, however, there is "no simple and obvious way to define this complex semantic domain precisely, clearly, and readably" [HR04]. In this thesis, we do not consider semantic conflicts and inconsistencies, but list them here as an interesting direction for future research.

CHAPTER 7

Evaluation

In this chapter, we present the evaluation of the contributions provided in this thesis. First, we presented an *adaptable model versioning framework* (cf. Chapter 3), which provides a merge process that is extensible by user-specified knowledge for improving the overall quality of the merge result. Second, we proposed a novel approach, referred to as *model transformation by demonstration* (cf. Chapter 4), with the goal of easing the burden of developing model transformations manually. Third, we extended existing approaches for state-based model differencing that are only capable of detecting atomic operations so that also applications of *composite operations* can be *detected* a posteriori (cf. Chapter 5). Finally, we contributed mechanism for *detecting conflicts* (besides other merge issues) among a set of concurrently applied operations (cf. Chapter 6). In the following, we outline our applied methods for evaluating each contribution presented in this thesis.

Adaptable Model Versioning Framework. The first contribution, namely the adaptable model versioning framework, is the foundation of the third and fourth contributions and can hardly be evaluated on its own. This contribution allows to *extend* the quality of the operation and conflict detection by plugging in additional knowledge on the processed modeling languages. Thus, the benefit of this contribution is to *enable* the extensibility of the operation and conflict detection. Therefore, we omit to evaluate the adaptable model versioning framework on its own and assess the benefits of this framework indirectly by evaluating the components that are based on this foundation.

Model Transformation By Demonstration. With the second contribution, we aimed at easing the burden of developing model transformations. In order to evaluate the usefulness and the ease of use, there hardly is any more reliable way than conducting a case study with users, who are asked to accomplish the same task using different approaches, and ask them, which approach allowed them to achieve the task more easily. Therefore, we carried out a study with 57 users for assessing the usability of our proposed *model transformation by demonstration* approach. The details on this study are presented in Section 7.1.

Operation Detection. The third contribution extends existing approaches by a novel method for detecting applications of composite operations a posteriori. There are two major attributes for such an approach. First, an approach for detecting applications of composite operations should be as accurate as possible. Therefore, we performed a case study based on real-world models and their evolution and measured how many applications could be correctly detected by our approach and how many applications have been wrongly indicated. This case study is described in Section 7.2.1. Second, the a posteriori detection of composite operation applications is a computation-intense task. Thus, besides the accuracy, also the performance and scalability of such an approach is of major interest. Therefore, we conducted an experiment to assess the required runtime for different scenarios, which is discussed in Section 7.2.2.

Conflict Detection. The fourth contribution is an approach for detecting conflicts. The best way to evaluate approaches for detecting conflicts probably is to assess the accuracy based on real-world models and their evolution. However, the evolution including the information needed for conducting such a study is very hard to obtain. Available model repositories do not provide enough information, because traditional versioning systems, which are currently used to manage the evolution history of models, force users to update their local working copy before they are allowed to commit. If conflicts occur during this update, the user has to resolve all conflicts immediately before the merged version may be checked in again. As a result, in the new checked in version, all conflicts are already resolved and the information on occurred conflicts is lost. Due to the lack of the knowledge on the real-world evolution of models and the lack of existing means for evaluating conflict detection approaches, we developed a novel benchmark. Using this benchmark, we assessed the accuracy of our conflict detection tool and compared the obtained results with the results obtained from applying the same benchmark to two other matured representatives of conflict detection tools in the realm of EMF models. The benchmark and the results gained from this comparison are presented in Section 7.3.

7.1 Model Transformation By Demonstration

The proposed model transformation by demonstration approach (MTBD) introduced in Chapter 4 aims at enabling users, who are not trained in model transformation languages and who are unfamiliar with the modeling language's metamodel, to specify model transformations without posing any restrictions regarding the modeling language and modeling editor. In particular, we presented an approach for specifying *endogenous model transformations* (also referred to as composite operations in this thesis), as well as *exogenous model transformations* [MG06]. However, in the following, we only evaluate our approach for specifying *endogenous model transformations*, as it is the most important basis for the other contributions of this thesis.

Therefore, we conducted a case study for assessing the *usefulness* and the *ease of use* (EOU) [KBK95] of our proposed MTBD tool called EMO. According to Keil et al. [KBK95], the term usefulness refers to the "relative advantage [...] or degree to which the innovation is perceived as better than existing practice". They further state that "usefulness is seen as a function of task/tool fit". In contrast to usefulness, the EOU "is viewed as a task-independent construct reflecting intrinsic properties of the user interface" [KBK95]. Obviously, both dimen-

sions have a strong impact on the adoption of a tool by users. Tools that are very useful in terms of that they are capable of adequately fulfilling the user's indented task may still be very hard to learn and difficult to use. Keil et al. refers to such tools as *power user tools*. If, however, the opposite applies, that is, the tool is very easy to learn and use, but lacks functionality in order to help users to accomplish a certain task, Keil et al. denote such tools as *toys*. In the best case, a tool offers both an easy-to-use interface and a wide range of features. This combination is very hard to achieve, as a growing number of features usually impedes the ease of use. Anyway, as both of these dimensions are of major importance, we aim at evaluating the quality of EMO concerning both the usefulness and the ease of use.

In Section 7.1.1, we first introduce the research questions we aimed to answer with this case study. In Section 7.1.2, we elaborate on the design of this case study. Subsequently, we present the results of this case study in Section 7.1.3 and, finally, discuss these results concerning the previously mentioned research questions in Section 7.1.4.

7.1.1 Research Questions

The goal of this study is to quantitatively evaluate the *usefulness* and the *ease of use* of EMO. We feel that, in comparison to existing approaches for endogenous transformations, such as graph transformation tools [ABJ$^+$10, BEK$^+$06], ATL [JABK08], Epsilon Wizard Language (EVL) [KRP11], or QVT [OMG05a], the benefits of EMO, or MTBD in general, are the following. EMO allows to specify a transformation based on a user-specified *concrete example*, to which the user applies all operations in order to *demonstrate* the transformation. Therefore, users may create and modify these models using the *concrete syntax* and their preferred editor. EMO generates an initial set of conditions from the example models using dedicated heuristics for activating or deactivating certain conditions, which can be fine-tuned by the user in subsequent steps. To help users to cope with more complex transformation scenarios, EMO provides *high-level transformation language constructs*, such as iterations or user inputs.

To evaluate how these features offered by EMO actually improve the usefulness and ease of use for specifying endogenous model transformations, we formulated the following research questions that we aim to answer by this case study.

1. *Concrete example*: Is it easier to specify the input model elements of a transformation by providing a concrete example model instead of generically specifying abstract input patterns? Is a concrete model helpful in general to reason about designing the transformation?

2. *Concrete syntax*: Is it easier to use the concrete (graphical) syntax and the users' preferred editor for creating the example models and demonstrating the transformation instead of working with the abstract syntax (i.e., the metamodel concepts) and a fixed generic transformation specification editor?

3. *By demonstration*: Is it easier to demonstrate the transformation at once based of an example instead of generically specifying the transformation using multiple orchestrated transformation actions or rules?

4. *High-level transformation concepts*: Is it helpful for users to annotate the transformation with high-level transformation concepts, such as iterations and user inputs instead of realizing the same functionality with low-level commands (e.g., using a general purpose programming language)?

5. *Generated conditions*: Is it easier to customize a set of automatically generated conditions instead of creating only the intended ones from scratch?

6. *Required time*: Besides the usefulness of specific features, the usefulness and EOU of a tool is also indicated by the time users required to achieve a certain task. Thus, we aim at answering the question whether users require less time for specifying a model transformation when using EMO in comparison to another tool.

7.1.2 Case Study Design

Reliably assessing the usefulness and ease of use of a tool obviously requires to conduct a study with users, whereby the number of users obviously is the key to an expressive result. According to the definitions of Keil et al., it is very hard to measure the usefulness of a tool in terms of absolute metrics [KBK95].

We conducted an empirical case study with 57 users for evaluating whether EMO advances the ease of use in comparison to an existing model transformation approach and whether it is, at the same time, at least as useful as the other approach. More precisely, we designed two tasks, which had to be achieved by these 57 users. Once they accomplished these tasks using EMO and another comparable approach, they have been asked to fill out a questionnaire, which allows us to compare both approaches quantitatively. In the following, we provide some more details on the particular attributes of this case study and discuss our rationale for certain decisions we made when designing this case study.

Competing approach. One of the first task for designing the case study was to choose the approach(es) to which EMO shall be compared. The most comparable approach is MT-Scribe by Sun et al. [SWG09] as it also implements the idea of MTBD (cf. Section 4.1.7). However, when comparing our tool with MT-Scribe, the comparison would only allow us to evaluate the ease of use and usefulness of the respective *tools* rather than the *underlying approach*. Therefore, we decided to choose a different tool that indeed implements a comparable but not the same paradigm. Probably the most comparable paradigm is *model transformation by example* (MTBE) as discussed in Section 2.3.2. Unfortunately, MTBE approaches mainly focus on specifying exogenous transformations and not on endogenous transformations as EMO does. To prevent an unfair comparison due to the different focus of the compared tools, we did not choose an MTBE approach. Besides other MTBD and MTBE approaches, also the concept of graph transformations is very related to the functionality of EMO. On the one hand, the concept of templates and the process of creating an origin model (i.e., left-hand side in graph transformations) and then a revised model (i.e., the right-hand side) for specifying a transformation is very similar. Thus, we chose to compare EMO to EMF Tiger[1] [BEK+06], which was at the time

[1] `http://user.cs.tu-berlin.de/~emftrans`

when we conducted this case study *the state-of-the-art implementation* of the graph transformation concepts in the realm of EMF. The version of EMF Tiger that has been used for this case study was 1.5.8.

Selected users. Probably one of the most problematic hurdle for conducting a case study is to find a sufficient number of suitable users, who agree to participate. Luckily, we were able to gain 57 participants out of the students that have been attending our lecture on *model engineering*[2] at the Vienna University of Technology. These 57 participants have been in their fourth or fifth academic year; approximately one half of them in the field of *Business Informatics* and the other half in the field of *Software Engineering & Internet Computing*. Consequently, all participants had a strong background on object-oriented programming with Java and object-oriented modeling with UML, as well as little experience with metamodeling. However, none of them had any experience with developing model transformations before. Thus, before assigning the tasks to these 57 participants, we gave them a short demonstration of both tools (EMO and EMF Tiger). Of course, both demonstrations have been of equal depth and we used exactly the same example transformation for demonstrating both approaches.

Tasks to be achieved. As the participants have been experienced in object-oriented modeling and programming, we decided to assign the task of realizing two well-known refactorings of Ecore models using EMO and EMF Tiger.

More precisely, the first refactoring was *Extract Attributes*, which extracts one or more attributes (EAttribute) from two or more classes (EClass) into a new class and adds containment references (EReference) from the original classes to the newly created class. The refactoring implementation had to ensure that there must be at least two classes containing at least one *equal attribute*. Attributes, in this context, are equal if they have the same name the type. Furthermore, the refactoring had to be applicable to two classes up to an arbitrary number of classes, whereas extracting one (equal) attribute in each class up to an arbitrary number of (equal) attributes. The name of the new class and the name of the created reference had to be set according to an input of the user applying the refactoring.

The second refactoring, named *Push Down Operations*, moves one or more operations (EOperation) from a superclass down to two or more of its subclasses. As a result of this refactoring, all subclasses contain a copy of the operations previously contained by the superclass. The realized implementation had to ensure that there must be at least two classes and one common superclass of these classes. Furthermore, the superclass must contain at least one operation. The refactoring had to be applicable to two subclasses up to an arbitrary number of subclasses and it had to be capable of pushing down one operation or an arbitrary number of operations to all subclasses.

Questionnaire. After the participants completed the task, we asked them to fill out an online questionnaire containing several questions on whether they preferred a particular characteristic of one tool or the other. Besides comparative questions, we also asked questions concerning the

[2] `http://tiss.tuwien.ac.at/course/courseDetails.xhtml?courseNr=188395&semester=2010W`

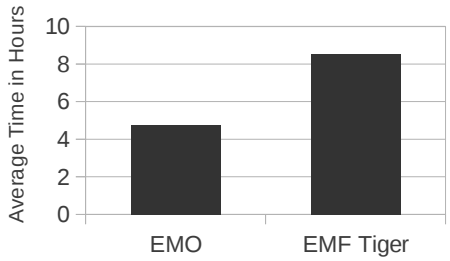


Figure 7.1: Average Time Required to Develop both Refactorings

ease of use of EMO's user interface. Furthermore, we asked how much time the participants spent to realize the implementation of both refactorings with one tool and the other. The precise questions will be presented when discussing the results in Section 7.1.3. Of course, the questionnaire has been carried out anonymously.

7.1.3 Results

The results of the questionnaire are summarized in Table 7.1[3]. The questions in this table are grouped using horizontal lines. The first five groups (questions 1 to 20) correspond to the research questions 1 to 5 (cf. Section 7.1.1). These questions in these groups are designed to be not specifically tied to our MTBD tool EMO; they rather aim at determining the usefulness of the underlying MTBD approach in general. At the end of each of these groups, we also list a control question, which where located out of context in the original questionnaire, to validate the answers of the participants to the other questions. In contrast to the questions of the upper five groups, the sixth group (question 21 to 23) contains questions concerning the usefulness of our tool in particular. In this group, we questioned the usefulness of our tool in general and in comparison to EMF Tiger, as well as in comparison to developing model transformations directly in Java. The last group (question 24 and 25) are geared towards directly evaluating the ease of use of EMO.

In the columns denoted with Answer at the right part of Table 7.1, we provide the fraction of participants in percent indicating that they either strongly agree (+2), agree (+1), disagree (-1), or strongly disagree (+2) with the respective question (or rather statement).

To answer research question 6, we also asked the participants to provide the number of hours they spent in order to realize both tasks using, on the one hand, EMO and, on the other hand, EMF Tiger. The number of hours includes learning how to use the respective tool, as well as testing the developed transformation. The results of this question are illustrated in Figure 7.1. This figure depicts the average time per participant. Thus, the participants required on average approximately 5 hours to learn how to use EMO, as well as to develop and test both transformations with EMO. Using EMF Tiger, the participants spent nearly twice as long.

