Calculation and Propagation of Model Changes Based on User-level Edit Operations

A Foundation for Version and Variant Management in Model-driven Engineering

Genehmigte

DISSERTATION

zur Erlangung des Grades eines

Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
Dipl.-Inform. Timo Kehrer

eingereicht bei der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen
Siegen 2015

Gedruckt auf alterungsbeständigem holz- und säurefreiem Papier.

Als Dissertation genehmigt von
der Naturwissenschaftlich-Technischen Fakultät
der Universität Siegen

Einreichung	20. Mai 2015
Mündl. Prüfung	20. August 2015
Dekan	Prof. Dr. Dr. h. c. Ullrich Pietsch
1. Gutachter	Prof. Dr. Udo Kelter, Universität Siegen
2. Gutachter	Prof. Dr. Gabriele Taentzer, Universität Marburg

Abstract

Model-driven engineering (MDE) has become a widespread approach for developing software in many application domains. Models are primary development documents in MDE and subject to continuous evolution. Models therefore have many versions and variants during system lifetime. Thus, the specification and recognition of changes in models is the key to understand and manage the evolution of a model-based system.

However, currently available model versioning tools operate on low-level, sometimes tool-specific model representations which can be considered as an implementation of the abstract syntax graph (ASG) of a model. Moreover, primitive graph edit operations are used to specify model changes. This leads to two serious problems: Firstly, the resulting model differences are hard to understand for normal tool users who are not familiar with the internal, graph-based representation of models and the related types of nodes and edges which are usually defined by a meta-model. Secondly, generic graph operations lead to further problems in change propagation scenarios since they can violate consistency constraints in ASGs. In the worst case, which particularly applies to all kinds of visual models, the synthesized result can no longer be opened in standard visual editors and must be corrected based on the serialized data format (e.g. XML) by using low-level textual editors, which is obviously a tedious task prone to errors.

Model differences should therefore be based on edit operations which are understandable for modelers and which preserve elementary consistency constraints in the sense that models remain displayable in visual editors. Edit operations which are offered as editing commands in standard editors or modern refactoring tools are examples of this. The tight integration of editing and versioning tools requires consistent specifications of edit operations; this integration is a largely open problem. The main objective of this thesis is to provide a solution for this problem and to systematically lift model versioning concepts, algorithms and tools to a higher level of abstraction.

The central idea is to formally specify the available edit operations for a given type of models as transformation rules using the model transformation language Henshin

which is based on graph transformation concepts. These formal specifications are automatically transformed into recognition rules being used by a lifting algorithm which recognizes edit operations in a low-level difference of two model versions. The approach has been implemented and evaluated in a framework which is known as SiLift and which is based on the widely used Eclipse Modeling Project.

Kurzfassung

Modellbasierte Softwareentwicklung ist in einigen Domänen inzwischen gängige Praxis. Modelle sind hier primäre Artefakte. Sie entwickeln sich daher ständig weiter und existieren im Laufe ihrer Evolution in zahlreichen Versionen und Varianten. Die Spezifikation und Erkennung von Änderungen an Modellen sind somit fundamentale Voraussetzung, um die Evolution modellbasierter Systeme zu verstehen und zu kontrollieren.

Derzeitig verfügbare Werkzeuge des Versions- und Variantenmanagements für Modelle arbeiten jedoch auf systemnahen, fallweise werkzeugspezifischen Repräsentationen von Modellen, letzten Endes einer Implementierung des abstrakten Syntaxgraphen (ASG) eines Modells. Ferner werden primitive Graphoperationen zur Beschreibung von Änderungen an Modellen unterstellt. Dies führt zu zwei wesentlichen Problemen: Zum einen ist die Darstellung solcher "low-level" Änderungen meist unverständlich, ohne Kenntnisse der internen, ASG-basierten Repräsentation der Modelle teilweise sogar unmöglich. Zum anderen birgt die Anwendung von low-level Änderungen im Rahmen der Propagation von Änderungen die Gefahr der Synthetisierung inkonsistenter Modelle. Im schlimmsten Fall kann ein Modell so inkorrekt werden, dass es nicht mehr mit Standard-Modelleditoren verarbeitet werden kann. Dies gilt insbesondere für visuelle Modelle, welche in diesem Fall nur noch mit einfachen textuellen Editoren auf Basis der Repräsentation des serialisierten Datenformats (z.B. XML) bearbeitet und korrigiert werden können.

Modelldifferenzen sollten daher auf für den Benutzer verständlichen, konsistenzerhaltenden Editieroperationen basieren, wie sie bspw. von Modelleditoren oder modernen Refactoring-Werkzeugen angeboten werden. Zur Erkennung derartiger Editieroperationen existieren bislang nur erste Ansätze. Die enge Integration von Editier- und Differenzwerkzeugen erfordert konsistente Spezifikationen der Editieroperationen; diese Integration ist ein offenes wissenschaftliches Problem. Ziel dieser Arbeit ist es, dieses Integrationsproblem zu lösen und die Versionierungskonzepte, -algorithmen und

-werkzeuge von einem derzeit systemnahen Niveau auf ein möglichst hohes, benutzernahes Abstraktionsniveau anzuheben.

Kernidee des im Rahmen dieser Dissertation entwickelten Ansatzes ist es, die für einen Modelltyp verfügbaren Editieroperationen formal zu spezifizieren. Hierzu wird die Graphersetzungs- und Modelltransformationssprache Henshin genutzt. Editierregeln werden automatisiert in Erkennungsregeln übersetzt, welche zur Erkennung von Editieroperationen in low-level Differenzen genutzt werden. Eine Referenzimplementierung des Ansatzes ist im Rahmen des Eclipse-basierten Frameworks SiLift verfügbar und wurde in zahlreichen Fallstudien evaluiert.

Acknowledgements

This thesis would have been impossible without the contributions of many people. I would like to thank all of them for supporting me directly or indirectly during the last years.

First of all, I am very thankful to my supervisor, Prof. Dr. Udo Kelter, who has guided me with his knowledge and experience while giving me the freedom to work in an independent and self-organized way. Furthermore, I thank Prof. Dr. Gabriele Taentzer for co-supervising my thesis. She has been involved in my research work from the very beginning and her continuous feedback has always been an invaluable contribution to this thesis.

It was a pleasure to be part of the Software Engineering Group at the University of Siegen. Sincere thanks are given to my (former) colleagues for proofreading parts of my thesis and countless enlightening discussions: Dr. Stefan Berlik, Dr. Christian Köhler, Christopher Pietsch, Pit Pietsch, Dennis Reuling, Michaela Rindt, Maik Schmidt, Dr. Sven Wenzel and Dr. Hamed Shariat Yazdi. Special thanks goes to Frank Schuh for his technical support, and to Roswitha Eifler, Nina Weyand and Katharina Zetzsche for their organizational support. Moreover, I would like to thank the students Dennis Koch, Manuel Ohrndorf, Tim Sollbach and Manuel Wörmann for their great work in the SiLift project.

In addition, I would also like to thank the following researchers with whom I had the honor to collaborate in recent years and who inspired my work in one way or the other: Dr. Thorsten Arendt, Kristopher Born, Johannes Bürdek, Prof. Dr. Marsha Chechik, Prof. Dr. Christian Gerth, Sinem Getir, Prof. Dr. Lars Grunske, Jens Folmer, Prof. Dr. Gerti Kappel, Matthias Kowal, Dr. Christian Krause, Dr. Philip Langer, Christoph Legat, Sascha Lity, Dr. Malte Lochau, Lukas Märtin, Dr. Thomas Ruhroth, Dr. Rick Salay, Prof. Dr. Ina Schaefer, Prof. Dr. Andy Schürr, Daniel Strüber, Prof. Dr. Matthias Tichy, Prof. Dr. Birgit Vogel-Heuser and Dr. Manuel Wimmer.

Most importantly, however, this thesis would have been not possible without the love and patience of my family. In particular, I am deeply grateful to Tanja and Hanna for their patient care, for giving me the confidence to go on, and for simply filling my life with joy and happiness.

Last but not least, I would like to thank the DFG (German Research Foundation) for partially funding this work under the Priority Programme SPP1593: Design For Future - Managed Software Evolution.

Contents

1 Introduction **1**
1.1 Model-driven Engineering . . . 1
1.2 Model Evolution, Version and Variant Management . . . 3
1.3 Problem Motivation . . . 5
1.4 Contributions . . . 6
1.5 Thesis-related List of Publications . . . 10
1.6 Thesis Outline . . . 12

2 Model Version and Variant Management **15**
2.1 Motivating Scenarios . . . 15
2.1.1 SC1: Understanding Model Changes . . . 15
2.1.2 SC2: Updating a Copy of an Original Model . . . 18
2.1.3 SC3: Cherrypicking Changes . . . 18
2.1.4 SC4: Propagation of Changes in a Product Family . . . 19
2.1.5 SC5: Workspace Updates of Models . . . 22
2.2 A Feature-oriented Domain Analysis . . . 24
2.2.1 Modeling Languages and MDE Development Environments . . . 25
2.2.2 User-level Edit Operations . . . 29
2.2.3 Difference Calculation and Presentation . . . 30
2.2.4 Change Propagation . . . 31
2.2.5 Errors and Conflicts . . . 32
2.2.6 Consistency-preservation . . . 34
2.3 State of the Art . . . 35
2.3.1 Matching Algorithms . . . 37
2.3.2 Calculation of a Difference . . . 40
2.3.3 Model Merging . . . 41
2.3.4 Patching vs. 3-Way Merging . . . 45
2.4 Conclusions . . . 47

3 Representation and Editing of Models **51**
3.1 Graph-based Representation of Models . . . 52
3.1.1 Graphs and Graph Mappings . . . 53
3.1.2 Meta-models as Type Graphs . . . 54
3.1.3 Models as Typed Graphs . . . 57
3.1.4 Attributed Graphs . . . 59
3.1.5 Visual Representation of Type and Instance Graphs . . . 59
3.2 Consistency of Models . . . 61
3.3 Rule-based Specification of Edit Operations . . . 64
3.3.1 Informal Specifications of Edit Operations . . . 65
3.3.2 Implementing Edit Operations in Henshin . . . 66
3.3.3 Potential Conflicts and Dependencies . . . 70

4 Semantic Lifting of Model Differences **75**
4.1 Low-level Differences . . . 77
4.2 Semantic Change Sets . . . 82
4.2.1 Rule-based Specification of Semantic Change Sets . . . 83
4.2.2 Generation of Recognition Rules . . . 85
4.2.3 Non-static Change Patterns . . . 89
4.3 Edit Operation Recognition . . . 90
4.3.1 Rule Application Strategy . . . 90
4.3.2 Postprocessing . . . 91
4.4 Restrictions of the Approach . . . 92
4.4.1 Transient Effects . . . 92
4.4.2 Discussion . . . 93