[3]Please note that we rephrased the questions so that they better fit into the table cells

Question		Answer +2	+1	-1	-2
1.	Working with a concrete example is easier than with an abstract input pattern	37 %	53 %	11 %	0 %
2.	A concrete example helps for designing the transformation	53 %	30 %	14 %	4 %
3.	Finding an appropriate example model is difficult	5 %	30 %	53 %	12 %
4.	Using the concrete syntax is easier to use than the abstract syntax	28 %	54 %	14 %	4 %
5.	Using my preferred modeling editor is easier than using a proprietary transformation editor	37 %	40 %	21 %	2 %
6.	Creating the example models is difficult	4 %	18 %	42 %	37 %
7.	Demonstrating the transformation is easier than specifying a mapped output patterns	42 %	37 %	14 %	7 %
8.	Demonstrating the transformation at once is easier than splitting it into a number of separated rules	39 %	33 %	23 %	5 %
9.	Demonstrating the transformation is difficult	5 %	26 %	51 %	18 %
10.	Annotating iterations is easier than programmatically repeating applying transformation rules	25 %	37 %	25 %	14 %
11.	Understanding how to use iterations is easy	4 %	32 %	44 %	21 %
12.	Annotating user inputs is easier than programmatically querying the user	32 %	27 %	23 %	9 %
13.	Understanding how to use user inputs is easy	39 %	37 %	19 %	5 %
14.	Identifying the right iterations is difficult	14 %	56 %	28 %	2 %
15.	Understanding the meaning of templates is easy	18 %	44 %	32 %	7 %
16.	Customizing initially generated conditions is easier than specifying them from scratch	18 %	42 %	28 %	12 %
17.	Understanding the initially generated conditions takes more time than writing only the required one from scratch	14 %	40 %	30 %	16 %
18.	The initially generated conditions are useful in general	26 %	54 %	16 %	4 %
19.	The default (de-)activation of conditions is adequate	26 %	58 %	12 %	4 %
20.	Configuring the conditions is difficult	11 %	42 %	37 %	11 %
21.	EMO is a useful tool for specifying model transformations	40 %	49 %	9 %	2 %
22.	With EMO, it is easier to specify model transformations than with EMF Tiger	53 %	35 %	11 %	2 %
23.	With EMO, it is easier to specify model transformations than with plain Java	32 %	54 %	12 %	2 %
24.	The user interface for the demonstration is easy to use	21 %	53 %	18 %	9 %
25.	The user interface for editing conditions is easy to use	14 %	30 %	28 %	28 %

Legend: +2. . . Strongly Agree, +1. . . Agree, -1. . . Disgree, -2. . . Strongly Disagree

Table 7.1: Results of the Case Study on Model Transformation By Demonstration

7.1.4 Interpretation of the Results

Having obtained the data presented in the previous section, we proceed with analyzing the results of the questionnaire and draw some conclusions for each research question raised in Section 7.1.1.

Research Question 1. The first research question concerns the benefits of creating a *concrete model* in contrast to an abstract specification of the input pattern of a transformation. Nearly nine out of ten participants stated that it was easier to work with a concrete example rather than an abstract input pattern. Beyond that, more than 80 % indicated that a concrete example helped them to reason about designing the transformation. According to these answers of the participants, we may safely draw the conclusion that working with concrete examples instead of abstract patterns eases the development of transformations. Interestingly enough, the preference for concrete examples is even significant, despite that the participants of the case study are well-trained in abstract thinking as they all have experiences in software development. Anyway, this conclusion does not only support MTBD approaches, the answers favor also MTBE in general. However, 65 % of the participants predicated that finding an appropriate example is difficult. Although this control question contradicts the aforementioned answers, we argue that the answers to the control question are mainly caused by a limitation of EMO as already mentioned in Section 4.3. More precisely, our implementation currently does not allow to merge multiple distinct templates into *one iterative template* after the demonstration has been performed and the templates have been automatically derived. Consequently, the participants had to identify the *minimal model* for a particular transformation in advance, which requires some exercise with and understanding of the concept of iterations in EMO.

Research Question 2. Beyond investigating the preference of users for concrete examples, with research question 2, we examine whether users find it to be easier when working with the *concrete syntax* instead of the abstract syntax, as well as whether they prefer to use their editor of choice rather than the fixed modeling editor from the transformation specification tool. Based on the answers of the participants to question 4 to 6 in Table 7.1, we realize that there is a strong preference for the concrete syntax (82 %) and for using the personally preferred editor (77 %). These answers are also confirmed by the results for the control question in this question group (question 6 in Table 7.1). Again, these results do not only confirm the design principles of MTBD approaches, but also those of MTBE in general. The reason for this strong preference is probably that, using the concrete syntax and the preferred editor, users do not have to get in touch with the modeling language's metamodel for specifying the input and output pattern of the transformation. The results are unambiguous although the participants only had to create transformations of Ecore models, which has a very small metamodel in comparison to, for instance, UML.

Research Question 3. The third research questions aims at assessing the benefits of demonstrating the transformation rather than implementing it using multiple transformation rules (in the case of EMF Tiger) or commands (e.g., when directly using Java to implement the transfor-

mation). Also this research question can be confirmed in favor of MTBD approaches in general. Around 80 % of the participants stated that they prefer to demonstrate the transformation based on their own example and 72 % indicated that splitting the whole transformation into multiple transformation rules is more difficult than a single demonstration. However, we have to be aware of the fact that the transformations that have been developed by the participants are rather small. Thus, the benefits of splitting a transformation into several rules might rise with an increasing overall size of the transformation.

Research Question 4. With this research question, we scrutinize the advantages of high-level transformation concepts, such as iterations and user input annotations. The results for the corresponding questions in the questionnaire (question 10 to 14 in Table 7.1) are somewhat heterogeneous. Whereas 62 % stated that iterations are easier to use than a programmatically orchestrated repetition of transformation rules, nearly the same number of participants also stated that it was difficult to understand how to use iterations correctly. Nevertheless, these results are comprehensible, because iterations increase the efficiency, once one has learned how to use them; understanding the affect of iterations, however, is not easy to understand for untrained users. What we can learn from these results is that high-level concepts for realizing more difficult tasks have to be well-documented in order to increase the efficiency and the acceptance among users. The result for high-level functions that allow users to achieve simpler tasks, such as user input annotations, are rather different. The fraction of participants, who prefer user input annotations over programmatically querying the user is only 59 %, although 76 % stated that it is easy to understand how to configure user input annotations. We believe that the reason for the relatively low number of 59 %, who prefered user input annotations over programmatically querying users, is that the participants have been allowed to query the user only via command line inputs instead of developing a full-fledged graphical user interface. Thus, being experienced in writing Java programs, which is the case for our participants, the realization of user inputs is only a matter of writing a few well-known lines of Java code. Consequently, the relative advantage of user input annotations over programmatically querying the user was quite low for the specific tasks of this case study.

Research Question 5. The fifth research question assesses the usefulness of our approach for generating and configuring pre- and postconditions. This is probably the most difficult task for inexperienced users in the whole transformation specification process in EMO. As a consequence, only approximately six out of ten participants stated that the meaning of templates, as well as the automatically generated conditions are easy to understand and that working with automatically conditions increases the efficiency in comparison to writing only the required ones from scratch (cf. question 15 to 17 and question 20 in Table 7.1). We believe that the reason for these results is that the participants did not have any experience in understanding and writing OCL constraints and that understanding and modifying these constraints certainly requires an understanding of the Ecore metamodel to a certain extent. Nevertheless, once users understood the structure of the conditions, we believe it is more efficient for them to also specify more complicated conditions than using EMF Tiger or plain Java, because more than 80 % of the participants confirmed that the initially generated conditions are useful and that the default

configuration of the generated conditions are adequate. We may learn from these results that reviewing and editing conditions still poses a challenge to inexperienced users and should be addressed in future research work in the domain of MTBD.

Research Question 6. The last research question concerns the time that was required to learn the respective tool and use it to accomplish both tasks. The required time may serve as an overall indicator of the ease of use (and ease of learning) and the usefulness. The results are very clear. On average, the participants spent twice as much time with EMF Tiger as they did with EMO.

Summary. The participants' answers to the more general questions (question 21 to 25 in Table 7.1) largely reflect the overall result of this case study. Nearly nine out of ten participants are of the opinion that EMO is, in general, a useful tool. Even in comparison to EMF Tiger or programming transformations in Java, 88 % and 84 % of the participants prefer to use EMO, respectively. Three out of four participants stated that the user interface of the transformation demonstration process in EMO is easy to use. However, the results for the most difficult task, the fine-tuning of the conditions, indicate that there is a potential for improving the usefulness and the EOU; only 44 % indicated that the user interface for editing conditions is easy to use in general. As already mentioned, the concept behind iterations in EMO are not easily comprehensible; however, once users understand how to use them, they clearly cherish using iterations.

7.2 Composite Operation Detection

The detection of operations that have been applied between two versions of a model without directly tracking the execution of these operations in the editor constitutes a major topic of this thesis. For detecting the applied *atomic operations*, several remarkable approaches have already been proposed (cf. Section 5.2.3 for a survey). In this thesis, however, we introduced a novel approach, which is also capable of detecting applications of *composite operations* a posteriori (cf. Section 5.3). Consequently, we focus on evaluating the usefulness of our approach for detecting the application of composite operations in this section.

Approaches for detecting applied operations are mainly distinguished by two major attributes. First, such approaches should be as accurate as possible. Therefore, in Section 7.2.1, we present a case study based on real-world models and their evolution in order to measure the number of applications that could be correctly detected, as well as the number of applications that are wrongly indicated by our approach. Second, the a posteriori detection of composite operations is a computation-intense task. Thus, besides the accuracy, also the performance and scalability of such an approach is of major interest. Therefore, we conducted an experiment, which is discussed in Section 7.2.2, for assessing the time that is required by our approach for correctly detecting applied composite operations in different detection scenarios.

7.2.1 Case Study

In order to evaluate the accuracy of our approach for detecting applications of composite operations (cf. Section 5.3) in practice, we performed a positivist case study [Lee89] based on

real-world models and their evolution[4]. In particular, following the guidelines for conducting empirical explanatory case studies by Roneson and Hörst [RH09], we applied our approach to detect composite operations that have been performed in the course of the evolution of Ecore metamodels coming from the open source project called *Graphical Modeling Framework*[5].

Research Questions

The study was performed to quantitatively assess the completeness and correctness of our approach when applied to a real-world scenario. More specifically, we aimed at answering the following research questions.

1. *Operation Specifications versus Detection Rules*: Can operation specifications for executing the operation, in general, also be reused for detecting applications of the respective operation or is any information missing for properly detecting them?

2. *Correctness*: Are the detected operation applications correct in the sense that all raised applications have really been applied? If our approach raises incorrect applications of composite operations, what is the reason for these failures?

3. *Completeness*: Are the detected operation applications complete in the sense that all actually applied composite applications are correctly detected; or does our approach miss to detect applications? If the set of detected operations applications is incomplete, what is the reason for missed applications?

Case Study Design

Requirements. As an appropriate input to this case study, we first need EMF-based models that have an extensive evolution history. Furthermore, we do not only need to be equipped with all intermediate versions of these models, we further require the information on the actual composite operations that have been applied in the course of the models' evolution; otherwise, we would not be able to compare the results obtained by our approach with the actual correct set of the composite operation applications. Thereby, in order to accomplish an appropriate coverage of different detection scenarios, the evolution of these models should comprise scenarios having a small set of applied atomic operations through to scenarios having a large number of atomic operations applied at a time. Moreover, there should be scenarios that comprise only a few applications of composite operations, as well as scenarios, in which a higher number composite operations have been performed at once. Finally, the evolution should comprise a large number of different types of composite operations to avoid distorting the results.

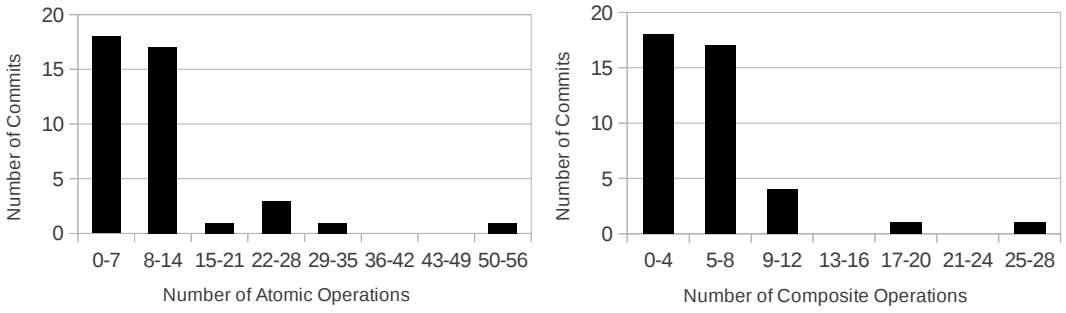
Setup. We chose to analyze the extensive evolution of three Ecore metamodels coming from the Graphical Modeling Framework (GMF), an open source project for generating graphical modeling editors. In our case study, we considered the evolution from GMF's release 1.0 over

[4]We thank Markus Herrmannsdoerfer for providing the tediously gathered data about the GMF evolution and for his great help in conducting this case study.

[5]`http://www.eclipse.org/modeling/gmf`

2.0 to release 2.1 covering a period of two years. For achieving a broader data basis, we analyzed the revisions of *three models*, namely the Graphical Definition Metamodel, the Generator Metamodel, and the Mappings Metamodel. Therefore, the respective metamodel versions had to be extracted from GMF's version control system and, subsequently, manually analyzed to determine the actually applied composite operations between successive metamodel versions. These two steps have already been done in the course of the case study for evaluating COPE [HRW09]. Please note that, as a consequence, the manually determined set of composite operations are unbiased in relation to our case study, because the information on applied operations has not been gathered having the evaluation of our approach in mind. On the contrary, the data has been independently collected. Moreover, metamodel/model co-evolution, which was actually the purpose of gathering the data in the first place for evaluating COPE, is in fact one major field of application of our operation detection approach. Thus, comparing the operation applications obtained by our approach with the gathered data of the original evaluation of COPE perfectly evaluates the usefulness for one of the intended use cases.

Additionally, we had to specify all composite operations manually that have been applied across all metamodel versions using EMO. In total, 32 different types of composite operations have been applied; however, we had to create 48 operation specifications because EMO does not support to specify operations generically towards abstract metaclasses, as they are, however, foreseen in COPE. For instance, the operation *Push Down Feature* has been realized by two distinct operation specifications; one for pushing *attributes* and one for pushing *references*. Having created the operation specifications, we developed a program that automatically performs the operation detection with all revisions of the models and compares the results with the expected results represented in the operation history from the COPE case study.



(a) Atomic Operations (b) Composite Operations

Figure 7.2: Distribution of Operations per Commit

Characteristics of the input data. The evolution of three different models provides a relatively large set of revisions, atomic operation applications, and composite operation applications. In total, the evolution of the considered models comprises 45 revisions that involved at least one composite operation; we omitted revisions, to which only atomic operations have been applied.

Overall, in the course of the models' evolution, 141 composite operations and 342 atomic operations have been applied, whereas one transition between two revisions (called *commit* hereafter) contains on average 5 atomic operations and 2 composite operations. Thereby, we encountered several different types of commits. As depicted in Figure 7.2, most of the commits comprise between 1 and 14 atomic operations or 1 and 8 composite operations. Nevertheless, the evolution also includes also commits having between 15 and 35 atomic operations, as well as one commit, which even poses 52 atomic and 26 composite operations.

Measures. To assess the accuracy of our approach, we compute the measures *precision* and *recall* [OD08] originally stemming from the area of information retrieval for denoting the completeness of pattern recognition algorithms. When applying precision and recall in the context of our study, the precision denotes the fraction of *correctly detected* composite operations among the set of *all detected* operations (i.e., how many detected operations in fact are correct). The recall indicates the fraction of *correctly detected* composite operations among the set of *all actually applied* composite operations (i.e., how many operations have not been missed). These two measures may be thought of as probabilities: the precision denotes the probability that a detected operation is correct and the recall is the probability that an actually applied operation is detected. Thus, both values may range from 0 to 1, whereas a higher value is better than a lower one. The precision and recall may be further combined into the so-called *f-measure* in terms of a harmonic mean ($F = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$).

Results

The results of our case study are depicted in Table 7.2. In particular, in the upper area of this table, we show the results grouped by the three considered models. In the lower part, the results are grouped by type of composite operation. Overall, using our iterative operation detection approach, we were able to detect 99 composite operations correctly among all 141 composite operations (i.e., around 70 %), whereas only two composite operations have been incorrectly detected, which leads to a precision of around 0.98. It is worth noting that the evolution history of these three models is very different. The Graphical Definition Metamodel (GMF Graph for short) was extensively modified within only *one large revision* comprising 52 atomic operations and 26 composite operations, which lead to a quite low recall of 0.5; that is, only thirteen composite operations could be detected. On the contrary, the Generator Metamodel (GMF Gen for short) was subjected to 40 revisions, some comprised a large number of atomic operations and some only a low number. Thus, the evolution of this model is a very representative mixture of different scenarios for the detection of composite operations leading to a precision of 0.98 and a recall of 0.73. The evolution of the third model under consideration, the Mappings Metamodel (GMF Map for short), contained four revisions and in the course of each revision at maximum three composite operations have been applied. Using our approach, we could identify *all* applied composite operations correctly.

When considering the results grouped by the type of composite operation, we can see that the two most occurring operation types, *Rename* and *Delete Feature*, have largely been detected correctly. These composite operation types are, however, comparatively small in terms of comprised atomic operations; thus, the detection is easier. Nevertheless, also for the larger composite

Case Study	# Expected	# Correct	# Wrong	Precision	Recall	F-Measure
GMF Graph	26	13	0	1.00	0.50	0.67
GMF Gen	107	78	2	0.98	0.73	0.84
GMF Mappings	8	8	0	1.00	1.00	1.00
Overall	141	99	2	0.98	0.70	0.82

Composite Operation	# Expected	# Correct	# Wrong	Precision	Recall	F-Measure
Collect Feature	4	0	0	0.00	0.00	0.00
Combine Feature	1	0	0	0.00	0.00	0.00
Delete Feature	18	17	2	0.89	0.94	0.91
Drop Opposite	1	0	0	0.00	0.00	0.00
Extract and Group Attribute	1	0	0	0.00	0.00	0.00
Extract Subclass	1	1	0	1.00	1.00	1.00
Extract Superclass	9	9	0	1.00	1.00	1.00
Flatten Hierarchy	1	0	0	0.00	0.00	0.00
Generalize Attribute	1	0	0	0.00	0.00	0.00
Generalize Reference	6	4	0	1.00	0.67	0.80
Imitate Supertype	1	0	0	0.00	0.00	0.00
Inline Superclass	3	2	0	1.00	0.67	0.80
Make Abstract	1	0	0	0.00	0.00	0.00
Make Containment	1	1	0	1.00	1.00	1.00
Make Feature Volatile	6	6	0	1.00	1.00	1.00
Move Feature	3	0	0	0.00	0.00	0.00
New Opposite Reference	14	9	0	1.00	0.64	0.78
Operation to Volatile	3	2	0	1.00	0.67	0.80
Propagate Feature	1	1	0	1.00	1.00	1.00
Pull Up Feature	3	0	0	0.00	0.00	0.00
Pull Up Operation	3	2	0	1.00	0.67	0.80
Push Down Feature	7	2	0	1.00	0.29	0.45
Push Up Operation	1	0	0	0.00	0.00	0.00
Remove Supertype	1	1	0	1.00	1.00	1.00
Rename	29	27	0	1.00	0.93	0.96
Replace Class	2	2	0	1.00	1.00	1.00
Replace Enum	4	4	0	1.00	1.00	1.00
Replace Inheritance	3	2	0	1.00	0.67	0.80
Replace Literal	1	1	0	1.00	1.00	1.00
Specialize Reference Type	4	0	0	0.00	0.00	0.00
Specialize Supertype	6	5	0	1.00	0.83	0.91
Volatile To Opposite	1	1	0	1.00	1.00	1.00
Overall	141	99	2	0.98	0.70	0.82

Table 7.2: Results of the Case Study on the Operation Detection

operation types, such as *Extract Superclass* and *Specialize Supertype*, we achieved good results. However, there are several composite operation types, whether they are small or large, which our approach could not detect at all (e.g., *Specialize Reference Type* and *Pull Up Feature*).

Interpretation of the Results

Research Question 1. The overall results across all considered models and commits with an f-measure of 0.82 are, in our opinion, very promising. As the operation specifications used in this study have been created using EMO just as we would create them for only *executing* them, we may already answer the first research question and conclude that, in general, it is possible to reuse the same operation specifications also for *detecting* them *a posteriori*.

Research Question 2. Especially the *precision* obtained by our approach is de facto optimal. That means, nearly all detected composite operation applications are correct. The reason for the lost 2 % of the precision in GMF Gen is actually not because the two indicated occurrences of the operation Delete Feature are incorrect. In fact, the reason is that the composite operation Flatten Hierarchy has not been detected and in the course of this operation, two features (containment references) have been deleted. Thus, not detecting the larger composite operation caused the "incorrect" detection of two smaller operations that would have been part of the missed larger composite operation.

Research Question 3. Although the precision is very satisfying, the recall values are rather low for some commits and operation types. For investigating the causes for these low recall values, we analyzed the missed operation applications in more detail. Our first guess is that the low recall values are caused by the complexity and size of the respective composite operation type could not be verified by analyzing the resulting data. Several large composite operations having complex pre- and postconditions could be detected without any issues and the data representing the size of the operation types does not seem to correlate with the recall values of their detection. Admittedly, the sample size is quite low for arguing the statistical independence of these two variables though because there are some types of operations for which we only have one or two expected but no detected applications.

However, by a more detailed analysis of the specific cases, in which expected applications could not be properly detected, we identified the actual cause for the low recall values: *overlapping sequences* of composite operation applications. Some specific cases of overlapping sequences of composite operations can be addressed using our iterative detection approach. This works, however, only if at least the first composite operation can be detected based on its diff pattern; otherwise, the first operation is unknown and, thus, no intermediate state can be computed. Unfortunately, in many scenarios it is not possible to detect the first application, because it is hidden by a subsequent operation. Consider, for instance, an example from the GMF Gen model evolution, a subset of the operations applied between revision 1.229 and 1.230 (cf. Figure 7.3). In the course of this revision, the developer first applied a Pull Up Feature by shifting the attribute requiredPluginIDs from GenExpressionInterpreter to its superclass GenExpressionProviderBase. The resulting intermediate state is shown in V 1.229' in Figure 7.3.

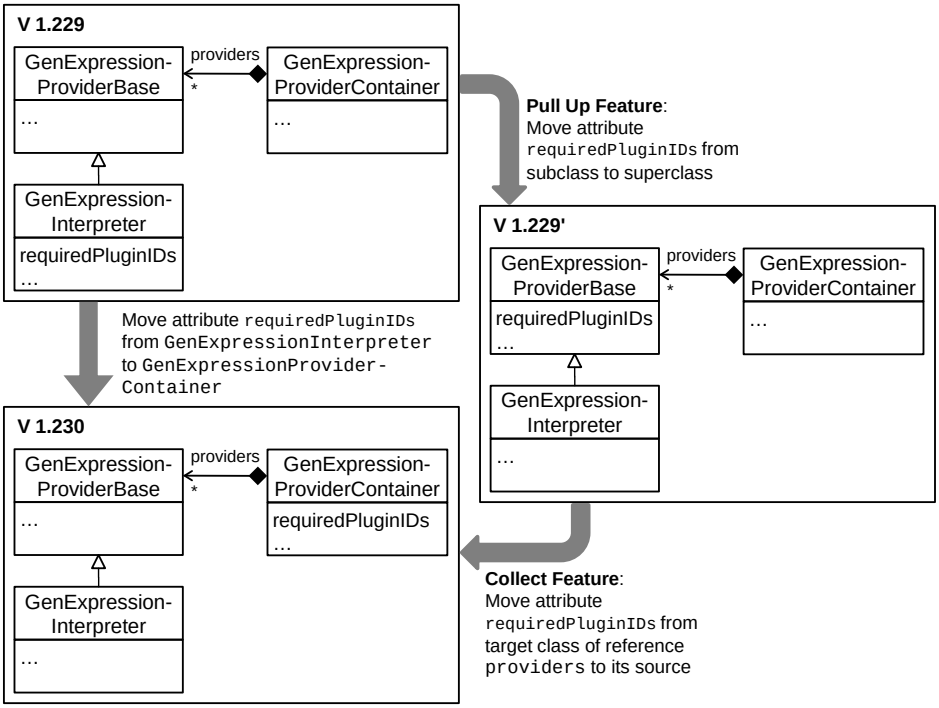


Figure 7.3: Overlapping Sequence of Composite Operations taken from GMF Gen

Subsequently, the developer applied the composite operation Collect Feature by again shifting the same attribute over the reference provides from this reference's target to its source class GenExpressionProviderContainer. However, when only considering the state of the model depicted in V 1.229 and V 1.230, as done by our approach, only one atomic operation can be obtained, which is the move of attribute requiredPluginIDs from GenExpressionInterpreter to GenExpressionProviderContainer; consequently, the postconditions of Pull Up Feature, as well as the preconditions of Collect Feature are violated in the origin state in V 1.229 and the revised state in V 1.230, respectively.

The correlation between the number of applied (i.e., expected) composite operations and the recall value can also be statistically shown based on the data gathered in our study. More precisely, we computed the *relative number of composite operation applications* of each commit; that is, the number of expected composite operations in one commit divided by the number of model elements in the respective model, and compared it to the achieved recall values for the corresponding commit. Although the sample size is relatively small, we obtained a Pearson's correlation [RN88] of around -0.67 between these two variables. Our interpretation of this correlation is as follows: the more composite operations have been applied within one commit, the more likely it is that composite operations are sequentially overlapping with the consequence that the overlapping composite operations cannot be detected. This, as a result, leads to a lower recall value. The relationship between the number of applied composite operations to the recall value is depicted in Figure 7.4. Please note that, for the sake of readability, we grouped equal

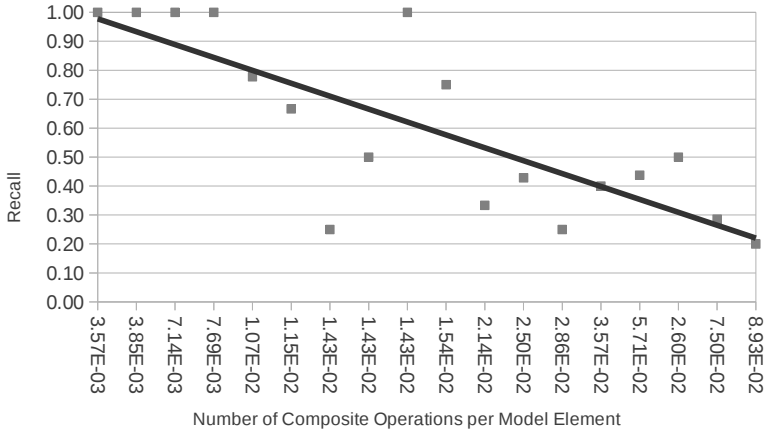


Figure 7.4: Recall versus Number of Applied Composite Operations

numbers of composite operations per model element and averaged the respective recall values within one group in this graph. The black solid line depicts the linear regression (i.e., the estimated trend), whereas the coefficient of determination R^2 is around 0.62. As the goal of this analysis is to convey the basic idea, we omit to compute the statistical significance.

7.2.2 Performance Evaluation

Besides the accuracy of our operation detection approach (cf. Section 5.3), we also aim at exploring its scalability and performance. In particular, we investigated the effects on the runtime with increasing model size and increasing size of applied atomic operations. Therefore, we conducted an experiment based on carefully synthetic examples.

Setup. For assessing the scalability of our approach, we measured the required runtime to detect composite operations in four detection scenarios successfully, whereas *(i)* five, *(ii)* two, *(iii)* one, and *(iv)* zero applied composite operation(s) shall be detected. For these scenarios, we measured the steady state performance[6] of our implementation while, on the one hand, stepwise increasing the size of the evolving models and, on the other hand, stepwise increasing the number of concurrently applied atomic operations. Thereby, we isolate the effects on the runtime when the size of the model increases or the number of concurrently applied atomic operations increases. The experiment was conducted using an Intel® Core™2 Duo with 2.53 GHz running Ubuntu 11.04.

Results: Increasing model size. The results of our experiment for increasing the model size are illustrated in the left graph in Figure 7.5. In this graph, we depicted the overall runtime required for the four detection scenarios, as well as the runtime required for comparing the models (dashed grey line). When stepwise increasing the model size for all four detection scenarios

[6] A program is run repeatedly until the execution time of each run stabilizes.

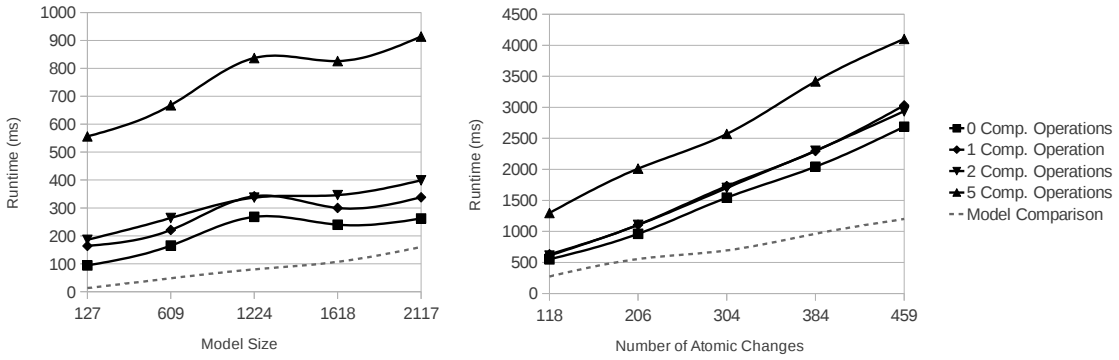


Figure 7.5: Operation Detection Runtime

from 127 to 2117 model elements, whereas keeping the number of concurrently applied atomic operations constantly at around 60, the increase in runtime is largely equal to the increase of the runtime needed for the model comparison. Especially in the scenario, in which *no composite operation* has been applied, the overall runtime for every evaluated model size is constantly around 100 milliseconds (ms) higher than the time required for only obtaining the atomic operations by applying the model comparison. However, for successfully detecting *five applications* of composite operations, the additional runtime of our approach over the time needed for only comparing the models grew from 543 ms for a model containing 127 model elements to 745 ms for a large model having 2117 model elements.