5 Generation of Edit Scripts **97**
5.1 Edit Scripts . . . 97
5.2 Prerequisite: Tracing of Edit Rule Elements . . . 100
5.3 Retrieval of Actual Parameters . . . 101
5.4 Dependency Analysis . . . 103
5.4.1 Dependencies of Kind create/use and use/delete . . . 105
5.4.2 Dependencies of Kind delete/forbid and forbid/create . . . 107

6 Propagation of Changes based on Consistency-preserving Edit Scripts **113**
6.1 Using Edit Scripts as Model Patches . . . 115
6.1.1 Representation of Edit Scripts . . . 115
6.1.2 Guided Editing of an Initial Version of an Edit Script . . . 117
6.1.3 Distribution of Edit Scripts . . . 118
6.2 Error and Conflict Detection . . . 118
6.2.1 Detection of Wrongly Chosen Arguments . . . 119
6.2.2 Warnings against Blind Overwriting of Local Changes . . . 120
6.3 Interactive Application of Edit Scripts . . . 123
6.3.1 Patch Application without Common Base Version . . . 124
6.3.2 Updating Workspace Copies by Patching . . . 126

7 Creation of a Set of Consistency-preserving Edit Rules **129**
7.1 Mandatory Edit Rules 131
7.2 Basic Design Decisions 132
7.3 Generation of Consistency-preserving Edit Rules 134
7.3.1 Preparations and Prerequisites 134
7.3.2 Generation of Creation and Deletion Rules 142
7.3.3 Generation of Move and Change Rules 152
7.3.4 Consistency-preservation of Generated Rules 153
7.4 Completeness of the Generated Edit Rule Set 155
7.5 Analysis of Potential Transient Effects 161
7.6 Requirements Induced by Consistency-preserving Edit Rules 162
7.7 Adapting the Generated Rule Set 164
7.7.1 Handling of Cycle-capable Containment Edge Types 164
7.7.2 Supporting Arbitrary Well-formedness Rules 166

8 A Generic Model Versioning Framework **167**
8.1 Configuration of Differencing Tools 168
8.2 Configuration of Patching Tools 170
8.2.1 Core Components and Feature Mappings 170
8.2.2 Implementation Variants 172
8.3 Reference Implementation based on the Eclipse Modeling Project 173
8.3.1 EMF-based Implementation of Meta-models and ASGs 174
8.3.2 Tool Integration 176
8.3.3 Meta-tool Support 177

9 Evaluation **181**
9.1 Evaluation Goals 181
9.2 Case Studies and Example Applications 183
9.2.1 Study I: Comparison and Versioning of Ecore Models 183
9.2.2 Study II: Model Variant Management in Industrial Plant Automation 186
9.2.3 Study III: Documenting and Reasoning about Feature Model Changes 187
9.2.4 Study IV: Understanding Complex Changes in Domain-Specific Models 189
9.2.5 Study V: Statistical Analysis of Changes in Evolving Software Models 189
9.2.6 Study VI: Analyzing the Co-Evolution of Interrelated Models 190
9.2.7 Conclusions and Critical Discussion 191
9.3 Experimental Results 192
9.3.1 Semantic Lifting of Model Differences 193
9.3.2 Generation and Application of Edit Scripts 195
9.3.3 Conflict Detection 198

10 Related Work **201**
10.1 Generic Model Versioning 201
10.2 Language-specific Approaches 204
10.3 Model Repositories 205
10.4 Approaches from other Domains 207

11 Conclusions and Future Work **211**
11.1 Summary 211
11.2 Outlook 213

Bibliography **237**

CHAPTER 1

Introduction

1.1 Model-driven Engineering

Modeling has a long tradition in software engineering in order to deal with the complexity of large-scale software systems. Many modeling languages, notations and techniques such as state charts, markov chains, data flow diagrams, Petri nets etc. have been proposed for different purposes, a brief historical perspective is given in [170]. A classification criterion for models in software engineering is whether they are used in a descriptive or prescriptive way [168].

Descriptive modeling is in line with the frequently cited notion of a model according to Stachowiak [227]. A descriptive model serves as an abstract representation of an original (mapping feature) and thereby reflects a subset of its properties (reduction feature) which are considered as relevant for some specific purpose (pragmatical feature). An original can be, for instance, an already existing system, a development process being applied in a running project, etc. Thus, descriptive models in software engineering usually capture some knowledge for the purpose of documentation, facilitate the effective communication of information to project stakeholders, or enable the analysis of certain quality attributes of a system or process.

Most of the models used in software engineering are *prescriptive*. Here, Stachowiak's definition is extended to allow an original not yet to be existent [153]. Prescriptive models are used as a specification of sth. to be created (e.g. a software system), or sth. to be performed or executed (e.g. a development process). They serve as a construction plan, provide instructions to some activity, are used to simulate a system before it actu-

ally exists or allow us to forecast critical non-functional properties such as performance or security.

Models as primary development artifacts. Model-driven engineering (MDE), also known as model-driven development (MDD), is a software development methodology which has recently gained a lot of interest in many application domains. The core idea is to use prescriptive models as a construction plan of a system to be created. Most often, a variety of models (or sub-models) is used to describe different structural and behavioral aspects of a system. Most of these aspects or "views" are typically specified using a dedicated modeling language which abstracts from the underlying implementation technology and which is intended to be much closer to the problem domain than classical general purpose programming languages [222]. In contrast to their traditional usage as design blueprints or sketches, models in MDE are rather used to generate various types of development artifacts, notably source code, configuration files, test data or other types of models. Consequently, model transformers or compilers generate machine code from models in one or several steps. Models with well-defined execution semantics can also be interpreted directly in a runtime environment. Ideally, developers will no longer see the compiled intermediate models or the final code. Thus, compared to traditional code-centric development, MDE has the potential to increase development productivity, to achieve a higher degree of portability across different runtime environments and to synthesize software which is "correct by construction" through the use of (certified) code generators. Moreover, the use of models enables the application of supplementary quality assurance techniques such as model checking or model-based testing.

Model-driven engineering and its variations. Different approaches to MDE vary substantially with respect to the adopted modeling languages and notations, the transformation chains being used to convert abstract models into executable programs, and the degree to which traditional code-centric development techniques shall be replaced. The latter aspect is sometimes used to distinguish *model-driven* from *model-based* engineering (MBE); while MDE aims at generating most parts of a system, the degree of automation is typically much lower in MBE. Nonetheless, models still play an important role in the development process.

The most prominent MDE variant is probably the Model Driven Architecture (MDA) [47, 194] initiative as promoted by the Object Management Group (OMG). MDA may be defined as the realization of MDE principles around a set of OMG standards like MOF, XMI, OCL, UML and QVT [48]. Moreover, MDA introduces a set of standard notions which are widely adopted by the MDE community. In particular, a dedicated runtime environment and related implementation technologies are referred to as *platform*. Consequently, abstract analysis models which are typically created in early development phases are referred to as *platform-independent models*. Design-level models which can be used to generate executable implementation artifacts are called *platform-specific models*.

There are several other approaches to MDE, all of them having a slightly different focus. In particular, many approaches, e.g. [117, 135], promote the use of *domain-specific modeling languages* (DSMLs). DSMLs have a small scope and formalize the key concepts of a particular domain of interest; developers use DSMLs to build applications using elements which directly represent concepts of a certain problem space.

Conclusions. While the specific characteristics of different MDE variants are largely irrelevant within the scope of this thesis, we can conclude that this development methodology comes with two important characteristics: Firstly, models are systematically used as primary development artifacts throughout the engineering lifecycle. Secondly, MDE typically involves a multitude of different modeling languages being used for different purposes. The latter one significantly distinguishes MDE from computer-aided software engineering (CASE) in the 1980s. Early CASE tools were rarely adopted in practice for several reasons, one of them was that they didn't support specific application domains effectively because their "one-size-fits-all" visual representations were too generic [215].

1.2 Model Evolution, Version and Variant Management

Complex software systems are subject to continuous change and heavily evolve during all stages of their lifecycle. Traditionally, software evolution is considered as a phenomenon which can be observed for software systems being operated for a long period of time [160]. During operation and maintenance, new requirements emerge when a system is used, errors must be corrected, and non-functional properties such as reliability or performance must be improved [44]. Using platform-independent models as primary development artifacts reduces adaptive maintenance cost caused by the evolution of platforms, but does not prevent the models being subject to all other kinds of maintenance. When requirements change, models must change, too. Models of software systems therefore have many versions during system lifetime. Typically, several versions of a model have been rolled-out to customers and evolve as variants in parallel branches. Thus, there is not "the one and only" latest version of a model, but a number of evolving parallel branches.

Besides the traditional, "long-term meaning" of evolution, development artifacts involved in modern software development are subject to evolution from the early beginning as it is common practice to iteratively develop large software systems [159]. Iterative development helps to break down the whole system functionality into smaller, manageable increments and allows a developer to take advantage of what has been learned during earlier iterations [39]. Thus, software development can be seen as an evolutionary process, too [115]. The principle of iterative development obviously also applies to models. Moreover, models of large and complex systems must be collaboratively developed in teams [113]. Collaborative work is usually supported by an opti-

mistic versioning approach based on local workspaces and a central repository, which leads to another form of parallel evolution of model variants which have to be finally consolidated.

As a consequence, models of a model-based system typically have many versions during their lifetime. Two versions of a model can be a revision or a variant of one another. A version v_2 is a *revision* of a version v_1 if v_2 was declared to be the successor of v_1. Revision v_2 is typically produced by incrementally changing v_1. Thus, the chronological evolution of a model is represented by a sequence of revisions. Two versions of a model are *variants* of each other if they model the same system (or aspect of a system) in different ways. Consequently, variants coexist for some period of time. Parallel development leads to variants residing in local workspaces which are intended to be continuously integrated into consolidated versions. Long-living variants coexist and evolve independently of each other in parallel development branches. Thus, each branch has its own history of successive revisions. In this context, the term *variant* is often used as an abstraction that represents the complete branch including its sequence of revisions.

Traditionally, software configuration management (SCM) is the discipline of managing the evolution of large and complex software systems [77, 243]. As a support discipline for project management, SCM activities such as change control, status accounting, reviews and audits are tightly integrated into the project management process. Thus, strict procedures define when to perform a change, the corresponding change requests and their current implementation status are clearly documented. From a software development point of view, SCM introduces *system building* as well as *version and variant management* into the development process, the latter one is addressed in this thesis. Version and variant management helps to maintain overview in large software systems (or system families) evolving into many revisions and variants, and to coordinate developers working in teams [243]. Essential tool support is provided by version management systems, which are also known as versioning, version control or revision control systems. Besides basic storage services including the management of successor relationships between revisions, users and user permissions, and other administrative data, version control systems essentially rely on a set of tool functions which are commonly referred to as *differencing* and *merging* of documents. Conventional version management systems such as CVS [105], Subversion [76] or Git [165] provide these services only for textual documents. Text-based services work satisfactorily for all kinds of textual documents such as source code, build scripts or requirements specifications. Developers are used to edit these documents in their textual representation, and line-based differences between versions of these documents can be easily understood.

However, with the advent of model-driven engineering, visual models became another, non-textual type of documents which are an integral part of the software and which therefore have to be put under version control, too. At first sight, it appears to be a straightforward solution to store visual models in some textual representation, for example based on the XML Metadata Interchange (XMI) [196] standard, and to use a conventional version management system for software models. All the well-known services such as check-in/out to workspaces, retrieval of old versions, transport

of documents via networks, notification, tagging etc. can readily be reused. Everything works fine in this approach, except model differencing, patching and merging. Despite a decade of research in the field of model version and variant management, there is still a substantial lack of basic tool functions for comparing and reliably merging models. Available tools are not nearly as mature and widely applicable as their traditional counterparts. This is frequently being reported as one of the biggest obstacles in the practical application of the MDE paradigm [25, 40, 41, 42, 98]. Consequently, model evolution and model versioning are identified as one of the key challenges in MDE. For instance, the research roadmap in [108] identifies model manipulation and management as one of three major challenges that must be tackled in order to fully realize the MDE vision of software development. Van der Straeten et al. [248] additionally mention model evolution, inconsistency management and collaborative work.

1.3 Problem Motivation

Basic services of tools supporting version management include comparison, merging, and patching of models. Conceptually, these services are based on several notions, the most fundamental one being a *difference* (or delta). Comprehensive conceptual frameworks for versioning have been developed in the context of source code versioning [77, 180]. These frameworks define a *directed delta* (or difference) as a *sequence of (elementary) edit steps* $s_1 \dots s_m$ which, when applied to a document version v_1, yields another version v_2. An edit step invokes an edit operation and supplies appropriate actual parameters.

These definitions leave open how documents are represented conceptually and which edit operations are available for modifying a document. A classical approach is to use textual representations of documents and text editing operations [210, 241]. It has often been suggested to conceptually represent source code as abstract syntax tree (AST) and to use elementary tree editing operations in edit steps, which enables syntactic differencing [268] and usually leads to a better conflict detection compared with textual merging [67]. More advanced approaches propose to use non-elementary transformations such as refactorings as edit operations for ASTs. Directed deltas based on complex edit operations enable "structural merging" [180] which exploits the semantics of complex restructurings of object-oriented programs [84, 164]. While the advantages of structural merging are undisputed, appropriate tool support is difficult to implement and hard to find in practice [99].

The above considerations also apply to models. It is state of the art to consider models conceptually as abstract syntax graphs (ASG) and to use primitive graph operations such as creating/deleting single nodes/edges as edit operations for ASGs. Primitive graph editing operations are generic in the sense that they can operate on arbitrary ASGs. Generic operations are an attractive solution as they are easy to implement and can be used to modify models of any type. However, from a tool user's point of view, the use of primitive edit operations causes several serious problems.

Most notably, describing model changes based on primitive graph operations leads

to low-level differences which are hard to understand for normal tool users. They are usually not familiar with the ASG-based representation of models and the related types of nodes and edges which are typically defined by a meta-model. Such “meta-model based difference reports” are not intuitive and rather confusing for modelers [25]. This problem is further aggravated by the fact that versioning tools work on internal representations of models which can normally be considered as implementations of abstract syntax graphs. Internal representations of models depend on the technologies being used to implement difference tools, notably the specific programming language and, where applicable, modeling frameworks such as the Eclipse Modeling Framework (EMF) [88]. The according “design-level meta-models” are often tool-specific and can deviate from standard meta-models as, for instance, defined by the OMG.

Generic graph operations lead to further problems in patching or merging scenarios as they can violate consistency constraints in ASGs. In the worst case, which particularly applies to all kinds of visual models, the synthesized result can no longer be opened in standard visual editors and must be corrected based on the serialized data format (e.g. XML) by using low-level textual editors, which is obviously a tedious task prone to errors.

Difference tools for models should therefore be based on edit operations which are understandable for modelers and which preserve elementary consistency constraints in the sense that models remain displayable in visual editors. Edit operations which are offered as editing commands in standard editors are an example of this. If available, the set of edit operations should further include complex edit operations which are meaningful from a user's point of view, e.g. complex edit operations being offered by modern refactoring tools. We generally refer to these kinds of edit operations as *user-level edit operations*. These edit operations are *high-level* in the sense that they usually comprise several low-level modifications of an ASG. User-level edit operations are *consistency-preserving* in the sense that they transform a model from one displayable state into another.

Although it is commonly accepted that model versioning tools must be specifically tailored to each modeling language and usage scenario, the set of available edit operations is an important variation point which has received little attention in the literature. In this regard, it is a largely open problem how to use high-level edit operations to calculate, present and handle model differences in the context of model version and variant management. Moreover, the tight integration of editing and versioning tools requires consistent specifications of edit operations; this integration is another open problem. Both problems are addressed in this thesis, the main objectives and contributions are briefly summarized in the following section.

1.4 Contributions

The main problem addressed in this thesis is a significant deficiency of currently available versioning tools for models; they present and handle differences in terms of low-level changes related to internal, sometimes tool-specific representations of models. This

problem is tackled by systematically lifting model versioning concepts, techniques and tools for the calculation and propagation of model changes to a higher level of abstraction based on user-level edit operations (s. Figure 1.1). We use model transformation rules implemented in Henshin [34] in order to specify edit operations in a precise and meaningful way. Such specifications of edit operations play two different roles:

- A *descriptive* role as a means to describe the observed difference between two versions, notably past changes in the history of a model.
- A *prescriptive* role, i.e. as specification of modifications to be performed on an existing model version, notably to propagate changes from one model version to another.

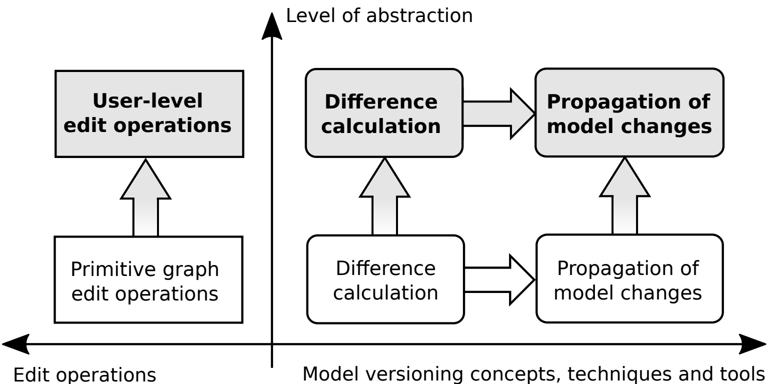


Figure 1.1: Overview of the approach and thesis objective

As a second problem, this thesis addresses the huge challenge that high-quality difference tools for models must be specifically engineered for each modeling language and, more often than not, be adapted to other tools, application scenarios and user preferences. Methods for developing customized difference tools for models with limited implementation effort are therefore a primary concern. Difference tools for models are, arguably, a product family, thus it makes sense to apply development methods for software product lines. In this regard, we basically support variability in two main dimensions:

1. The *model type* (or modeling language), notably the set of edit operations available to modify instances of this type.
2. The *tool functionality* being offered to detect and handle errors and conflicts when propagating model changes from one version to another.

Now that we have presented a high-level overview of the approach and the thesis objective, we outline the contributions of this thesis in greater detail.

Contribution 1: Semantic lifting of model differences. The first contribution of this thesis is a rule-based approach to the *semantic lifting* of low-level model differences obtained from state-of-the-art model comparison tools, which cover only the first two processing steps of the model differencing pipeline shown in the lower part of Figure 1.2. Low-level differences are transformed into representations of user-level edit operations in a fully automated way. Since the execution of an edit operation leads to a well-defined change pattern on an ASG, we use the Henshin transformation engine to find instances of these patterns and to finally group the low-level changes involved in a change pattern to a so-called *semantic change set*. The transformation rules, called recognition rules, which are necessary to semantically lift a low-level difference are automatically derived from the edit rules implementing the available edit operations (s. upper part of Figure 1.2).

Contribution 2: Generation of edit scripts. As a second contribution, we define an extended kind of directed delta to which we refer to as *edit script*. While a directed delta is a sequence of elementary edit steps, an edit script is a complex data structure which contains (a) the involved Henshin rules serving as implementations of edit operations, (b) representations of the detected edit operation invocations, including mappings of the parameters to objects in the low-level difference, and (c) representations of dependencies between edit steps. Each dependency is annotated with information about its reason, e.g. one step produces data used by the later step. From a conceptual point of view, an edit script is a partially ordered set of edit steps. The partial order expresses that an edit step es_1 must be executed before edit step es_2 if $es_1 < es_2$.

We present a technique which, given a set of edit rules which adheres to certain correctness criteria, converts a semantically lifted difference into an executable edit script.

Contribution 3: Consistency-preserving edit scripts. A consistency-preserving edit script is a special kind of edit script which, when being applied as patch to a model, prevents inconsistencies in the patched model. More precisely, a patched model is guaranteed to meet the consistency constraints required by the standard (visual) editor of an MDE development environment. The key idea is to use only consistency-preserving edit operations (CPEOs) on models, which obviously depend on the modeling language and the effective consistency level required by standard editors. To that end, we present a method and semi-automated approach for creating complete sets of CPEOs for a given modeling language and MDE environment. An brief overview is shown in the upper part of Figure 1.2.

Contribution 4: Controlled propagation of model changes. Another contribution of this thesis is a method and graphical user interface (GUI) which enables developers to apply consistency-preserving edit scripts to a target model in a controlled, interactive way. Various types of errors are handled and developers are enabled

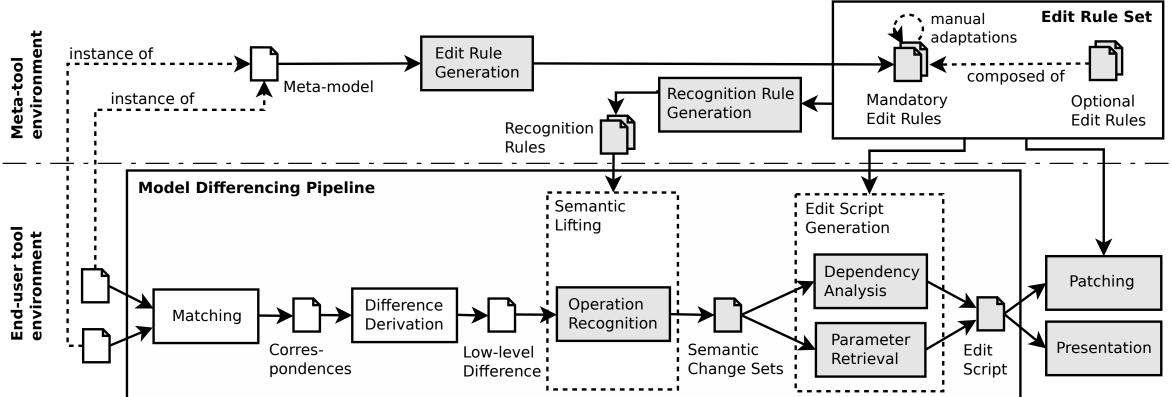


Figure 1.2: Overview of the model differencing tool chain and related meta-tools

to manually intervene. The solution is based on the principle of document patching and covers normal as well as more advanced patching scenarios.