Results: Increasing size of atomic operations. The results of our experiment for increasing the size of concurrently applied atomic operations are depicted in the right graph in Figure 7.5. Again, we also plotted the time required for comparing the models (dashed grey line) as a reference. In this experiment, we used a rather large model consisting of 2117 model elements. The range of concurrently performed atomic operations has been stepwise increased from 118 to 459 applied operations. To correctly determine that *no composite operation* has been applied among 118 atomic operations, the algorithm needed only 550 ms. That are additional 320 ms to the time needed for the only comparing the models. However, when 459 operations have been concurrently applied instead, the runtime was 2688 ms, which already are 1411 ms more than the time required for only comparing the models. A similar increase of runtime was measured for the other scenarios. For instance, the required runtime increased to 1269 ms for finding *five composite operations* among 118 atomic operations and 4101 ms for finding the same composite operations among 459 atomic operations.

We also measured the share of the runtime that each phase accounts for. Thereby, we gained the results that model comparison on average causes 34,54 % of the runtime, whereas the preselection accounts for 41,69 % and the condition evaluation had a share of 23,77 %. Please note, however, that these shares strongly vary due to the different characteristics of the respective detection scenario. For instance, in scenarios having a low number of atomic and composite operation, the share of the preselection time may be much lower.

Interpretation of the results. What we can learn from this experiment is that the runtime of our approach only slightly depends on the model size but, obviously, overproportionally grows with increasing numbers of atomic operations. Of course, the overall runtime depends on the model size to a certain extent. However, this is mainly caused by the time required for comparing the models. The additional time required for detecting composite operations only slightly depends on the size of the model, because the condition evaluation potentially has to consider more model elements. Nevertheless, thanks to the efficient preselection phase, the model elements to be considered are kept at a minimum: only those model elements have to be evaluated that have been affected by atomic operations according to the operation specifications' diff patterns. This phenomenon can be observed in the scenario with 1224 model elements. As we randomly applied atomic operations alongside the composite operations, in this scenario, we "accidentally" applied a diff pattern that is very similar to a composite operation specification. Thus, the preselection phase reported a potential occurrence and, consequently, triggered the evaluation of the preconditions, which is obviously time consuming.

Nevertheless, our approach significantly depends on the number of concurrently applied atomic operations. Because the more operations have been applied between two successive versions of a model, the larger is the search space to be examined for finding diff patterns. Furthermore, having a large number of atomic operations, it is more likely to encounter a matching diff pattern, which forces the algorithm to perform the runtime-expensive evaluation of the pre- and postconditions.

In summary, we feel that the runtime of our approach is satisfying. When considering the potential fields of applications of our approach, which are, among others, mining of model repositories and operation detection for model versioning, we face very different detection scenarios entailing diverging runtimes. Mining of model repositories may pose a very large number of atomic and composite operations potentially causing longer runtimes. For instance, the time required for processing the 45 revisions and 342 atomic changes in the GMF case study (cf. Section 7.2.1) was around two minutes. Luckily, runtime is usually not a crucial criterion in such scenarios. On the contrary, in model versioning, a fast execution time has high priority, as it would cause developers to wait while they check in their models. Fortunately, however, the number of operations applied between two successively modified revisions of one model is rather small: on average, one commit had five atomic and two composite operations in the GMF case study.

7.3 Conflict Detection

Having presented several techniques for detecting different types of merge issues in Chapter 6, we evaluate these techniques in this section. The best way to evaluate approaches for detecting conflicts is to assess its accuracy based on real-world models and their evolution. Although it is possible to obtain an evolution history of models from repositories (i.e., traditional versioning system, such as SVN, Git) of open source projects or from cooperating companies, the *crucial information* for evaluating conflict detection approaches, however, *is missing* in such an evolution history. That is because the evolution history gained from mining repositories lacks, on the one hand, the *conflicting revision* of a model, which a developer intended to check into the repos-

itory, and, on the other hand, it misses the *actually occurred conflicts*. The lack of conflicting versions of a model is caused by the process that is applied in current versioning systems: before a developer may check in her local modified working copy, she has to perform an update, if other revisions have been checked in meanwhile. In case the local working copy contains conflicting changes, the developer has to resolve them before she may check in the merged version. As a result, in this new merged version, all occurred conflicts are already resolved and the information on these conflicts is entirely lost. The only way to overcome this loss, is to use an adapted implementation of a versioning system, which also saves the occurred conflicts. However, even if we develop such an adapted implementation of a versioning system, this system would have to be used by several projects for quite a time before enough meaningful versions and conflicts can be gathered. Moreover, thereby we could only gather the conflicts that are detected by *text-based conflict detection techniques*, the conflict types that are specific to models would have to be manually analysed from each checked in version.

Besides the lack of usable real-world evolution histories of models, also synthetic conflict scenarios in terms of benchmarks are missing for assessing and consistently comparing conflict detection approaches or implementations of them.

Therefore, in this section, we introduce a *novel benchmark* for evaluating conflict detection tools. From our evaluations in the past [ABK+09, BLS+10a], which we performed manually by reconstructing a catalogue of versioning scenarios in each tool and logging the results, we learned that comparing different conflict detection tools is a very difficult and tedious task for several reasons. First of all, different tools require different information; for instance, operation-based conflict detection tools require the information on applied operations in a format they can process and state-based conflict detection tools need the states of a model. Second, the outcome of the tools is heterogeneous and, therefore, difficult to interpret, because each tool reports the conflicts in different ways using different terms and user interfaces. Finally, manually performing such an evaluation requires a great deal of time, is error-prone, and hardly reproducible. Moreover, whenever a new release of a conflict detection tool emerges, the manual evaluation would have to be repeated.

To avoid the tedious manual evaluation, we developed a *generic application interface* for conflict detection tools and designed an *automatic* benchmark that interacts with this generic interface. In order to evaluate a specific tool, all one has to do is to implement the generic interface for the specific tool to be evaluated and start the benchmark.

For evaluating our approach, we implemented the aforementioned interface of the benchmark for our conflict detection tool, as well as for the two most adopted tools in the EMF ecosystem. Having created these interface implementations, we applied the benchmark and performed a detailed comparison of the conflict detection accuracy of all three considered tools. The procedure for this evaluation and the results are presented in this section.

More precisely, we first reveal the goals of this study in Section 7.3.1. Subsequently, we introduce the automatic benchmark in Section 7.3.2 and present the results of applying this benchmark to our conflict detection tool and two other comparable tools in Section 7.3.3. Finally, in Section 7.3.4, we draw some conclusions from the obtained results.

7.3.1 Research Questions

The goal of this study is to evaluate the completeness and correctness of our conflict detection tool in general and in comparison to state-of-the-art tools in the EMF ecosystem. As the approach proposed in this thesis is currently the only available tool that truly supports detecting conflicts caused by violations of the preconditions and postconditions of composite operations (cf. Section 2.1.2), this comparative evaluation only considers generic *atomic operation conflicts* (cf. Section 6.1). Therefore, we developed and conducted a generic and automatically executable benchmark in order to investigate three research questions, which are listed below.

1. *Correctness*: Are the detected conflicts always correct, or are there scenarios, in which the conflict detection tool raises incorrect conflicts?

2. *Completeness*: Are the detected conflicts complete or does the conflict detection tool miss to detect any conflicts? If conflicts are missed, is there a specific type of conflict that is disregarded in general or is the conflict undetected because of an accidental constellation (i.e., it is "only a bug")?

3. *State-based versus operation-based conflict detection*: Is the obtained accuracy of the conflict detection tool proposed in this thesis as accurate as an approach based on operation recording? In particular, with this research question, we aim at investigating whether our conflict detection approaches, which is built upon state-based *model differencing*, achieves the same accuracy as a state-of-the-art approach that is based on *operation recording* (cf. Section 2.1.2).

7.3.2 Benchmark Design

The goal of our benchmark is to provide means for evaluating any conflict detection tool that is capable of processing *EMF-based models*. The benchmark supports conflict detection tools that build upon state-based model differencing, as well as operation recording and covers a wide range of different atomic operation conflict types.

In the following, we provide details on the the generic interface for conflict detection tools, the versioning scenarios that are used to evaluate conflict detection tools, as well as the measures computed from the obtained results. Furthermore, we discuss the tools that have been considered in this evaluation. Please note that the benchmark is available as open source project[7].

Generic conflict detection API. To allow our benchmark to uniformly work with all conflict detection tools to be evaluated, we developed a generic conflict detection API, which is depicted in Figure 7.6. This tool-independent programming interface incorporates the types and methods required to initiate the detection of conflicts for a specific versioning scenario irrespectively of whether the evaluated conflict detection tool builds upon state-based model differencing or operation recording in the editor. More specifically, the API contains the interface IBenchmarkableFacade, which represents the exterior simplified interface to the conflict detection

[7] `http://code.google.com/a/eclipselabs.org/p/model-versioning-benchmarks`

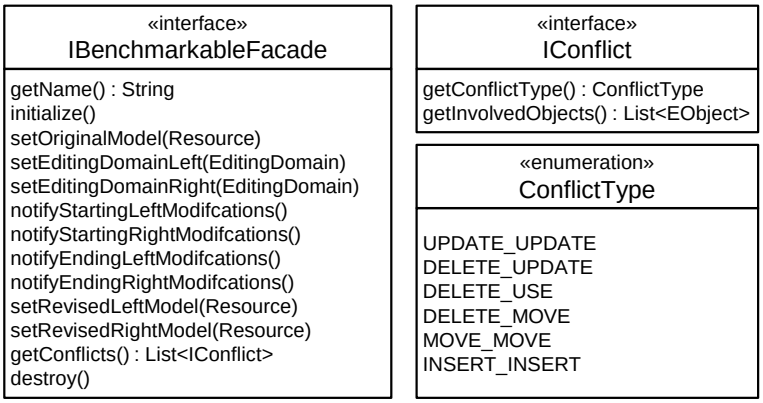


Figure 7.6: Generic Conflict Detection API

tool. This facade dictates several methods to be implemented. These methods are called by the benchmark to provide the conflict detection tool with all the required information for detecting conflicts for a specific model versioning scenario. This information is, on the one hand, the original model resource, as well as the left and right revised model resource. To also allow conflict detection tools that are based on operation recording to collect the applied operations on each side, the benchmark further sets the left and right editing domain within which the operations of the respective model versioning scenario are automatically applied by the benchmark. Thereby, operation tracking approaches may exploit the command API of EMF to track every applied operation as if a user would perform the respective operations. In case, operation-based approaches rather use the notification framework for tracking operations, they may simply register themselves as observer to the specified origin model. Once, all operations of the left side and right side have been automatically applied, the benchmark provides the revised left and right model to the conflict detection facade and calls the method getConflicts(). This method should return a list of all detected conflicts for the particular model versioning scenario in the form of instances of the IConflict interface. Such an instance of this interface describes the involved model elements, as well as the respective conflict type. Having the information on the detected conflicts, the benchmark compares the detected conflicts provided by the evaluated tool with the expected set of conflicts.

Versioning scenarios. The benchmark consists of 23 purposefully selected versioning scenarios, which are partly taken from the collaborative conflict lexicon [BLS+10a], we developed to induce a broad collection of challenging conflict scenarios, and partly from our experiences gained from practice. These scenarios cover all types of atomic conflicts presented in Section 6.1. Furthermore, these 23 versioning scenarios also include some cases in which no conflict is expected at all.

The benchmark is realized as a unit test suite, whereas each test case evaluates a single versioning scenario. A versioning scenario is specified in terms of an origin model, which is annotated with unique IDs (XMI IDs), two operation sequences, and a specification of the ex-

pected conflicts. We decided to attach unique IDs, because, with the help of this benchmark, we aim at evaluating the accuracy of the *conflict detection* and not of the heuristic model matching capabilities.

For evaluating conflict detection tools, the benchmark executes each scenario by first initializing the conflict detection tools, setting the origin model as well as the editing domains and then replaying the operation sequences to the origin model within the editing domains. Finally, it provides the left and right revised models, obtains the detected conflicts of each tool, and computes the evaluation results in terms of specific measures for each evaluated tool.

Measures. To assess the accuracy of the considered conflict detection tools, we compute the measures *precision* and *recall* [OD08], as we already did for assessing the accuracy of our composite operation detection approach (cf. Section 7.2). In the context of conflict detection, we define the precision as the fraction of *correctly detected* conflicts among the set of *all detected* conflicts (i.e., how many detected conflicts in fact are correct). The recall indicates the fraction of *correctly detected* conflicts among the set of *all actually occurred* conflicts (i.e., how many conflicts have not been missed). As these two measures may be thought of as probabilities, their values may range from 0 to 1, whereas a higher value is better than a lower one. In this study, we further compute the *accuracy* as defined by Olsen and Delen [OD08], which is computed from the number of correctly raised conflicts (*tp*), number of correctly handled conflict-free scenarios (*tn*), number of incorrectly raised conflicts (*fp*), and the number of missed conflicts (*fn*) as follows: $\text{accuracy} = \frac{\text{tp}+\text{tn}}{\text{tp}+\text{tn}+\text{fp}+\text{fn}}$

Evaluated conflict detection tools. Using the proposed benchmark, all conflict detection tools can be assessed that are capable of generically handling EMF-based models. Currently, the most prominent state-of-the-art tools that are available for detecting conflicts among concurrent revisions of EMF-based models are EMF Compare [BP08] and EMFStore [KHWH10]. EMF Compare is the defacto standard in the realm of EMF for model differencing and conflict detection. As EMF Compare applies *state-based* model differencing, it follows a very similar approach as the conflict detection technique proposed in this thesis. EMFStore, on the contrary, relies on operation recording and, recently, has gained much attention in the EMF community due to its more precise conflict detection capabilities that are achieved by treating operations instead of model states as first-class artifacts in the conflict detection process. Both EMF Compare and EMFStore work with EMF models in a generic way and do not support to be easily extended by users for detecting conflicts involving specifics of a modeling language or the preconditions of custom composite operations; such mechanisms would have to be programmatically extended by developers. Please note that these tools have also been discussed in Section 2.1.2.