A specific variant of this concept provides a completely new approach to update workspace copies of models. It differs methodologically and technically from previous approaches based on 3-way merging. From the point of view of a tool user, the GUI is less complex than GUIs of interactive 3-way merge tools, and the user interaction concept does not force a user to mentally return to the common base version. From the point of view of a tool chain configurator, an interactive workspace update tool can be integrated into an existing MDE tool environment with minimal effort.

Contribution 5: A generic model versioning framework. Finally, all of the proposed concepts are integrated in a versatile framework for implementing a family of model difference tools that cover a broad range of use cases, modeling languages, user requirements and application contexts. The design of this framework is driven by basic principles known from the field of software product line engineering [202]. Its scope is formally documented in a variability model which is the result of a detailed analysis of our problem domain. In the solution space, we identify a set of reusable components and describe how they can be composed to create individual tools and tool functions. Domain features are mapped to components or configuration artifacts being used to adapt a particular component. Difference tool development (or tailoring) thus becomes a semi-automated and guided engineering process.

A reference implementation of the proposed framework on top of the widely used Eclipse Modeling Project (EMP) demonstrates the technical feasibility of the approach. The implementation is known as the *SiLift model versioning framework* [247] and has been developed in a sub-project of the larger and substantially older *SiDiff project* [246]. During the last years, a considerable amount of individual difference tools and tool components based on SiLift has been developed; we use several of them to evaluate key concepts of our approach in different case studies and experiments.

1.5 Thesis-related List of Publications

The following papers related to this thesis have been published in publication outlets with scientific quality assurance (listed in chronological order):

[1] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. "A rule-based approach to the semantic lifting of model differences in the context of model versioning." In: *26th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lawrence, KS, USA*. IEEE, 2011, pp. 163–172.

[2] Timo Kehrer, Udo Kelter, Pit Pietsch, and Maik Schmidt. "Adaptability of model comparison tools." In: *IEEE/ACM International Conference on Automated Software Engineering (ASE), Essen, Germany.* ACM, 2012, pp. 306–309.

[3] Timo Kehrer, Udo Kelter, Manuel Ohrndorf, and Tim Sollbach. "Understanding model evolution through semantically lifting model differences with SiLift." In: *28th IEEE International Conference on Software Maintenance (ICSM), Trento, Italy.* IEEE Computer Society, 2012, pp. 638–641.

[4] Udo Kelter, Timo Kehrer, and Dennis Koch. "Patchen von Modellen." In: *Software Engineering 2013: Fachtagung des GI-Fachbereichs Softwaretechnik, Aachen, Germany.* Vol. 213. LNI. GI, 2013, pp. 171–184.

[5] Timo Kehrer, Michaela Rindt, Pit Pietsch, and Udo Kelter. "Generating Edit Operations for Profiled UML Models." In: *Proceedings of the Workshop on Models and Evolution (ME) co-located with ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Miami, FL, USA.* Vol. 1090. CEUR Workshop Proceedings. 2013, pp. 30–39.

[6] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. "Consistency-preserving edit scripts in model versioning." In: *28th IEEE/ACM International Conference on Automated Software (ASE), Silicon Valley, CA, USA*. IEEE, 2013, pp. 191–201.

[7] Timo Kehrer. "Generierung konsistenzerhaltender Editierskripte im Kontext der Modellversionierung." In: *Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, Kiel, Germany.* Vol. 227. LNI. GI, 2014, pp. 57–58.

[8] Timo Kehrer, Udo Kelter, and Dennis Reuling. "Workspace updates of visual models." In: *ACM/IEEE International Conference on Automated Software Engineering (ASE), Vasteras, Sweden*. ACM, 2014, pp. 827–830.

[9] Timo Kehrer, Udo Kelter, and Gabriele Taentzer. "Propagation of Software Model Changes in the Context of Industrial Plant Automation." In: *Automatisierungstechnik* 62.11 (2014), pp. 803–814.

[10] Michaela Rindt, Timo Kehrer, and Udo Kelter. "Automatic Generation of Consistency-Preserving Edit Operations for MDE Tools." In: *Proceedings of the Demonstrations Track of the ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Valencia, Spain.* Vol. 1255. CEUR Workshop Proceedings. 2014.

[11] Timo Kehrer, Christopher Pietsch, Udo Kelter, Daniel Strüber, and Steffen Vaupel. "An Adaptable Tool Environment for High-level Differencing of Textual Models." In: *Proceedings of the Workshop on Models and Evolution (ME) co-located with ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Ottawa, Canada.* accepted publication. 2015.

A subset of the key ideas and concepts developed in this thesis are presented in [1, 3, 4, 6, 7, 8, 9]. Note that [2, 5, 10] report on techniques that have been collaboratively developed in the SiDiff project; several of these techniques are tightly integrated into the model versioning framework developed in the context of this thesis.

Moreover, the author presented the SiLift framework and tool suite at the following conference events:

[12] Timo Kehrer. *Henshin and SiDiff: Specifying and Recognizing Model Changes Based on Edit Operations.* Tutorial at SPP1593 Workshop on Managed Software Evolution (WMSE) co-located with Software Engineering 2013, Aachen, Germany. 2013.

[13] Thorsten Arendt, Timo Kehrer, and Gabriele Taentzer. *Understanding Complex Changes and Improving the Quality of UML and Domain-Specific Models.* Tutorial at the ACM/IEEE 16th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Miami, FL, USA. 2013.

In addition to our own experiments which have been conducted to evaluate the contributions of this thesis, we showcased the flexibility of our approach and the benefits of semantically lifted differences in several collaborative research activities. These cooperations led to peer-reviewed publications [14, 15, 16, 17, 18, 19], the article [20] has been submitted for publication.

[14] Hamed Shariat Yazdi, Pit Pietsch, Timo Kehrer, and Udo Kelter. "Statistical Analysis of Changes for Synthesizing Realistic Test Models." In: *Software Engineering 2013: Fachtagung des GI-Fachbereichs Softwaretechnik, Aachen.* Vol. 213. LNI. GI, 2013, pp. 225–238.

[15] Timo Kehrer, Pit Pietsch, Hamed Shariat Yazdi, and Udo Kelter. "Detection of High-Level Changes in Evolving Java Software." In: *Softwaretechnik-Trends* 33.2 (2013).

[16] Hamed Shariat Yazdi, Pit Pietsch, Timo Kehrer, and Udo Kelter. "Synthesizing Realistic Test Models." In: *Computer Science–Research and Development (CSRD)* (2014), pp. 1–23.

[17] Hamed Shariat Yazdi, Mahnaz Mirbolouki, Pit Pietsch, Timo Kehrer, and Udo Kelter. "Analysis and Prediction of Design Model Evolution Using Time Series." In: *Advanced Information Systems Engineering Workshops - CAiSE 2014 International Workshops, Thessaloniki, Greece.* Vol. 178. Lecture Notes in Business Information Processing. Springer, 2014, pp. 1–15.

[18] Sinem Getir, Michaela Rindt, and Timo Kehrer. "A Generic Framework for Analyzing Model Co-Evolution." In: *Proceedings of the Workshop on Models and Evolution (ME) co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Valencia, Spain.* 2014, pp. 12–21.

[19] Birgit Vogel-Heuser, Jens Folmer, Matthias Kowal, Ina Schaefer, Sascha Lity, Alexander Fay, Winfried Lamersdorf, Timo Kehrer, Matthias Tichy, and Bernhard Beckert. “Selected Challenges of Software Evolution for Automated Production Systems.” In: *Proceedings of the 13th IEEE International Conference on Industrial Informatics (INDIN), Cambridge, UK.* accepted publication. 2015.

[20] Johannes Bürdek, Timo Kehrer, Malte Lochau, Dennis Reuling, Udo Kelter, and Andy Schürr. “Reasoning about Product-Line Evolution using Complex Differences on Feature Models.” In: *Automated Software Engineering* (2015). submitted for publication.

1.6 Thesis Outline

The thesis is structured to step-wise motivate, elaborate and finally evaluate the proposed approach. A brief overview of each chapter is given in the remainder of this section.

Chapter 2: Model Version and Variant Management. In this chapter, we introduce a set of selected model versioning scenarios. Some of the scenarios will be illustrated by sample models serving as running examples throughout the thesis. Subsequently, we apply a feature-oriented analysis to the domain of model version and variant management. A feature diagram formally documents the tool functions being required in our versioning scenarios and identifies major variation points. Next, we give an overview of the state of the art on how to implement these tool functions. The chapter concludes with a discussion of major challenges and open problems. Parts of this chapter have been published in [2, 4, 8, 9].

Chapter 3: Representation and Editing of Models. In this chapter, we introduce our conceptual notion of an abstract syntax graph (ASG) which is formally considered as typed, attributed, directed graph. A meta-model defines the allowed types of nodes and edges of an ASG and is formally treated as a distinguished graph called type graph. Based on these fundamental definitions for representing models as graphs, we introduce the notion of model consistency being relevant in the context of our problem domain. Finally, we present our approach to formally specify edit operations on models using the graph rewriting and model transformation language Henshin. We briefly introduce the underlying foundations, including static analysis techniques, being required in terms of this thesis.

Chapter 4: Semantic Lifting of Model Differences. In this chapter, we present our approach to semantically lifting low-level model differences being obtained from currently available model differencing tools to high-level model differences which are based on user-level edit operations. Therefore, we first introduce our representation of low-level differences serving as basis for the definition of semantic change sets. Next, we present our approach to specify instances of semantic change sets by so-called recognition rules. A recognition rule is basically a specification of a certain change pattern

which has to be found on a low-level difference representation; it can be automatically derived from its corresponding edit rule which serves as formal specification of a user-level edit operation. Thereafter, we present our overall strategy for the application of recognition rules. The chapter closes with a discussion of the general restrictions of our approach to operation recognition. The chapter is a revised and substantially extended version of [1].

Chapter 5: Generation of Edit Scripts. Semantically lifting a low-level difference is helpful for understanding model changes, but not sufficient for replaying them as an executable difference in change propagation scenarios. In this chapter, we define an extended kind of directed delta to which we refer to as edit script. A semantically lifted difference must be further processed to become an executable edit script. The additional detection of actual parameters and dependencies between edit steps will be explained in this chapter. The chapter is a revised version of [6].