For evaluating the accuracy of the conflict detection technique for atomic operation conflicts proposed in this thesis (cf. Section 6.1), we compare the precision and recall obtained by our tool with the precision and recall achieved by the most current version of EMF Compare (v1.1.2), as a representative for state-based conflict detection tools, and EMFStore (v1.0pre), which is the current state of the art for operation-based conflict detection. Therefore, we implemented[8]

[8]We thank Markus Herrmannsdoerfer, Maximilian Koegel, and Edgar Mueller for their great help in implementing the generic conflict detection API for EMF Compare and EMFStore.

the generic conflict detection API for these approaches and assessed the accuracy of these three tools using the proposed benchmark introduced above.

7.3.3 Results

The results obtained from applying the benchmark to EMF Compare, EMFStore, and our conflict detection tool called AMOR are summarized in Table 7.3.

Conflict Detection Tool	# Correct	# Wrong	Precision	Recall	Accuracy
EMF Compare	2	1	0.67	0.17	0.54
EMFStore	12	0	1.00	1.00	1.00
AMOR	12	0	1.00	1.00	1.00

Table 7.3: Results of the Conflict Detection Benchmark

Using EMF Compare, only 2 out of 12 expected conflicts in the 23 versioning scenarios of the benchmark have been correctly detected and for one versioning scenario, in which no conflict was expected, a conflict has been raised. This leads to a precision of 0.67 and a recall of 0.17. Consequently, EMF Compare accomplished only an accuracy of approximately 54 %.

In contrast, for EMFStore and AMOR, we obtained optimal results. All expected conflicts have been detected by these tools and no unexpected conflict has been raised by both tools. Thus, these tools accomplished an accuracy of 100 %.

7.3.4 Interpretation of the Results

In the following, we discuss the research questions raised in Section 7.3.1 based on the benchmark results that have been presented in the previous section.

Research Question 1. The first question to be investigated by this evaluation concerns the correctness of the conflicts raised by the considered tools. As depicted in Table 7.3, EMFStore and AMOR achieved a recall of 100 % for the evaluated versioning scenarios. That is, every conflict that has been raised by these two tools are actually correct conflicts. In the case of EMF Compare, the results are not as optimal, but still satisfactory. Only one conflict has been raised for a scenario, in which we did not expect a conflict. More precisely, EMF Compare reported a conflict for two concurrent updates of the same model element at the same feature, however, to the *same value*. Therefrom, we may conclude that EMF Compare does not consider whether the revised models are actually the same, in case the origin model in fact differs from *both* revised models. This is indeed surprising. State-based model differencing tools (to which EMF Compare belongs) usually do not focus on applied operations for detecting conflicts. In contrast, they rather work on model states solely. Keeping this fact in mind, the result is unexpected, as the final state in both revised models is actually *equal* and only the applied operations *suggest* that there might be a conflict. The reason for the wrong indication of this conflict is probably due to the internal implementation of the three way comparison and the conflict detection. EMF Compare does not match the revised models with each other; it only compares the revised

models with the common origin version. Subsequently, it seems to detect conflicts solely on the base of the derived differences. As two differences are detected in this scenario that concern the same model element at the same feature and as the revised models are never compared to each other, a conflict is raised in consequence. EMFStore, being a solely operation-based conflict detection tool, does also *not compare* the revised models to each other; however, it checks whether the detected operations update a feature to an equal value. The same is true for AMOR; it also investigates whether the detected overlapping operations indeed have the same affect (cf. conflict patterns in Section 6.1).

Research Question 2. With the second research question, we aim at investigating the completeness of the considered conflict detection tools. EMF Compare did not perform too well concerning this measure. Only two out of twelve conflicts have been correctly detected leading to the unsatisfactory recall of 17 % and the accuracy of only 54 %. The reason for these low values are not bugs in the implementation of EMF Compare; it rather entirely disregards detecting several conflict types. More precisely, EMF Compare does not support to detect *upup* conflicts caused by ordered features, as well as delete-update conflicts and delete-use conflicts in general.

There is not much left to say about EMFStore and AMOR, as these two tools reached an accuracy of 100 %. Consequently, all investigated conflict types are supported and no specific scenario, in which a conflict occurs, has been missed.

Research Question 3. As already mentioned, EMFStore is an entirely operation-based conflict detection tool, which relies on the operation tracking capabilities of a tool called Operation Recorder [HK10] for directly recording all operations that have been applied by developers. Of course, only those operations can be recorded that have been performed using the modeling editors that are supported to be observed by the Operation Recorder; this is true for editors that implement EMF's EditingDomainProvider interface and accordingly use EMF's command API. If such an editor is used, all operations may, thereby, be precisely tracked and provided to EMFStore for detecting conflicts. Due to having this precise description of the applied operation sequences, EMFStore claims to provide an accurate conflict detection, which could be confirmed indeed by this benchmark.

The question arises whether conflict detection tools that refrain from relying on editor-specific operation tracking may provide a comparable accuracy while not posing the severe restriction regarding the editors. As it can be seen from the results for EMF Compare, which is an editor-independent state-based differencing approach, the accuracy of at least this editor-independent tool is far below the accuracy of the state-of-the-art competitor EMFStore that uses operation tracking.

However, with our conflict detection tool AMOR, we showed that it *is possible* to accomplish the same accuracy while not relying on operation recording. Based on this benchmark, we obtained the same optimal results for AMOR as we did for EMFStore. Nevertheless, we have to be aware of the fact that, although the accuracy is the same, AMOR's required computation time may be much higher in comparison to EMFStore. This is because model differencing is a time-consuming task, which is not necessary when directly tracking the operations in the modeling

editor. However, a detailed comparison of the computation time required by existing conflict detection tools is left to future work.

In summary, the question of whether to use operation tracking approaches or state-based model differencing approaches for conflict detection purposes is not a question of *accuracy*; however, it is a trade-off between *editor dependency* and required *computation time*.

CHAPTER 8

Conclusion

In this thesis, we presented several techniques to overcome currently existing deficiencies of state-of-the-art approaches in the realm of model versioning and related research domains. In Section 8.1, we briefly revisit the contributions of this thesis and elaborate on how they tackle heretofore open challenges. Subsequently, we point out current limitations and refer to interesting directions for future work in Section 8.2. Finally, in Section 8.3, we conclude with discussing some lessons learned from conducting the research presented in this thesis.

8.1 Contributions of This Thesis

Adaptable Model Versioning Framework. Recent model versioning systems either realize *generic* model versioning or *language-specific* model versioning. With the adaptable model versioning framework proposed in this thesis, we established the basis for combining the best of both worlds. Whereas the one allows for more flexibility concerning the supported modeling languages, the other one enables a superior quality with respect to the detection of conflicts. To this end, we first realized a generic model versioning framework, which offers out-of-the-box support for all modeling languages conforming to Ecore. Based on this generic framework, we identified and designed necessary adaptation points for plugging in additional knowledge on the modeling language and extended the generic merge process with *adaptable* components, which make use of this additional knowledge for enhancing the quality of the respective step in the merge process. Thereby, developers are empowered to flexibly balance between reasonable adaptation efforts and the required level for versioning support.

Model Transformation By Demonstration. One adaptation point of the proposed system concerns *language-specific composite operations* or, in more general terms, endogenous model transformations. By using this adaptation point, the model versioning system is extensible with composite operation specifications to allow for the specifics of these composite operations in the conflict detection and model merging. Following our main design principle that users themselves should be empowered to adapt the proposed model versioning system, we presented a

novel method for easing the manual specification of model transformations. The ease of creation is achieved by introducing *model transformation by demonstration* (MTBD). More precisely, developers apply or "demonstrate" the transformation to an example model once and, from this demonstration as well as from the provided example model, the generic transformation is semi-automatically derived including its explicit preconditions, operations to be applied, and postconditions. For model versioning purposes, mainly endogenous transformations are of major importance. However, we also showed how this approach for endogenous transformation can be extended to also enable the specification of exogenous transformations

Operation Detection. To avoid any dependencies concerning the employed modeling editors (cf. contribution 1), we use state-based model differencing instead of directly tracking all applied operations. Thus, the third contribution aims at enhancing the precision and completeness of existing model differencing approaches. More specifically, we presented a combination of generic ID-based and *language-specific rule-based model matching*. Furthermore, we contributed a novel technique for detecting atomic operations as well as *applications of composite operations* a posteriori. Both the presented model matching approach as well as the composite operation detection approach may easily be adapted by users for new modeling languages.

Conflict Detection. Finally, we advanced the state of the art for detecting conflicts among operations. More specifically, we contributed a clear specification of *atomic operation conflicts* for EMF models in terms of dedicated conflict patterns. Furthermore, we introduced a novel approach for detecting additional types of conflicts caused by *concurrent composite operations*. Again, the set of considered composite operations for detecting such conflicts may easily be extended by using the proposed approach for specifying composite operations by demonstration. Lastly, we presented techniques for revealing a potentially obfuscated or unfavorable merge result. To this end, we introduced the notion of *signifiers*, which, besides being adaptable by users, build the basis for detecting such scenarios.

8.2 Limitations and Future Work

Science is not about selling a product, it is about *exploring and telling the truth*. To this end, we listed the limitations of each proposed solution in the respective chapter introducing the solution. More specifically, for a detailed discussion of the limitations and interesting directions for future work of the particular presented techniques, we kindly refer to Section 4.3, Section 5.3.6, and Section 6.5. Nevertheless, in this section, we discuss additional general research work that partly has been left to future work.

Usefulness and ease of use of creating adaptation artifacts. One design principle for the proposed model versioning system is to accomplish a highest possible usefulness and ease of use for specifying adaptation artifacts. Therefore, we introduced an approach for developing model transformation by demonstration. However, we did not elaborate on easing the creation of match rules, signifier specifications, and validation rules. As these adaptation artifacts are basically "some OCL-like conditions", this does not seem to be a difficult task. However, the results of

the case study on the usefulness and ease of use of our MTBD approach (cf. Section 7.1) showed that developing conditions is indeed very challenging for untrained users.

Therefore, an interesting topic for future research is to develop novel techniques for specifying conditions for models in general. Easing the specification of conditions would benefit several applications, from match rules, signifier specifications, through to validation rules and many more. We believe there is much potential left for using a combination of MTBD approaches [BLS+09, SWG09], programming by example/demonstration approaches [Lie01], and model annotation (e.g., EMF Profiles [LWWC11]) for developing model conditions on top of the models' concrete syntax.

Further real-world case studies. In Chapter 7, we empirically evaluated the usefulness of our model transformation by demonstration approach as well as the accuracy of the presented technique for detecting composite operations a posteriori. Furthermore, we compared the accuracy of our atomic conflict detection approach to current state of the art tools. However, we left out an evaluation of the composite operation conflict detection and signifier-related merge issue detection. As argued in Section 7.3, real-world data for conducting such an evaluation is very hard to obtain. Nevertheless, we intend to perform an empirical case study for assessing the practical benefits of our approach for detecting composite operation conflicts and signifier-related merge issues in future work.

8.3 Lessons Learned

Having discussed the limitations and future research challenges, we finally conclude this thesis with some more general lessons that we have learned from conducting the presented research work.

Automation versus usefulness versus ease of use. This thesis deals with the semi-automatic specification of composite operations. From these specifications, we automatically derive detection rules for revealing their applications and conflicts caused by those applications. In other words, the user semi-automatically specifies a composite operation, plugs it into the system, and the system automatically adapts its behaviour. One aspect that we have learned from this work concerns the trade-off that has to be made when choosing among the degree of automation, the usefulness, and the ease of use. The trade-off between usefulness and ease of use is largely obvious. The more features are offered by a tool (i.e., the more useful it is), the more complicated it usually gets to use this tool. However, what we have learned besides this trade-off is the impact of the offered automation on the ease of use. Automating several steps by deriving some kind of artifact or providing high-level functions, helps to increase the users' efficiency. However, one has to keep in mind that the user might not initially understand the employed automation mechanisms and, consequently, easily get confused; this, in further consequence, potentially mitigates the system's ease of use. For instance, the pre- and postconditions of composite operations are automatically derived from the user-specified model examples. If the user has to fine-tune these conditions, she first has to understand what and why something has been generated. If, more-

over, the model versioning system automatically uses the generated conditions to detect new types of conflicts or to improve the merge result, the user might be even more confused.

Of course, this is also a matter of how the automation is reflected in the user interface. Thus, developers should be aware of these implications when designing the user interface and teaching the software. Finding the optimal balance among automation, usefulness, and ease of use is very hard. The only way to approach this optimal balance is to focus on a target group of users, who should be tightly involved in the software design.

Predictability is crucial for versioning systems. Related to the matter of automation versus ease of use is also the issue of the versioning system's predictability. The history of versioning systems shows that the most sophisticated and full-fledged solutions to the challenge of versioning often failed to find broad adoption in practice. The software specification, which is managed by the versioning system, should obviously be one of the most valuable assets of a software project. As a consequence, practitioners are not willing to take any risk of letting the versioning system automatically make potentially wrong decisions concerning their software specification; even if the probability of such wrong decisions is quite low. The behavior of the versioning system must be entirely predictable and, if the versioning system automatically makes decisions at all, they must be clearly documented in the history of the managed software specification and easily revertible. To summarize, the realm of software versioning is not considered in practice to be the right place for experimental features.

Versioning and people. An aspect of versioning that is often neglected by researchers is that versioning is not only a technical problem. Versioning is also about people, their cultures, objectives, and opinions. A major facet of software development in a team is the process of communicating people's objectives, working out misunderstandings, and finding the right compromise in order to obtain a consolidated view on the software being developed. Versioning systems only provide one part of the infrastructure for supporting this process. In this sense, a conflict among concurrent operations might be an indicator for different objectives, diverging opinions, or underspecified requirements, but also for an unclear assignment of software development tasks. Thus, the root cause of a conflict might stem from social, technical, through to organizational issues. That should always be kept in mind when working on solutions in the realm of software versioning.

APPENDIX A

Open Source Implementation

The approaches introduced in this thesis have been prototypically realized using the Eclipse Modeling Framework[1] [SBPM08]. All of these prototypes have been published under the terms of the Eclipse Public License[2] (EPL 1.0), a business-friendly free software license. Thus, we kindly invite everyone who is interested in the prototypes to test, use, and evaluate them, or even consider to contribute to the implementation. In the following, we provide some additonal information on these prototypes and refer to the software repositories from where the sources can be obtained.

EMF Modeling Operations (EMO)

The implementation of our MTBD approach for endogeneous model transformations, called EMO (cf. Section 4.1), is the largest open source project that has been developed in the course of this thesis. The source code, comprising more than 52.000 lines of code (including comments) in more than 260 Java files and is hosted at our sourceforge project[3]. Besides the realization of our MTBD approach, this project also comprises the implementation of the model matching and model diffing functionality for atomic operations, as well as for composite operations (cf. Section 5.2 and Section 5.3). EMO has been developed for Eclipse Helios (version 3.6) and relies on EMF Compare 1.0.0. Unfortunately, it has not been upgraded yet to also work with Indigo (version 3.7). Thus, to use EMO, download Eclipse Modeling Tools Helios (version 3.6) from the Eclipse web site[4] and install EMO from our update site[5]. For screencasts, documentation, and more information on contributing to EMO, we kindly refer to EMO's project web page[6].