Chapter 6: Propagation of Changes based on Consistency-preserving Edit Scripts. In this chapter, we present our approach to the controlled propagation of model changes which is based on the principle of document patching. Changes which are to be propagated are basically expressed as a consistency-preserving edit script obtained from the original and the revised version of a model. This edit script can be further processed to become a patch which is finally distributed and applied to a target model. Concerning the patch application, we define two basic variants of a patch operator; the variation point is whether the original model and the target model of a patching scenario are revisions of a common base model or not. We present a set of analysis procedures for the detection of different kinds of errors and conflicts. Finally, we present our approach to the interactive application of patches which enables tool users to handle the detected errors and conflicts. Parts of this chapter have been published in [4, 8, 9].

Chapter 7: Creation of a Set of Consistency-preserving Edit Rules. Our approach to model differencing and change propagation is adapted to a given modeling language by providing edit rules for this language, each edit rule is a formal specification of an edit operation available for this language. In general, this set of edit rules must be properly designed and meet certain correctness criteria. In this chapter, we clarify these criteria and identify requirements for a set of mandatory edit rules; these rules must be provided in order to guarantee that correct edit scripts are generated. While complete sets of mandatory edit rules can be easily defined based on primitive graph operations, engineering a mandatory set of consistency-preserving edit rules (CPERs) is much more challenging. Therefore, this chapter introduces a semi-automated approach to create a complete set of CPERs for a given modeling language. Early ideas have been presented in [5, 10], the fundamentals presented in this chapter provide a significant improvement over this previous work.

Chapter 8: A Generic Model Versioning Framework. In this chapter, we present SiLift, a generic framework for building highly customized differencing and patching tools for models which are based on the concepts developed in this thesis. According to basic principles of Software Product Line Engineering (SPLE), we choose a compositional approach in order to represent the required variability in the technical solution space. Most of the user-visible features are mapped to distinct implementation artifacts which can be flexibly composed to form a particular tool configuration. Thereby, solution space variability in SiLift is primarily represented on the architectural level, i.e. we identify a set of software components as re-usable implementation artifacts, some of these components can be adapted by additional configuration data. We finally give a brief overview of our prototypical implementation of the framework based on the widely used Eclipse Modeling Project (EMP). Parts of this chapter have been published in [3, 8].

Chapter 9: Evaluation. In this chapter, we evaluate our approach from two different perspectives: From the point of view of a tool chain configurator, the evaluation should show the suitability of the proposed framework. From the point of view of a model versioning tool user, we are mainly interested in certain quality aspects of different tool functions. Concerning the first perspective, we outline the configuration effort to implement of a set of example applications that have been used in different case studies, most of them have been conducted in collaborative works presented in [11, 13, 14, 15, 16, 17, 18, 19, 20]. W.r.t. the second perspective, we conducted several experiments assessing selected quality aspects for a set of representative tool configurations and data sets. A subset of the experimental results has been published in [6, 8].

Chapter 10: Related Work. In this chapter, other work which is closely related to ours will be investigated in detail. A special emphasis is put on generic approaches which, similar to our approach, can be adapted to a broad range of different modeling languages. Concerning language-specific approaches which cannot be transferred to other modeling (sub-) languages, we review those approaches being related the contributions of this thesis. In this regard, we identified two directions; the detection of complex edit operations and semantic differencing. Furthermore, we give an overview of available model repositories. The chapter closes with a review of approaches from other domains being closely related to a subset of the problems addressed in this thesis.

Chapter 11: Conclusions and Future Work. This chapter concludes the thesis with a summary and an outlook on possible future work and research directions.

CHAPTER 2

Model Version and Variant Management

A number of development tasks are involved in the management of versions and variants of models, a brief overview is given in Section 2.1. While these development tasks can be basically supported by a set of similar tools, each task comes with its specific requirements. Section 2.2 presents the results of analyzing the commonalities and variabilities of the required tool functionality. The state of the art on how to implement the most basic tool functions is summarized in Section 2.3. We conclude with a discussion of major challenges and open problems in Section 2.4.

2.1 Motivating Scenarios

Development tasks being involved in the version and variant management of models are basically the same as for the versioning of traditional software development documents. The most important scenarios in which these tasks typically occur are briefly outlined in the remainder of this section. Some of the scenarios will be illustrated by sample models serving as running examples throughout the thesis.

2.1.1 SC1: Understanding Model Changes

The evolution of a model-based system can only be controlled effectively if changes between versions of a model are well-understood. Precise and meaningful descriptions of model changes serve as an indispensable basis for many tasks being directly or indirectly associated with version and variant management, e.g. for tracing changes in

models to change requests or bug fixes, documenting design decisions, analyzing the history of a model, or to simply review recent changes in a model.

To that end, users compare models because they basically wish to know which model elements were added, removed, or modified. They are further interested in complex restructurings and the rationale of each of the applied changes. The result of a comparison, i.e. a difference, should thus deliver a specification of how to convert the first model into the second one in a step-wise manner. Each edit step should describe a model modification that is meaningful from a modeler's point of view. If we compare the model versions v_1 and v_2 of the following Examples 2.1 and 2.2, a description of the observed difference between these revisions should be similar to the editing sequences listed in the examples.

Example 2.1 (Sample modification of a UML class diagram)
Figure 2.1 shows two revisions of a simple UML class diagram. Base version v_1 has been edited to become the revised version v_2 as follows:

1. The navigability of association *worksFor* has been restricted to one end, indicated by adding an arrowhead[1].

2. A generalization relationship has been created such that class *Developer* specializes class *Person*.

3. Similar to step 2, another generalization relationship has been created such that class *Manager* specializes class *Person*.

4. Finally, the model has been refactored using the well-known refactoring operation pullUpAttribute [107]. Here, the attribute *name* has been pulled up along the generalization relationships created in steps 2 and 3.

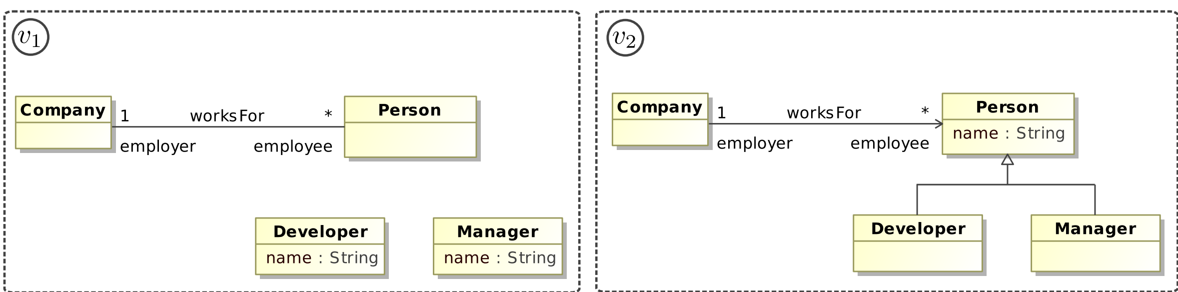


Figure 2.1: Sample modification of a UML class diagram: Original version v_1 and its revision v_2

[1]We assume the usual UML presentation option here that an association without arrowheads is navigable in both directions.

Example 2.2 (Sample refactoring of a UML state chart)
Figure 2.2 shows two revisions of a UML state chart modeling a simple telephone object as described in [190]. Note that both revisions specify the same behavior, while the readability has been significantly improved in version v_2; the states modeling the behavior of the phone when it is in use are grouped into a composite state called *Active*. Thus, transitions *hangup* which have to be specified for several simple states in version v_1 are replaced by a single transition in version v_2. This modification can be achieved in one user-level editing step to which we refer to as extractCompositeState, inspired by the refactoring catalogue presented in [231].

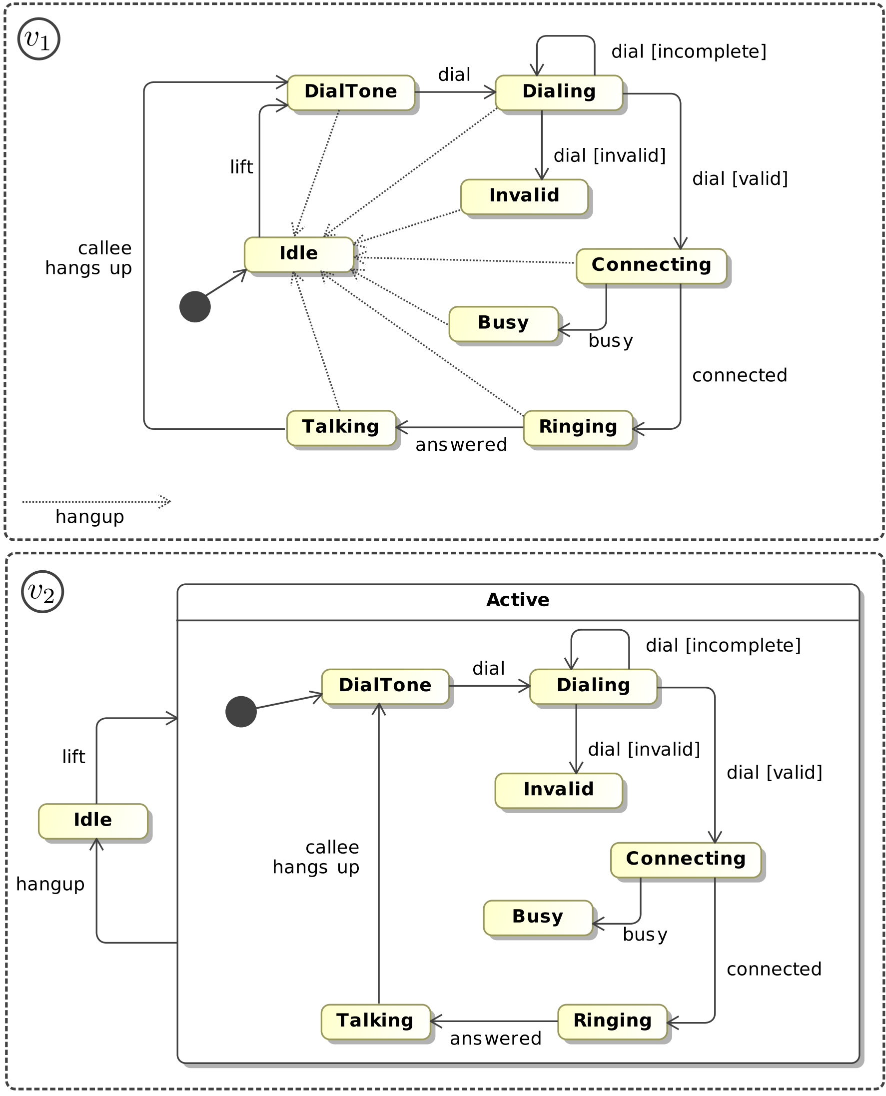


Figure 2.2: Sample refactoring of a UML state chart taken from [231]

2.1.2 SC2: Updating a Copy of an Original Model

Several copies of a model often reside at different locations. In the case of executable models, for instance, these copies reside in several environments or hosts, where the same application is installed. If a new release is rolled-out, all copies must be updated. One option is to distribute the complete new model. Another solution is to only deliver a change description that has to be applied in order to update a copy of an original model. Such a change description is commonly referred to as *patch*, the document to which the patch shall be applied is called *target document* in this case. The main advantage of propagating only the changes instead of delivering a complete model is the reduction of the amount of data to be distributed because patches can be more compact than entire models.