[1] http://www.eclipse.org/modeling/emf
[2] http://www.eclipse.org/legal/epl-v10.html
[3] https://sourceforge.net/projects/emfmo
[4] http://www.eclipse.org/downloads
[5] http://www.modelversioning.org/updatesite
[6] http://www.modelversioning.org/emf-modeling-operations

AMOR Conflict Detection

The prototypical implementation of the conflict detection of AMOR depends on the components provided in EMO and allows to detect atomic operation conflicts (cf. Section 6.1), as well composite operation conflicts and warnings (cf. Section 6.2). The realization of the detection of signifier-related merge issues (cf. Section 6.3), however, has not been implemented completely so far and is left to future work.

The sources of AMOR's conflict detection component with arround 20.000 lines of code (including comments) are available at EclipseLabs[7]. In this project repository, we also provide some example versioning scenarios for testing the conflict detection and more information on how to use the conflict detection component.

The component itself is primarily designed to be programmatically integrated into a traditional versioning system such as SVN or Git in order to replace their text-based conflict detection techniques with the more adequate model-based techniques presented in this thesis. Nevertheless, we also provide an Eclipse plug-in, which allows to test the conflict detection capablilities independently of a traditional versioning system. Therefore, we implemented a user interface for Eclipse, which allows to execute the conflict detection as described in the documentation at the EclipseLabs project page. The output of the conflict detection is a conflict report, which can be inspected using a dedicated viewer in Eclipse. The plug-ins realizing the conflict detection and the user interfaces for invoking it can be installed in Eclipse Helios (version 3.6) using our update site[8].

EMF Profiles

For annotating occurred conflicts directly in the model on top of the concrete syntax, we developed a novel annotation mechanism called EMF Profiles [LWWC11], which can be thought of as an adaptation of the UML profile concept to DSMLs residing in EMF. Profiles have been a key enabler for the success of UML by providing a lightweight language-inherent extension mechanism, which is expressive enough to cover an important subset of adaptation scenarios. We believe a similar concept for DSMLs would provide an easier extension mechanism, which has been so far neglected by current metamodeling tools. Apart from direct metamodel profiles, EMF Profiles also support reusable profile definition mechanisms whereby profiles are defined independently of any DSML and, later on, coupled with all DSMLs that can benefit from these profiles.

EMF Profiles can be installed easily using the dedicated update site[9]. The sources of EMF Profiles are available from our project web page[10] at EclipseLabs. For more information on this project as well as for screencasts and documentation, please consult the project web site[11].

[7] `http://code.google.com/a/eclipselabs.org/p/amor-conflict-detection`
[8] `http://www.modelversioning.org/updatesite`
[9] `http://www.modelversioning.org/emf-profiles-updatesite`
[10] `http://code.google.com/a/eclipselabs.org/p/emf-profiles`
[11] `http://www.modelversioning.org/emf-profiles`

Ecore Mutator

The Ecore Mutator is an EMF-based framework to randomly mutate EMF models conforming to an Ecore metamodel in order to test, benchmark, and evaluate tools and frameworks related to model matching, differencing, tracking, etc. Therefore, this framework provides several mutations, which, for example, add new instances of randomly selected meta classes in the metamodel, remove existing model elements from the model, change attribute values in existing model elements, etc. Furthermore, it provides a specific mutation type, which also allows for automatically replaying pre-specified change scripts. Moreover, the framework may be extended easily by custom mutations. Having configured the set of required mutations for a certain test scenario, the Ecore Mutator randomly applies them for a configured number of times in order to randomly evolve a specified model. The sources of the Ecore Mutator are hosted at our project web page[12] at EclipseLabs.

[12] `http://code.google.com/a/eclipselabs.org/p/ecore-mutator`

List of Figures

Bibliography

[ABJ+10] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: Advanced Concepts and Tools for In-place EMF Model Transformations. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6395 of *LNCS*, pages 121–135. Springer, 2010.

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

[ABV92] M. Akşit, L. Bergmans, and S. Vural. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, volume 615 of *LNCS*, 1992.

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Proceedings of the European Conference on Model Driven Architecture – Foundations and Applications, (ECMDA-FA'06)*, volume 4066 of *LNCS*, pages 361–375. Springer, 2006.

[And05] K. Andresen. *Design and Use Patterns of Adaptability in Enterprise Systems*, volume 5. GITO mbH Verlag, 2005.

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

[AP03] M. Alanen and I. Porres. Difference and union of models. In *Proceedings of the International Conference on the Unified Modeling Language (UML'03)*, volume 2863 of *LNCS*, pages 2–17. Springer Verlag, 2003.

[BBC10] S.C. Barrett, G. Butler, and P. Chalin. Mirador: A Synthesis of Model Matching Strategies. In *Proceedings of the International Workshop on Model Comparison in Practice (IWMCP'10)*, pages 2–10. ACM, 2010.

[BBFJ11] N. Bencomo, G. Blair, F. Fleurey, and C. Jeanneret, editors. *Proceedings of the Workshop "Models@Run.time" @ MoDELS'11*. CEUR-WS, 2011.

[BDN+07] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Squeak by Example*. Square Bracket Associates, 2007.

[Bec03] K. Beck. *Test-driven Development: By Example*. Addison-Wesley Professional, 2003.

[BEK+06] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, and E. Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, volume 4199 of *LNCS*, pages 425–439. Springer, 2006.

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

[BG03] K. Beck and E. Gamma. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley, 2003.

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

[BKL+11b] P. Brosch, H. Kargl, P. Langer, M. Seidl, K. Wieland, M. Wimmer, and G. Kappel. Conflicts as First-Class Entities: A UML Profile for Model Versioning. In *Proceedings of the Models in Software Engineering Workshops @ MODELS'10, Reports and Revised Selected Papers*, volume 6627 of *LNCS*, pages 184–193. Springer, 2011.

[BKnIN09] O. Ben-Kiki and C. Evansa nd I. Net. YAML Ain't Markup Language (YAML), Version 1.2. `http://yaml.org/spec`, 2009.

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

[BLH01] T. Berners-Lee and J. Hendler. Scientific Publishing on the Semantic Web. *Nature*, 410:1023–1024, 2001.

[BLS+09] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger. An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, volume 5795 of *LNCS*, pages 271–285. Springer, 2009.

[BLS+10a] P. Brosch, P. Langer, M. Seidl, K. Wieland, and M. Wimmer. Colex: A Web-based Collaborative Conflict Lexicon. In *Proceedings of the International Workshop on Model Comparison in Practice @ TOOLS'10*, pages 42–49. ACM, 2010.

[BLS+10b] P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, and G. Kappel. Concurrent modeling in early phases of the software development life cycle. In *Proceedings of the Collaboration Researchers' International Working Group Conference on Collaboration and Technology (CRIWG'10)*, pages 129–144. Springer, 2010.

[BLSW09] P. Brosch, P. Langer, M. Seidl, and M. Wimmer. Towards End-User Adaptable Model Versioning: The By-Example Operation Recorder. In *Proceedings of the Workshop on Comparison and Versioning of Software Models @ ICSE'09*. IEEE, 2009.

[BM07] P.A. Bernstein and S. Melnik. Model Management 2.0: Manipulating Richer Mappings. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*, pages 1–12, 2007.

[BMMM08] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting Model Inconsistency Through Operation-based Model Construction. In *Proceedings of the International Conference on Software Engineering (ICSE'08)*, pages 511–520. ACM, 2008.

[Boo90] G. Booch. *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc., 1990.

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

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

[BSF02] M. Boger, T. Sturm, and P. Fragemann. Refactoring Browser for UML. In *Proceedings of the International Conference NetObjectDays (NODe'02)*, volume 2591 of *LNCS*, pages 366–377. Springer, 2002.

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

[BV09] Z. Balogh and D. Varró. Model Transformation by Example Using Inductive Logic Programming. *Software and System Modeling*, 8(3):347–364, 2009.

[BW07] T. Baar and J. Whittle. On the Usage of Concrete Syntax in Model Transformation Rules. In *Proceedings of the Ershov Memorial Conference*, volume 4378 of *LNCS*, pages 84–97. Springer, 2007.

[Cab07] J. Cabot. From Declarative to Imperative UML/OCL Operation Specifications. In *Proceedings of the International Conference on Conceptual Modeling (ER'07)*, volume 4801 of *LNCS*, pages 198–213. Springer, 2007.

[CdLG+09] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al. Software Engineering for Self-adaptive Systems: A Research Roadmap. *Software Engineering for Self-Adaptive Systems*, 5525:1–26, 2009.

[CDRP07] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Metamodel Independent Approach to Difference Representation. *Journal of Object Technology*, 6(9):165–185, 2007.

[CDRP08] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Managing Model Conflicts in Distributed Development. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'08)*, volume 5301 of *LNCS*, pages 311–325. Springer, 2008.

[CH06] K. Czarnecki and S. Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

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

[CMT06] J.S. Cuadrado, J.G. Molina, and M.M. Tortosa. RubyTL: A Practical, Extensible Transformation Language. In *Proceedings of the European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA'06)*, volume 4066 of *LNCS*, pages 158–172. Springer, 2006.

[CR36] A. Church and J.B. Rosser. Some Properties of Conversion. *Transactions of the American Mathematical Society*, pages 472–482, 1936.

[CR04] E. Clayberg and D. Rubel. *Eclipse: Building Commercial-quality Plug-ins*. Pearson Higher Education, 2004.

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

[DCMJ06] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated Detection of Refactorings in Evolving Components. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *LNCS*, pages 404–428. Springer, 2006.

[DDN00] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding Refactorings via Change Metrics. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, pages 166–177. ACM, 2000.

[DHN09] X. Dolques, M. Huchard, and C. Nebut. From Transformation Traces to Transformation Rules: Assisting Model Driven Engineering Approach with Formal Concept Analysis. In *Proceedings of the International Conference on Conceptual Structures (ICCS'09)*, volume 483, pages 15–29. CEUR-WS, 2009.

[Dij82] E.W. Dijkstra. On the Role of Scientific Thought. *Selected Writings on Computing: A Personal Perspective*, pages 60–66, 1982.

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

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

[dLV02] J. de Lara and H. Vangheluwe. AToM^3: A Tool for Multi-formalism and Metamodelling. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE'02)*, volume 2306 of *LNCS*, pages 174–188. Springer, 2002.

[DMDH04] A.H. Doan, J. Madhavan, P. Domingos, and A. Halevy. Ontology Matching: A Machine Learning Approach. *Handbook on Ontologies*, pages 385–516, 2004.

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

[DS16] F. De Saussure. Nature of the Linguistic Sign. *Course In General Linguistics*, 1916.

[EA04] T. Ekman and U. Asklund. Refactoring-aware Versioning in Eclipse. *Electronic Notes in Theoretical Computer Science*, 107:57–69, 2004.

[Egy11] A. Egyed. Automatically Detecting and Tracking Inconsistencies in Software Design Models. *IEEE Transactions on Software Engineering*, 37(2):188–204, 2011.

[Ehr79] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars (A Survey). In *Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Springer, 1979.

[ELH+05] J. Estublier, D. Leblang, A. Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber. Impact of Software Engineering Research on the Practice of Software Configuration Management. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 14(4):383–430, 2005.

[EMC] EMF Compare Website. `http://www.eclipse.org/emf/compare`, last visited: 2011-07-27.

[ES07] J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer, 2007.

[FBB+99] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[FBJ+05] M.D.Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. AMW: A Generic Model Weaver. In *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM'05)*, 2005.

[Fea89] M.S. Feather. Detecting Interference When Merging Specification Evolutions. In *Proceedings of the International Workshop on Software Specification and Design (IWSSD'89)*, pages 169–176. ACM, 1989.

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

[FHLN08] J. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel Matching for Automatic Model Transformation Generation. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'08)*, volume 5301 of *LNCS*, pages 326–340. Springer, 2008.

[FR07] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Proceedings of Future of Software Engineering @ ICSE'07*, pages 37–54, 2007.

[FS97] M. Fayad and D.C. Schmidt. Object-oriented Application Frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[FV07] M.D. Del Fabro and P. Valduriez. Semi-automatic Model Integration Using Matching Transformations and Weaving Models. In *Proceedings of the Annual ACM Symposium on Applied Computing (SAC'07)*, pages 963–970, 2007.

[Gab11] S. Gabmeyer. Formalization of the Operation Recorder based on Graph Transformation Theory. Master's thesis, Vienna University of Technology, 2011.

[Gal05] B. Gallagher. Matching Structure and Semantics: A Survey on Graph-based Pattern Matching. In *Proceedings of the National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference (AAAI'05)*. AAAI Press, 2005.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Professional, 1995.

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

[GKLE10] C. Gerth, J. M. Küster, M. Luckey, and G. Engels. Precise Detection of Conflicting Change Operations Using Process Model Terms. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6395 of *LNCS*, pages 93–107. Springer, 2010.

[GMnGSFF09] I. García-Magariño, J.J. Gómez-Sanz, and R. Fuentes-Fernández. Model Transformation By-Example: An Algorithm for Generating Many-to-Many Transformation Rules in Several Model Transformation Languages. In *Proceedings of the International Conference on Theory and Practice of Model Transformations (ICMT'09)*, volume 5563 of *LNCS*, pages 52–66. Springer, 2009.

[GS03] J. Greenfield and K. Short. Software Factories: Assembling Applications With Patterns, Models, Frameworks and Tools. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, pages 16–27. ACM, 2003.

[Gsc02] T. Gschwind. *Adaptation and Composition Techniques for Component-based Software Engineering*. PhD thesis, Vienna University of Technology, 2002.

[GTK+07] J. Gray, J.P. Tolvanen, S. Kelly, A. Gokhale, S. Neema, and J. Sprinkle. Domain-specific Modeling. *Handbook of Dynamic System Modeling*, 2007.

[Har87] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HBJ09] M. Herrmannsdoerfer, S. Benz, and E. Juergens. COPE – Automating Coupled Evolution of Metamodels and Models. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'09)*, volume 5653 of *LNCS*, pages 52–76. Springer, 2009.

[HC01] G.T. Heineman and W.T. Councill. *Component-based Software Engineering: Putting the Pieces Together*, volume 17. Addison-Wesley, 2001.

[Hec06] R. Heckel. Graph Transformation in a Nutshell. *Electronic Notes in Theoretical Computer Science*, 148(1):187–198, 2006.