A similar scenario is the delta storage of a revision history in a repository. Instead of separately storing each new revision of a document, only the changes w.r.t. the preceding revision are stored in the repository. If required, a particular revision can be restored by applying a sequence of patches. Delta storage typically reduces space consumption to a small percentage of the full storage needs [128].

The most important characteristic of this scenario is that the patch is always applied to a *copy of the original version*.

2.1.3 SC3: Cherrypicking Changes

While several version models have been proposed in the literature, the mainstream of versioning systems is based on organizing versions in a directed acyclic graph which is commonly known as *version graph* [77]. "Cherrypicking" [76] assumes an organization in which a version graph is composed of parallel branches, where each of these branches consists of a sequence of revisions, basically as shown in Figure 2.3, where symbolic version identifiers are chosen according to the numbering scheme in CVS [105]. Special relationships between versions 1.2 and 1.2.2.1 as well as 1.4 and 1.4.2.1 identify the origin of a new branch. These relationships are typically referred to as offspring relationships, while the relationships within a branch are called successor relationships [77]. The sample revision graph in Figure 2.3 consists of three parallel branches; the main development branch, which is usually called trunk, a feature branch, in which a dedicated, yet instable, feature is being developed, and a release branch. In our feature branch, a bug has been fixed from revision 1.2.2.5 to revision 1.2.2.6. Most likely, this bug also exists in the trunk and the release branch. Thus, the respective bugfix has to be replicated in the latest versions of these branches.

This use case of replicating (or backporting) bug fixes from one branch to one or several others is a frequent reason for cherrypicking changes. [76] presents several other situations in which certain changes have to be propagated from one branch to another. Note that propagating changes from one branch to another does not mean that these branches are joined (or merged), each of them continues to exist.

In general, we use the term "cherrypicking" whenever changes which occurred between two revisions of one development branch shall be applied to a target model in

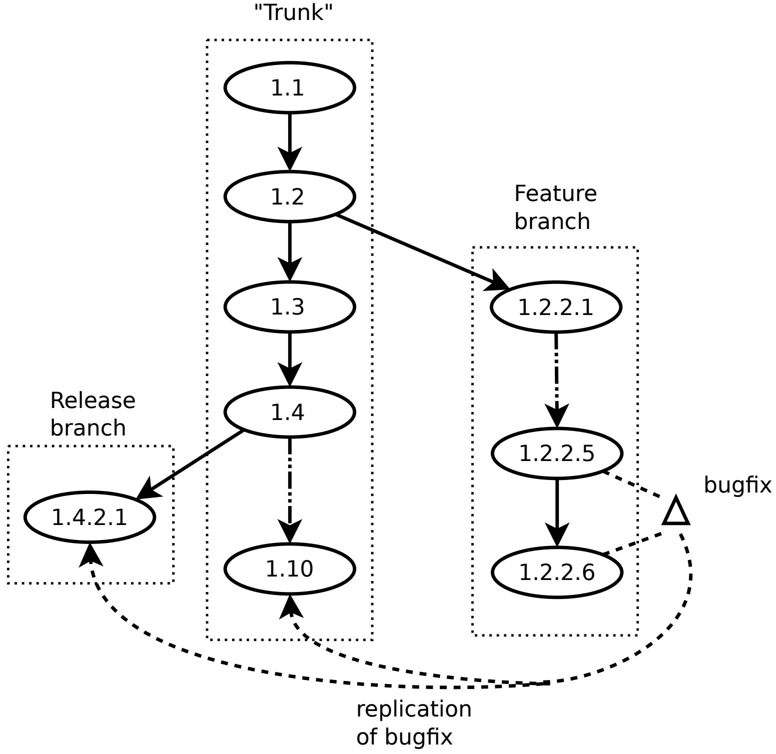


Figure 2.3: Example: Basic principle of "cherrypicking changes"

another branch. The original model and the target model are usually not identical, but they are managed in a common repository and can be traced back to a common base version.

2.1.4 SC4: Propagation of Changes in a Product Family

Development tasks of this type occurred in two of our projects. The first one deals with mechatronic components in the automotive domain whose embedded software is generated from large models developed in ASCET [100] or Matlab/Simulink [176]. Each component has many variants, typically between 10 and 50, one for each type of vehicle in which the component is integrated. The second project deals with model-based software development for plant automation using various diagram types of the SysML [191]. There are many copies of the plant, each with a local copy of the models. Local adjustments during operation and maintenance lead to local changes.

Both projects have several commonalities. Each variant develops rather autonomously and is assigned to a different responsible developer. This autonomy is the main reason why methods known from product line engineering, which develop and maintain all variants centrally, are not accepted although the variants are quite similar. Nonetheless, an improvement in one variant, e.g. a bug-fix or a new feature, usually has to be propagated to some or all other variants [103]. Moreover, local changes in one variant are often *unanticipated changes* [66]; they cannot be foreseen during the initial develop-

ment of a system and, as such, cannot be accommodated in the design, which is usually a basic prerequisite for successfully establishing an SPL.

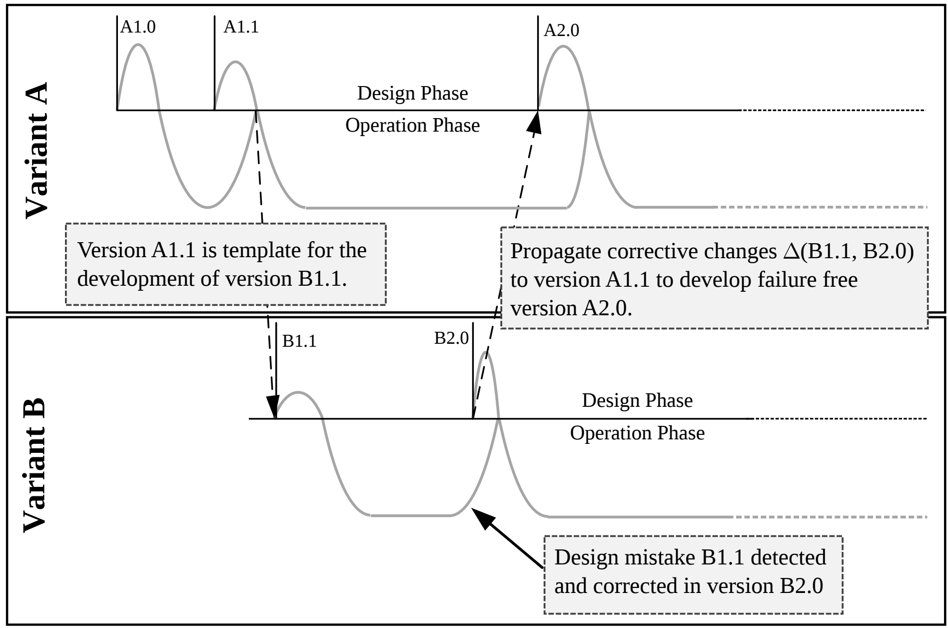


Figure 2.4: Parallel evolution of variants according to [253]

A typical example from the automation engineering domain is shown in Figure 2.4. The example is based on [253] and shows a parallel evolution of two variants of a model of a software module. Initially, the module has only one variant, called A. Its revision A1.1 serves as template for the development of variant B which henceforth evolves in parallel to variant A. At some point in time during the operation of variant B, a design mistake is detected and corrected. This mistake also occurs in variant A and must, of course, also be corrected in variant A. Thus, the corrective changes applied to variant B must be propagated to variant A.

Superficially, the propagation of such improvements is similar to the cherrypicking scenario: An improvement is represented as a patch which is created by comparing revisions in the original variant and which is to be applied to one or several target models. However, several conditions are different from cherrypicking.

- The variants are often managed in different repositories. Thus, in contrast to the cherrypicking scenario, the original and the target model do not have a common base version.

- The person responsible for a target variant, i.e. the patch applier, must be able to check whether the changes are appropriate and permitted in this variant and, if necessary, to adapt the patch. In a cherrypicking scenario, both involved branches are typically maintained by the same person and the developer which applies the patch is assumed to be familiar with both branches. In the maintenance of large

product families, the roles of patch creator and patch applier are typically assigned to different persons.

- The patch should not contain changes which occurred "accidentally" between check-out and check-in in the workspace where the improvement was implemented, i.e changes which are not related to the intended improvement. In this scenario, the patch applier is assumed to be incapable to identify superfluous parts in the patch.

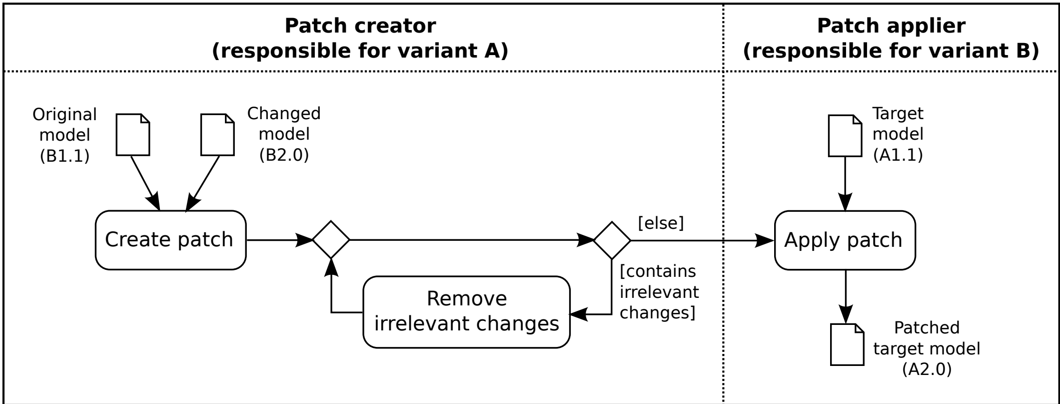


Figure 2.5: Propagation of changes in variants of a product family

We conclude that the patch creator must ensure that the patch does not contain changes which occurred at the same time as the improvement, but are not related to the intended improvement. Such irrelevant changes must be removed from the patch by the patch creator. The patch applier is responsible for a target variant and must be able to check whether the changes are appropriate and permitted in this variant. If necessary, the patch applier must adapt the patch, the target model, or both. A basic work flow of engineering tasks related to change propagation in product family variants is shown in Figure 2.5. In this work flow, the step *Apply patch* includes the adaptation of the patch and the target model.