[HGR10] M. Hartung, A. Gross, and E. Rahm. Rule-based Generation of Diff Evolution Mappings between Ontology Versions. *Computing Research Repository*, 1010.0122, 2010.

[HK10] M. Herrmannsdoerfer and M. Koegel. Towards a Generic Operation Recorder for Model Evolution. In *Proceedings of the International Workshop on Model Comparison in Practice @ TOOLS'10*. ACM, 2010.

[HKT02] R. Heckel, J. Küster, and G. Taentzer. Confluence of Typed Attributed Graph Transformation Systems. In *Proceedings of the International Conference on Graph Transformations (ICGT'02)*, volume 2505 of *LNCS*, pages 161–176. Springer, 2002.

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

[HO93] W.H. Harrison and H.L. Ossher. Subject-Oriented Programming — A Critique of Pure Objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*. ACM, 1993.

[Hol04] S. Holzner. *Eclipse Cookbook*. O'Reilly Media, Inc., 2004.

[HR04] D. Harel and B. Rumpe. Meaningful Modeling: What's the Semantics of Semantics? *Computer*, 37(10):64–72, 2004.

[HRW09] M. Herrmannsdoerfer, D. Ratiu, and G. Wachsmuth. Language Evolution in Practice: The History of GMF. In *Proceedings of the International Conference on Software Language Engineering (SLE'09)*, LNCS. Springer, 2009.

[HT00] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, 2000.

[JABK08] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. ATL: A Model Transformation Tool. *Sci. Comput. Program.*, 72(1-2):31–39, 2008.

[KB98] W. Kozaczynski and G. Booch. Component-based Software Engineering. *IEEE Software*, 15(5):34–36, 1998.

[KBK95] M. Keil, P.M. Beranek, and B.R. Konsynski. Usefulness and Ease of Use: Field Study Evidence Regarding Task Considerations. *Decision Support Systems*, 13(1):75–91, 1995.

[KDRPP09] D.S. Kolovos, D. Di Ruscio, A. Pierantonio, and R.F. Paige. Different Models for Model Matching: An Analysis of Approaches to Support Model Differencing. In *Proceedings of the International Workshop on Comparison and Versioning of Software Models @ ICSE'09*. IEEE, 2009.

[KGFE08] J. Küster, C. Gerth, A. Förster, and G. Engels. Detecting and Resolving Process Model Differences in the Absence of a Change Log. In *Business Process Management*, volume 5240 of *LNCS*, pages 244–260. Springer, 2008.

[KHvW+10] M. Koegel, M. Herrmannsdoerfer, O. von Wesendonk, J. Helming, and B. Bruegge. Merging Model Refactorings – An Empirical Study. In *Proceedings of the Workshop on Model Evolution @ MoDELS'10*, 2010.

[KHWH10] M. Koegel, M. Herrmannsdoerfer, O. Wesendonk, and J. Helming. Operation-based Conflict Detection on Models. In *Proceedings of the International Workshop on Model Comparison in Practice @ TOOLS'10*. ACM, 2010.

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

[KKK+07] G. Kappel, H. Kargl, G. Kramler, A. Schauerhuber, M. Seidl, M. Strommer, and M. Wimmer. Matching Metamodels With Semantic Systems – An Experience Report. In *Proceedings of the Workshop Model Management und Metadaten-Verwaltung @ BTW'07*, 2007.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the Europeen Conference on Object-Oriented Programming (ECOOP'97)*, 1997.

[KLR+11] G. Kappel, P. Langer, W. Retzschitzegger, W. Schwinger, and M. Wimmer. Model Transformation By-Example: A Survey of the First Wave. *Accepted for publication*, 2011.

[KMS+09] T. Kühne, G. Mezei, E. Syriani, H. Vangheluwe, and M. Wimmer. Explicit Transformation Modeling. In *Proceedings of the Workshop on Models in Software Engineering @ MoDELS'09*, volume 6002 of *LNCS*, pages 240–255. Springer, 2009.

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

[Kol09] D. Kolovos. Establishing Correspondences between Models with the Epsilon Comparison Language. In *Proceedings of the International Conference on Model Driven Architecture-Foundations and Applications (ECMDA-FA'09)*, volume 5562 of *LNCS*, pages 146–157. Springer, 2009.

[KPP08] D. Kolovos, R. Paige, and F. Polack. The Epsilon Transformation Language. In *Proceedings of the International Conference on Theory and Practice of Model Transformations (ICMT'08)*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.

[KPPR07] D.S. Kolovos, R.F. Paige, F. Polack, and L.M. Rose. Update Transformations in the Small with the Epsilon Wizard Language. *Journal of Object Technology*, 6(9):53–69, 2007.

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

[KRP11] D. Kolovos, L. Rose, and R. Paige. *The Epsilon Book*. Online: `http://www.eclipse.org/gmt/epsilon/doc/book/`, 2011.

[KSB08] M. Kessentini, H.A. Sahraoui, and M. Boukadoum. Model Transformation as an Optimization Problem. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'08)*, volume 5301 of *LNCS*, pages 159–173. Springer, 2008.

[KSB10] M. Kessentini, H. Sahraoui, and M. Boukadoum. Model Transformation as an Optimization Problem. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6395 of *LNCS*, pages 159–173. Springer, 2010.

[KT08] S. Kelly and J.P. Tolvanen. *Domain-specific Modeling: Enabling Full Code Generation*. Wiley – IEEE, 2008.

[Küh06] T. Kühne. Matters of (Meta-) Modeling. *Software and Systems Modeling*, 5:369–385, 2006.

[KWB03] A.G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[KWN05] Udo Kelter, Jürgen Wehren, and Jörg Niere. A Generic Difference Algorithm for UML Models. In *Software Engineering*, LNI, pages 105–116. GI, 2005.

[KWSK09] H. Kargl, M. Wimmer, M. Seidl, and G. Kappel. SmartMatcher: Improving Automatically Generated Transformations. *Datenbank-Spektrum*, 29:42–52, 2009.

[Lan09] P. Langer. Konflikterkennung in der Modellversionierung. Master's thesis, Vienna University of Technology, 2009.

[Lec04] Stephan Lechner. *Web-scheme Transformers By-Example*. PhD thesis, Johannes Kepler University Linz, 2004.

[Lee89] A.S. Lee. A Scientific Methodology for MIS Case Studies. *MIS quarterly*, pages 33–50, 1989.

[Lev66] V.I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In *Soviet Physics Doklady*, volume 10, pages 707–710, 1966.

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

[Lie96] K.J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.

[Lie01] H. Lieberman. *Your Wish is My Command: Programming By Example*. Morgan Kaufmann Publishers Inc., 2001.

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

[LWB10] P. Langer, K. Wieland, and P. Brosch. Specification, Execution, and Detection of Refactorings for Software Models. In *Proceedings of the Work-in-Progress Session at the 8th International Conference on the Principles and Practice of Programming in Java (PPPJ'10)*. CEUR-WS.org, 2010.

[LWK10] P. Langer, M. Wimmer, and G. Kappel. Model-to-Model Transformations By Demonstration. In *Proceedings of the International Conference on Theory and Practice of Model Transformations (ICMT'10)*, volume 6142 of *LNCS*, pages 153–167. Springer, 2010.

[LWWC11] P. Langer, K. Wieland, M. Wimmer, and J. Cabot. From UML Profiles to EMF Profiles and Beyond. In *Proceedings of the International Conference on Objects, Models, Components, Patterns (TOOLS'11)*, volume 6705 of *LNCS*, pages 52–67. Springer, 2011.

[MCPW08] L. Murta, C. Corrêa, J.G. Prudêncio, and C. Werner. Towards Odyssey-VCS 2: Improvements Over a UML-based Version Control System. In *Proceedings of the International Workshop on Comparison and Versioning of Software Models @ MoDELS'08*, pages 25–30. ACM, 2008.

[Mel04] S.J. Mellor. *MDA Distilled: Principles of Model-driven Architecture*. Addison-Wesley Professional, 2004.

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

[Men06] T. Mens. On the Use of Graph Transformations for Model Refactoring. In *Proceedings of the International Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, volume 4143 of *LNCS*, pages 219–257. Springer, 2006.

[Men08] T. Mens. Introduction and Roadmap: History and Challenges of Software Evolution. In *Software Evolution*, pages 1–11. Springer, 2008.

[Mey88] B. Meyer. *Object Oriented Software Construction*. Prentice Hall, 1988.

[MG06] T. Mens and P. Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

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

[MHPB09] E. Murphy-Hill, C. Parnin, and A.P. Black. How We Refactor, and How We Know It. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*, pages 287–297. IEEE, 2009.

[MSZJ04] H. Ma, W. Shao, L. Zhang, and Y. Jiang. Applying OO Metrics to Assess UML Meta-models. In *Proceedings of the International Conference on the Unified Modelling Language (UML'04)*, volume 3273 of *LNCS*, pages 12–26. Springer, 2004.

[MTR05] T. Mens, G. Taentzer, and O. Runge. Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. *Electronic Notes in Theoretical Computer Science*, 127(3):113–128, 2005.

[MVDSD06] T. Mens, R. Van Der Straeten, and M. D'Hondt. Detecting and Resolving Model Inconsistencies using Transformation Dependency Analysis. *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, pages 200–214, 2006.

[Nie11] O. Nierstrasz. Synchronizing Models and Code. Keynote at the International Conference on Objects, Models, Components and Patterns (TOOLS'11), 2011.

[NMB04] T.N. Nguyen, E.V. Munson, and J.T. Boyland. Object-oriented, Structural Software Configuration Management. In *Companion to the Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 35–36. ACM, 2004.

[NNZ00] U. Nickel, J. Niere, and A. Zündorf. The FUJABA Environment. In *Proceedings of the International Conference on Software Engineering (ICSE'00)*, pages 742–745. IEEE, 2000.

[NRB69] P. Naur, B. Randell, and F.L. Bauer. *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968*. Scientific Affairs Division, NATO, 1969.

[NSC+07] S. Nejati, M. Sabetzadeh, M. Chechik, S. Easterbrook, and P. Zave. Matching and Merging of Statecharts Specifications. In *Proceedings of the International Conference on Software Engineering (ICSE'07)*, pages 54–64. IEEE, 2007.

[OD08] D.L. Olson and D. Delen. *Advanced Data Mining Techniques*. Springer, 2008.

[OMG03] Object Management Group OMG. Unified Modeling Language 2.0 (UML). http://www.omg.org/cgi-bin/doc?ptc/2003-09-15, 09 2003.

[OMG04] Object Management Group OMG. Meta-Object Facility 2.0 (MOF). http://www.omg.org/cgi-bin/doc?ptc/03-10-04, 10 2004.

[OMG05a] Object Management Group OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. Final Adopted Specification. http://www.omg.org/docs/ptc/07-07-07.pdf, 11 2005.

[OMG05b] Object Management Group OMG. Model-driven Architecture (MDA). http://www.omg.org/mda/specs.htm, 04 2005.

[OMG07] Object Management Group OMG. XML Metadata Interchange 2.1.1 (XMI). http://www.omg.org/spec/XMI/2.1.1, 12 2007.

[OMG09] Object Management Group OMG. Business Process Modeling Notation (BPMN), Version 1.2. http://www.omg.org/spec/BPMN/1.2, 01 2009.

[OMG10] Object Management Group OMG. Object Constraint Language (OCL), Version 2.2. http://www.omg.org/spec/OCL/2.2, 02 2010.

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

[Opd92] W.F. Opdyke. *Refactoring Object-oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[Opp05] R. Oppermann. From User-adaptive to Context-adaptive Information Systems. *i-com*, 4(3):4–14, 2005.

[OR97] R. Oppermann and R. Rasher. Adaptability and adaptivity in learning systems. In *Proceedings on Knowledge Transfer*, volume 2, pages 173–179, 1997.

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

[OSG03] Alliance OSGi. *OSGi Service Platform, Release 3*. IOS Press, Inc., 2003.

[Ous98] J.K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. *Computer*, 31(3):23–30, 1998.

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

[Pap03] M. Papazoglou. Service-oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of the 4th International Conference on Web Information Systems Engineering*, pages 1–10. IEEE, 2003.

[Par72] D.L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Par75] D.L. Parnas. Software Engineering or Methods for the Multi-person Construction of Multi-version Programs. In *Proceedings of the Informatik Symposium on Programming Methodology*, volume 23 of *LNCS*, pages 225–235. Springer, 1975.

[Par79] D.L. Parnas. Designing Software for Ease of Extension and Contraction. *Transactions on Software Engineering*, pages 128–138, 1979.

[PI82] R.S. Pressman and D. Ince. *Software Engineering: A Practitioner's Approach*. McGraw-Hill New York, 1982.

[Por05] I. Porres. Rule-based Update Transformations and their Application to Model Refactorings. *Software and System Modeling*, 4(4):368–385, 2005.

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

[RB01] E. Rahm and P.A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10(4):334–350, 2001.

[Rei90] S.P. Reiss. Connecting Tools Using Message Passing in the Field Environment. *Software*, 7(4):57–66, 1990.

[RH09] P. Runeson and M. Höst. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

[Rie00] D. Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, ETH Zürich, 2000.

[RL08] R. Robbes and M. Lanza. Example-Based Program Transformation. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'08)*, volume 5301 of *LNCS*, pages 174–188. Springer, 2008.

[RN88] J.L. Rodgers and W.A. Nicewander. Thirteen Ways to Look at the Correlation Coefficient. *The American Statistician*, 42(1):59–66, 1988.

[Rob07] R. Robbes. Mining a Change-Based Software Repository. In *Proceedings of the Workshop on Mining Software Repositories (MSR'07)*, pages 15–23. IEEE, 2007.

[RSA10] J. Reimann, M. Seifert, and U. Aßmann. Role-Based Generic Model Refactoring. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6395 of *LNCS*, pages 78–92. Springer, 2010.

[RV08] J.E. Rivera and A. Vallecillo. Representing and Operating With Model Differences. *Objects, Components, Models and Patterns*, pages 141–160, 2008.

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

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 903 of *LNCS*, pages 151–163. Springer, 1994.

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

[SE05] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *Journal on Data Semantics IV*, pages 146–171, 2005.

[Sel03] B. Selic. The Pragmatics of Model-driven Development. *Software, IEEE*, 20(5):19–25, 2003.

[SG96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SG08] M. Schmidt and T. Gloetzner. Constructing Difference Tools for Models Using the SiDiff Framework. In *Companion of the International Conference on Software Engineering*, pages 947–948. ACM, 2008.

[SGK+11] Y. Sun, J. Gray, G. Kappel, P. Langer, M. Wimmer, and J. White. *Emerging Technologies for the Evolution and Maintenance of Software Models*, chapter A WYSIWYG Approach to Support Layout Configuration in Model Evolution. IGI Global, 2011.

[SGL+10] Y. Sun, J. Gray, P. Langer, M. Wimmer, and J. White. A WYSIWYG Approach For Configuring Model Layout Using Model Transformations. In *Proceedings of the Workshop on Domain-Specific Modeling*, 2010.