Example 2.3 (Propagation of changes in variants of a UML class diagram)
The left side of Figure 2.6 shows two versions of a class diagram of a sample flight ticketing system. They are revisions of a variant A of the system. The original version on top of Figure 2.6 has been modified to become the changed version on the bottom as follows:

1. Class *Schedule* has been extracted from class *Flight* following the well-known refactoring extractClass [107].

2. The data type of the attribute *price* has been changed from *int* to *float*.

3. A new attribute *birthday* has been added to class *Passenger*.

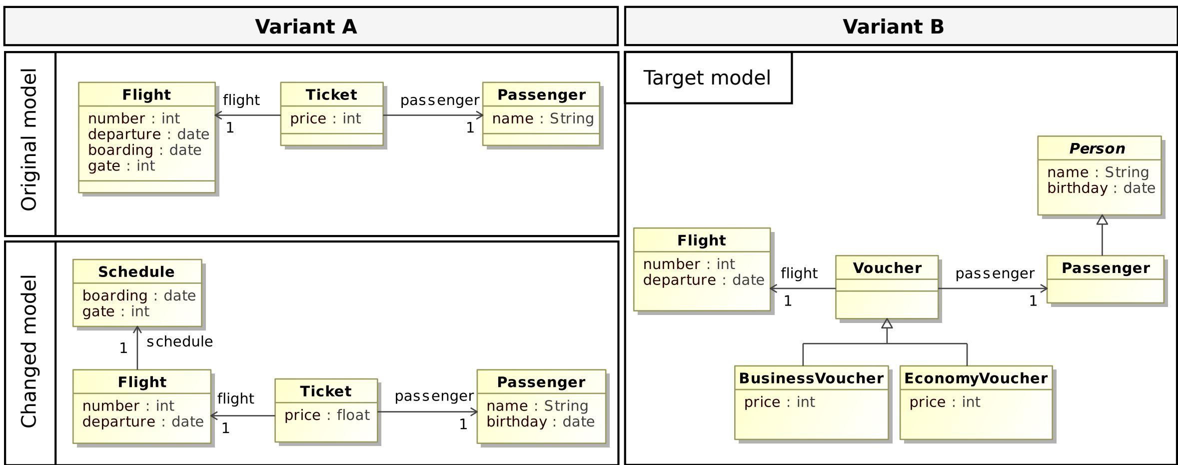


Figure 2.6: Original, changed and target model of a patching example

In our example, we assume another variant B of our system, the model shown on the right side of Figure 2.6 has the role of the target model. The task to be carried out is to repeat the changes between the original and changed model in the target model. The details and intricacies of this task are analysed in Section 2.2. Please note that even in this small example several independent problems have been resolved between the original and the changed version of variant A. The first change represents a typical structural refactoring, while the second change fixes a bug and the third change introduces a new feature.

2.1.5 SC5: Workspace Updates of Models

Large models are typically developed in teams. A *pessimistic* approach to support collaborative work is to avoid parallel modifications of the same model following the *lock-modify-unlock* paradigm [76] In practice, however, locking often leads to administrative problems [28]. Moreover, pessimistic versioning leads to a phenomenon known as “sequential parallelism”, i.e. semantic interferences between changes performed by different developers cannot be avoided and are likely to be unnoticed [238].

Thus, *optimistic versioning* is the common approach for supporting collaborative work in development projects. A central feature of optimistic versioning is a service which *updates* modified copies of documents in a workspace. An update propagates parallel changes which were made by other developers and which were checked-in into the repository. If several developers modify copies of a model then this can lead to incompatible changes. Such situations are commonly referred to as *conflicts* [180].

Example 2.4 (Concurrent modification of a model)
Figure 2.7 shows an example of a workspace update scenario which is based on an example presented in [60]. The base version v_b has been checked out into two workspaces and modified to versions v_1 and v_2, respectively. From v_b to v_1, the operation *Ticket.getInfo*

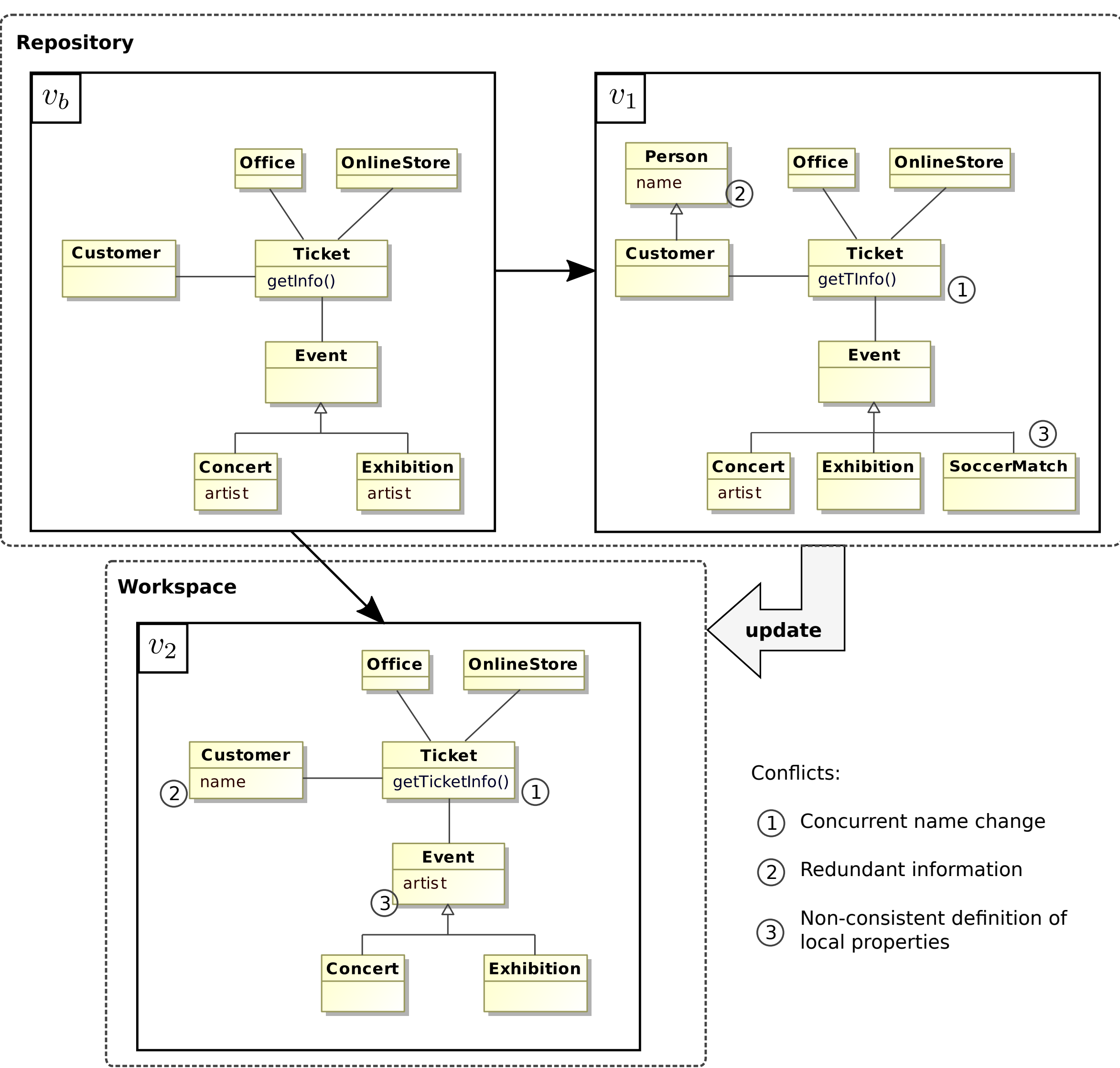


Figure 2.7: Example scenario of a concurrent modification of a model based on [60]

has been renamed, a new class *Person* and an attribute *Person.name* have been introduced, and the attribute *Exhibition.artist* has been deleted. From v_b to v_2, the operation *Ticket.getInfo* has been renamed, a new attribute *Customer.name* has been introduced, and the redundant attribute *artist* has been pulled up to the common superclass *Event*. Version v_1 was then checked-in into the repository. The changes between v_b and v_1 shall be propagated to v_2. In this example, three conflicts arise:

① The operation *Ticket.getInfo* was renamed concurrently.

② The attribute *name* was introduced twice.

③ The attribute *artist* within the inheritance hierarchy of the class *Event* is not defined uniformly.

Many conflicts can be resolved by choosing one of the alternatives. For example, in conflict ①, one can adopt one of the renamings and cancel the other one. Some conflicts cannot be resolved in this way, but only by restarting development work related to the part of the model which is affected by the conflict. In case of conflict ③, for instance, one has to rethink what the attribute *artist* actually means. This can even lead to the decision that both conflicting changes are cancelled and a completely new solution is devised creatively, i.e. one does not follow a standardized pattern at all.

2.2 A Feature-oriented Domain Analysis

Model-driven engineering strongly depends on sophisticated tool support, which applies in particular to the domain of model version and variant management. Difference tools for models are, arguably, a product family, i.e. they share a set of common features, while each application scenario comes with a set of individual requirements.

In this section, we present the results of applying a feature-oriented analysis to the domain of model version and variant management. Special emphasis is put on the scenarios presented in Section 2.1. The results are formally documented in the feature diagram shown in Figure 2.8. A FODA-like feature diagram [133] as shown in Figure 2.8 is basically a tree with additional cross-tree constraints. It represents the set of domain features as nodes and contains further notational constructs to denote different kinds of logical constraints, namely feature dependencies and mutual exclusions, among those features.

The features of Figure 2.8 define common and variable parts of the individual members of a family of model difference tools. On the one hand, these features denote a set of “user-visible” tool functions. We focus here on the calculation and propagation of model changes. These functions and their distinguishing characteristics are considered in sections 2.2.2 to 2.2.6. Note that we deliberately leave open whether these functions are realized as standalone tools or whether they are integrated in other tools such as a version control system. On the other hand, the modeling language, notably the way how instances of a model type are edited in a particular MDE development environment, is another important variation point for the construction of difference tools for models. Thus, the feature *Modeling Language* and its (indirect) sub-features do not directly refer to a certain tool function. They rather represent domain abstractions and important characteristics of an MDE environment having a strong impact on the design of difference tools for models of this particular type (s. Section 2.2.1).

Please note that complementary features of a version control system, such as version space organization, repository architecture, identification of configuration items, baselining and tagging etc. are out of the scope of our domain analysis. Thorough analyses of these repository characteristics can be found in [28, 65].

2.2.1 Modeling Languages and MDE Development Environments

In contrast to many other sciences and engineering disciplines, where models are often some kind of physical objects, models in software engineering are conceptual models being intangible artifacts. To allow developers to work with models, a modeling language is required in which they can formally write down a model. A modeling language is an artificially constructed language which is used to precisely describe the instances of a specific type of model. The concrete syntax of a modeling language defines a set of symbols, while the abstract syntax defines how these symbols may be combined to formulate semantically correct models. In some cases, a modeling language has a formal semantics. Frequently, however, there is no formal semantics and the meaning of a model is mainly defined by transformations to platform-specific models or code.