[SGW11a] Y. Sun, J. Gray, and J. White. MT-Scribe: An End-user Approach to Automate Software Model Evolution. In *Proceedings of the International Conference on Software Engineering (ICSE'11), Demonstrations Track*, pages 980–982. ACM, 2011.

[SGW+11b] Y. Sun, J. Gray, C. Wienands, M. Golm, and J. White. A Demonstration-based Approach to Support Live Transformations in a Model Editor. In *Proceedings of the International Conference on Theory and Practice of Model Transformations (ICMT'11)*, volume 6707 of *LNCS*, pages 213–227. Springer, 2011.

[SPLTJ01] G. Sunyé, D. Pollet, Y. Le Traon, and J.M. Jézéquel. Refactoring UML Models. In *Proceedings of the International Conference on the Unified Modeling Language (UML'01)*, volume 2185 of *LNCS*, pages 134–148. Springer, 2001.

[SS02] J.M.C. Smith and D. Stotts. Elemental Design Patterns: A Link Between Architecture and Object Semantics. Technical Report TR02-011, The University of North Carolina at Chapel Hill, 2002.

[SW08] M. Strommer and M. Wimmer. A Framework for Model Transformation By-Example: Concepts and Tool Support. In *Proceedings of the International Conference on Objects, Components, Models and Patterns (TOOLS'08)*, volume 11 of *LNBIP*, pages 372–391. Springer, 2008.

[SWG09] Y. Sun, J. White, and J. Gray. Model transformation by demonstration. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'09)*, volume 5795 of *LNCS*, pages 712–726. Springer, 2009.

[SZ07] C. Schneider and A. Zündorf. Experiences in using Optimisitic Locking in Fujaba. *Softwaretechnik Trends*, 27(2):2007, 2007.

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

[Tae03] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In *Proceedings of the International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *LNCS*, pages 446–453. Springer, 2003.

[TELW10] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer. Conflict Detection for Model Versioning Based on Graph Modifications. In *Proceedings of the International Conference on Graph Transformations (ICGT'02)*, volume 6372 of *LNCS*, pages 171–186. Springer, 2010.

[TELW11] G. Taentzer, C. Ermel, P. Langer, and M. Wimmer. A Fundamental Approach to Model Versioning Based on Graph Modifications. *Submitted to Software and System Modeling*, 2011.

[TMD09] R.N. Taylor, N. Medvidovic, and E.M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, 2009.

[TMO09] R.N. Taylor, N. Medvidovic, and P. Oreizy. Architectural Styles for Runtime Software Adaptation. In *Proceedings of the European Conference on Software Architecture (WICSA/ECSA'09)*, pages 171–180. IEEE, 2009.

[TOHS99] P.L. Tarr, H.L. Ossher, W.H. Harrison, and Stanley M. S., Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the Conference on Software Engineering (ICSE'99)*, pages 107–119, 1999.

[TP05] G.L. Thione and D.E. Perry. Parallel Changes: Detecting Semantic Interferences. In *Proceedings of the International Computer Software and Applications Conference*, pages 47–56. IEEE, 2005.

[Ull76] J.R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[Var06] D. Varró. Model Transformation by Example. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'06)*, volume 4199 of *LNCS*, pages 410–424. Springer, 2006.

[VEdM06] M. Verbaere, R. Ettinger, and O. de Moor. JunGL: A Scripting Language for Refactoring. In *Proceedings of the International Conference on Software Engineering (ICSE'06)*, pages 172–181. ACM, 2006.

[VWV11] S. Vermolen, G. Wachsmuth, and E. Visser. Reconstructing Complex Metamodel Evolution. Technical Report TUD-SERG-2011-026, Delft University of Technology, 2011.

[W3C08] W3C. Extensible Markup Language (XML), Version 1.0. `http://www.w3.org/TR/REC-xml`, 2008.

[W3C09] W3C. XML Schema Definition Language (XSD), Version 1.0. `http://www.w3.org/TR/xmlschema-0/`, 2009.

[WD06] P. Weissgerber and S. Diehl. Identifying Refactorings from Source-Code Changes. In *Proceedings of the International Conference on Automated Software Engineering (ASE'06)*, pages 231–240. IEEE, 2006.

[Wes91] B. Westfechtel. Structure-oriented Merging of Revisions of Software Documents. In *Proceedings of the International Workshop on Software Configuration Management*, pages 68–79. ACM, 1991.

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

[Wie11] K. Wieland. *Conflict-tolerant Model Versioning*. PhD thesis, Vienna University of Technology, 2011.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. The MIT Press, 1993.

[WMA+07] J. Whittle, A. Moreira, J. Araújo, P.K. Jayaraman, A.M. Elkhodary, and R. Rabbi. An Expressive Aspect Composition Language for UML State Diagrams. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, volume 4735 of *LNCS*, pages 514–528. Springer, 2007.

[WSKK07] M. Wimmer, M. Strommer, H. Kargl, and G. Kramler. Towards Model Transformation Generation By-Example. In *Proceedings of the Hawaiian International Conference on Systems Science (HICSS'07)*. IEEE, 2007.

[WVV+01] H. Wache, T. Voegele, U. Visser, H. Stuckenschmidt, G. Schuster, H. Neumann, and S. Hübner. Ontology-based Integration of Information — A Survey of Existing Approaches. In *Proceedings of the Workshop on Ontologies and Information Sharing (IJCAI'01)*, pages 108–117, 2001.

[XS05] Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Proceedings of the International Conference on Automated Software Engineering (ASE'05)*, pages 54–65. ACM, 2005.

[XS06] Z. Xing and E. Stroulia. Refactoring Detection based on UMLDiff Change-Facts Queries. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, pages 263–274. IEEE, 2006.

[ZLG05] J. Zhang, Y. Lin, and J. Gray. Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In *Proceedings of the International Conference on Model-driven Software Development—Research and Practice in Software Engineering*, pages 199–217. Springer, 2005.

[Zlo75] M.M. Zloof. Query-by-Example: the Invocation and Definition of Tables and Forms. In *Proceedings of the International Conference on Very Large Data Bases (VLDB'75)*, pages 1–24. ACM, 1975.

Europass Curriculum Vitae

Personal information

First name / Surname	**Dipl.-Ing. Philip Langer**
Address	Kulmgasse 32/9, 1170 Vienna, Austria
Telephone	+43 650 230 69 99
E-mail	langer@big.tuwien.ac.at

Work experience

Dates	**Since 03/2009**
Occupation or position held	**Researcher (full-time)**
Main activities and responsibilities	Scientific development and dissemination in the area of model-driven engineering, in particular model transformations by demonstration and model versioning based on the *Eclipse Modeling Framework;* Lecturer of several bachelor and master courses; Advisor of practicals and bachelor/master theses;
Name and address of employer	Business Informatics Group, Vienna University of Technology Favoritenstraße 9-11, A-1040 Vienna
Type of business or sector	Research and Education
Dates	**05/2006 – 12/2009**
Occupation or position held	**Software development and consulting (freelancer)**
Main activities and responsibilities	Consulting, design, and development of software solutions and add-ons for SAP BusinessObjects and XCelsuis
Name and address of employer	Mercury business solutions gmbh (in the meantime moved to) Löwelstraße 12/1, A-1010 Vienna
Type of business or sector	Consulting/Business Intelligence
Dates	**09/2008 – 02/2009**
Occupation or position held	**Web development and consulting (freelancer)**
Main activities and responsibilities	Consulting, design, and development of web-based survey applications with Ruby on Rails
Name and address of employer	Prolytic GmbH Donau-City-Straße 1, A-1220 Vienna
Type of business or sector	Marketing Engineering
Dates	**09/2006 – 07/2008**
Occupation or position held	**Teaching Assistant (part-time)**
Main activities and responsibilities	Supporting the bachelor classes “Introduction to Semantic Web”, “Web Application Development”, and “Object-oriented Modeling”
Name and address of employer	Business Informatics Group, Vienna University of Technology Favoritenstraße 9-11, A-1040 Vienna
Type of business or sector	Research and Education
Dates	**01/2004 – 12/2007**
Occupation or position held	**Web Development (freelancer)**
Main activities and responsibilities	Design and development of several web portals and applications: http://www.kidsnet.at, http://www.lehrerweb.at, http://www.ixlarge.at
Name and address of employer	Internet Center of Education (ICE Vienna) Alserbachstraße 23, A-1090 Vienna
Type of business or sector	Internet and Education

Dates	**10/2002 – 05/2003**
Occupation or position held	**Software Development (Military Service)**
Main activities and responsibilities	Design and development of software applications using Java, XML, and Web Services
Name and address of employer	Kommando Führungsunterstützung (IT-Department of the Austrian Army) Stiftsgasse 2a, A-1060 Vienna
Type of business or sector	Military Service

Education and training

Dates	**Since 03/2009**
Principal subjects/occupational skills covered	**Doctoral Program in Computer Science** Model-driven Software Development, in particular *Model Transformations By Demonstration* and *Model Versioning*
Name and type of organisation providing education and training	**Vienna University of Technology** http://www.tuwien.ac.at
Dates	**10/2006 – 02/2009**
Title of qualification awarded	Diplom-Ingenieur (equivalent to Master of Science)
Principal subjects/occupational skills covered	**Wirtschaftsinformatik** Computer Science and Business Administration Emphasis on Model Engineering and Software Engineering
Name and type of organisation providing education and training	**Vienna University of Technology** http://www.tuwien.ac.at
Dates	**10/2003 – 06/2006**
Title of qualification awarded	Bachelor of Science
Principal subjects/occupational skills covered	**Wirtschaftsinformatik** Computer Science and Business Administration Emphasis on Software Engineering, Web Engineering, and Semantic Web
Name and type of organisation providing education and training	**Vienna University of Technology** http://www.tuwien.ac.at

Personal skills and competences

Mother tongue	**German**
Other language(s)	**English**
Social skills and competences	Elected student council advocating around 1000 students (2-year period). Several years of teaching experience including holding lectures, tutorials, practical assignments, and supervising of practicals and BSC/MSC theses.
Computer skills and competences	- Strong skills in several programming languages and platforms - BusinessObjects Designer Certificate - Cisco Networking Certificate (Semester 1 and 2)

Awards

- Frequentis Advancement Award for master thesis (2009)
- 3rd Place Inits Award for master thesis and dissertations (2009)

Publications

An up-to-date list is available at http://www.big.tuwien.ac.at/staff/planger?area=publications

Peer Reviewed Journal Papers

G. Taentzer, C. Ermel, P. Langer, M. Wimmer:
A Fundamental Approach to Model Versioning Based on Graph Modifications: From Theory to Implementation
submitted to Software and Systems Modeling, Springer, 2011.

Peer Reviewed Book Chapters

P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel:
The Past, Present, and Future of Model Versioning
in Emerging Technologies for the Evolution and Maintenance of Software Models, IGI Global, pp. 410-443, 2012.

Y. Sun, J. Gray, P. Langer, G. Kappel, M. Wimmer, J. White:
A WYSIWYG Approach to Support Layout Configuration in Model Evolution
in Emerging Technologies for the Evolution and Maintenance of Software Models, IGI Global, pp. 92-120, 2012.

Peer Reviewed Conference Papers

P. Langer, K. Wieland, M. Wimmer, J. Cabot:
From UML Profiles to EMF Profiles and Beyond
in Proc. of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS'11), Springer, pp. 184-193, 2011.

P. Langer, M. Wimmer, G. Kappel:
Model-to-Model Transformations By Demonstration
in Proc. of the 3rd International Conference on Model Transformation (ICMT'10), Springer, pp. 153-167, 2010.

P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel:
Concurrent Modeling in Early Phases of the Software Development Life Cycle
in Proc. of the 16th Collaboration Researchers' International Working Group Conference on Collaboration and Technology (CRIWG'10), Springer, pp. 129-144, 2010.

C. Pichler, P. Langer, M. Wimmer, C. Huemer, B. Hofreiter:
Registry Support for Core Component Evolution
in Proc. of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA'10), IEEE Computer Society, 2010.

P. Brosch, G. Kappel, M. Seidl, K. Wieland, M. Wimmer, H. Kargl, P. Langer:
Adaptable Model Versioning in Action
in Proc. of Modellierung'10, GI, LNI 161, pp. 221-236, 2010.

G. Taentzer, C. Ermel, P. Langer, M. Wimmer:
Conflict Detection for Model Versioning Based on Graph Modifications in Proc. of the 5th International Conference on Graph Transformations (ICGT'10), Springer, pp. 171-186, 2010.

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

P. Brosch, M. Seidl, K. Wieland, M. Wimmer, and P. Langer:
We can work it out: Collaborative Conflict Resolution in Model Versioning
in Proc. of the 11th European Conference on Computer Supported Cooperative Work (ECSCW'09), Springer, pp. 207-214, 2009.

Peer Reviewed Workshop Papers and Poster Presentations

P. Langer, K. Wieland, P. Brosch
Specification, Execution, and Detection of Refactorings for Software Models
in Proc. of the WiP Session at the 8th International Conference on the Principles and Practice of Programming in Java, CEUR-WS.org, 2010.

Y. Sun, J. Gray, P. Langer, M. Wimmer, J. White:
A WYSIWYG Approach for Configuring Model Layout using Model Transformations
in 10th Workshop on Domain-Specific Modeling @ SPLASH'10, 2010.

P. Brosch, H. Kargl, P. Langer, M. Seidl, K. Wieland, M. Wimmer, G. Kappel:
Conflicts as First-Class Entities: A UML Profile for Model Versioning
in Proc. of the Models in Software Engineering Workshops and Symposia @ MODELS'10, Springer, pp. 184-193, 2010.

P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer:
Colex: A Web-based Collaborative Conflict Lexicon
in Proc. of the Model Comparison in Practice Workshop @ TOOLS'10, ACM, pp. 42-49, 2010.

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

K. Altmanninger, P. Brosch, G. Kappel, P. Langer, M. Seidl, K. Wieland, M. Wimmer:
Why Model Versioning Research is Needed!? An Experience Report
in Proc. of Joint MoDSE-MCCM 2009 Workshop - Models and Evolution, 2009.

P. Brosch, P. Langer, M. Seidl, K. Wieland, M. Wimmer:
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'09), ACM, pp. 791-792, 2009.

P. Brosch, P. Langer, M. Seidl, M. Wimmer:
Towards End-User Adaptable Model Versioning: The By-Example Operation Recorder
in Proc. of the 31st International Conference on Software Engineering (ICSE'09), IEEE, pp. 55-60, 2009.

P. Brosch, M. Seidl, K. Wieland, M. Wimmer, P. Langer:
By-example Adaptation of the Generic Model Versioning System AMOR:
How to Include Language-specific Features for Improving the Check-in Process
in Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'09), ACM, pp. 739-740, 2009.