Definition of modeling languages. Textual languages are typically specified by a context-free grammar. The abstract syntax of visual modeling languages is usually defined by a meta-model, which basically defines the allowed types of nodes and edges of an ASG. Widely adopted standards, such as the *UML Infrastructure Specification* [189] defined by the OMG, provide an object-oriented approach to meta-modeling, i.e. meta-models are defined using data modeling techniques known from object-oriented analysis and design. The concrete set of available data modeling concepts is defined by a so-called meta-metamodel. One widely adopted standard is OMG's *Meta-Object Facility* (MOF) [193] specification with its compliance levels *Complete MOF* (CMOF) and *Essential MOF* (EMOF).

Note that grammars define both abstract and concrete syntax of a textual language, while the concrete syntax of visual languages must be defined by an additional relationship that specifies how to map ASG elements onto graphical symbols. From a tool construction point of view, however, meta-models are very attractive as they provide data models for the representation of models of a particular model type. In fact, grammars can be converted to meta-models (and vice versa), basic mapping procedures can be found in [23]. Several optimizations have been proposed in order to get more "user-friendly" [262] or "meaningful" [45] meta-models.

Representations of a model. From a tooling point of view, we consider a model as an editable document which typically exists in several representations that can be roughly categorized into physical, internal and external representations.

MDE tools are basically faced with one or several models which are represented as contents of persistent storage media (physical representation) or as runtime objects (internal representation). Persistent storage media include XML files, proprietary file formats, and relational databases. In order to be processed by tools, persistent representations of models must be "loaded", i.e. they must be converted into an internal representation consisting of runtime objects. The internal representation can have the same structure as the persistent representation, e.g. some textual format, but often it is

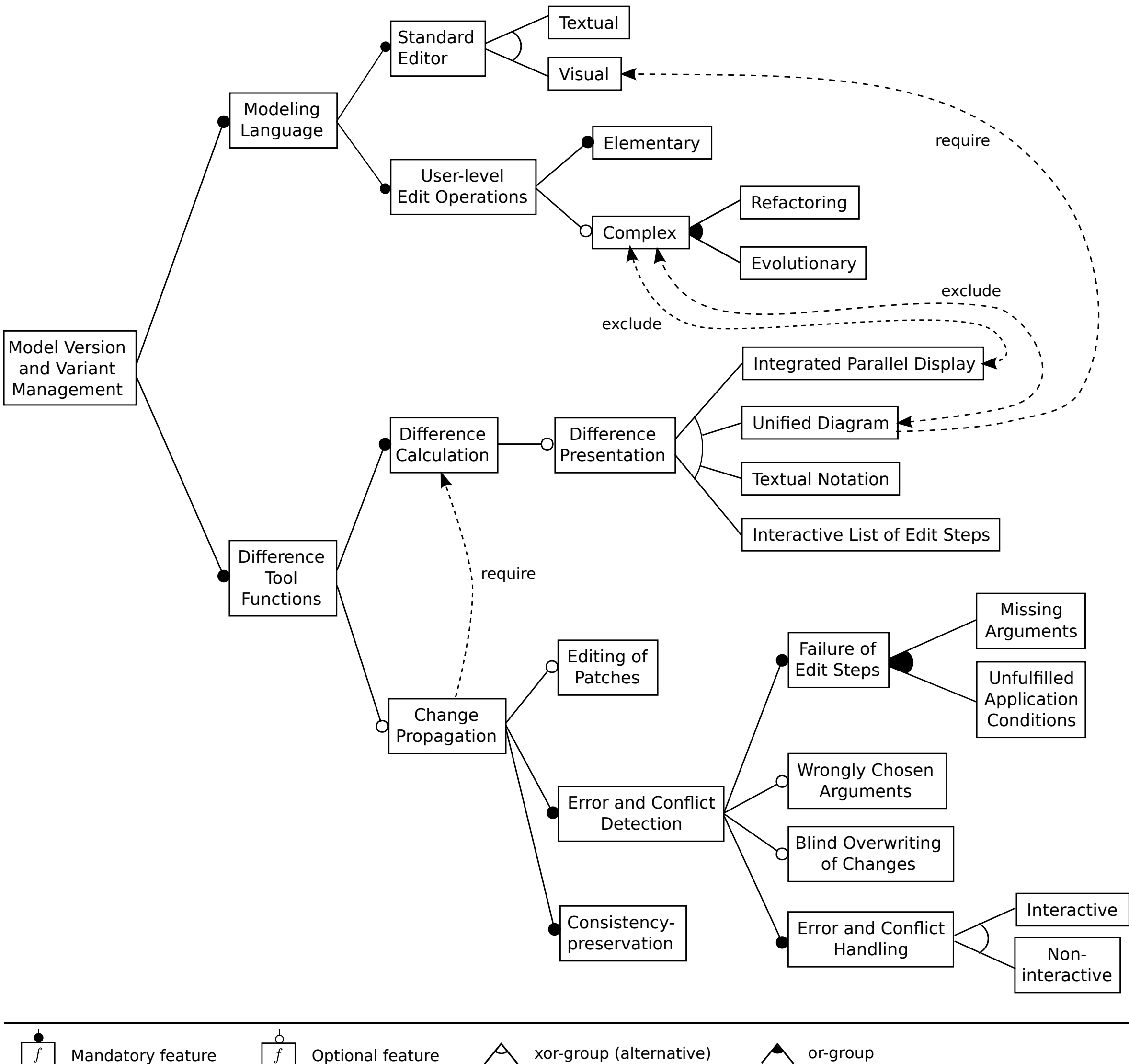


Figure 2.8: Results of a feature-oriented domain analysis

converted into a tree or graph structure which can be considered as an implementation of an ASG.

Developers are faced with the external representation of a model in some modeling tool, notably editors in which models are interactively created and modified. A model can be processed in different editors, where each editor comes with its own external representation. We distinguish between textual and visual editors. Usually, an MDE development environment provides a dedicated editor being used to edit models of a particular type. We refer to this editor as *standard editor*. As we will see in the remainder of this section, the standard editor plays an important role in the design of difference tools for models since it determines the level of consistency being required to externally display models.

Consistency of models. Many MDE tools are developed for single tasks like, e.g., transforming, editing, or refactoring of models. Thus, models are frequently exchanged in a series of tools. All of these tools of an MDE tool chain have different requirements on the consistency of the models to be processed. Code generators or interpreters, for instance, are very restrictive as they require "perfectly" consistent models having well-defined semantics. On the contrary, models are often not perfectly consistent during editing processes. The level of consistency being relevant in our context is defined by the standard editors of a development environment; as long as they can load a model, one can at least manually improve the consistency of a model. The minimal level of consistency always includes conditions required to produce external representations of models, which will be analyzed in more detail in the remainder of this section.

Perfect vs. effective consistency. Analogously to other kinds of textual development documents, textual models can be loaded and edited in common text editors even if the ASG being represented by a document is seriously incorrect.

The situation is quite different for visual editors. Here, essential parts of an ASG are represented externally as graphical objects, which is only possible if the ASG meets certain basic consistency constraints. Dangling edges, for instance, usually have to be strictly avoided. Moreover, some ASG elements can only be externally represented if they form a coherent ASG fragment, which is usually expressed by multiplicity constraints in a meta-model. For example, associations in a UML class diagram cannot be visualized without their respective association ends.

Meta-models such as the UML Superstructure Specification typically define further advanced well-formedness rules. In the context of MOF-based meta-models, these rules are usually expressed using the Object Constraint Language (OCL) [195]. For example, final states in UML state machines must not have an outgoing transition (s. well-formedness rule (a) in Table 2.1). The degree of consistency achieved by these constraints can be called "perfect" because models meeting these constraints can be interpreted or correctly translated to source code.

However, typical visual editors do not enforce these advanced constraints. Firstly, the advanced well-formedness rules are not required for displaying a model visually, i.e. a model violating these well-formedness rules can nonetheless be displayed. Secondly, some of these constraints, e.g. the well-formedness rules (g)-(j) listed by Table 2.1, are hard to enforce after each editing command. Enforcing such constraints would make editing very clumsy since transitions between perfectly consistent states of a model can be quite complex.

We have analyzed two leading visual editors for UML models, Papyrus [93] and MagicDraw [187], concerning the set of advanced constraints enforced by them. Table 2.1 shows the results of this analysis for selected well-formedness rules of the UML Superstructure Specification [190]. It turns out that Papyrus does not seem to enforce any advanced well-formedness rule and that MagicDraw enforces only a subset of these rules. In fact, almost all advanced well-formedness rules which are actually enforced restrict the cardinality of certain element sets and could be also specified as multiplicity

	Well-formedness rule	**Papy-rus**	**Magic-Draw**
(a)	A final state cannot have any outgoing transitions.	✗	✓
(b)	A final state cannot have regions.	✗	✓
(c)	A final state has no entry behavior.	✗	✓
(d)	A final state has no exit behavior.	✗	✓
(e)	An initial vertex can have at most one outgoing transition.	✗	✓
(f)	History vertices can have at most one outgoing transition.	✗	✓
(g)	A join vertex must have at least two incoming transitions and exactly one outgoing transition.	✗	✗
(h)	All transitions incoming a join vertex must originate in different regions of an orthogonal state.	✗	✗
(i)	A fork vertex must have at least two outgoing transitions and exactly one incoming transition.	✗	✗
(j)	All transitions outgoing a fork vertex must target states in different regions of an orthogonal state.	✗	✗
(k)	The outgoing transition from an initial vertex may have no trigger or guard.	✗	✗
(l)	A region can have at most one initial vertex.	✗	✓
(m)	A region can have at most one deep history vertex.	✗	✓
(n)	A region can have at most one shallow history vertex.	✗	✓
(o)	Only submachine states can have connection point references.	-	✓
(p)	The connection point references used as destinations/sources of transitions associated with a submachine state must be defined as entry/exit points in the submachine state machine.	-	✗
(q)	All the members of a namespace are distinguishable within it.	✗	✓

✓: enforced ✗: not enforced - not supported

Table 2.1: Implementation of selected advanced well-formedness rules of the UML Superstructure Specification v. 2.4.1 in visual UML editors Papyrus (version 0.10.1) and MagicDraw (Personal Edition version 17.0.3)

constraints. The only counterexample revealed by our analysis is the well-formedness rule (q) which applies to UML state machines in the sense that each state machine must have a unique name within its defining package.

Another general finding is that comprehensive standards such as the UML Superstructure Specification are only partially implemented by available modeling editors [97]. Well-formedness rules which are related to non-implemented parts of a standard meta-model are not enforced by the editor environment as a matter of fact. For example, the well-formedness rules (o) and (p) listed by Table 2.1 are not enforced by Papyrus as the state machine diagram editor does not support *connection point references* at all.

We conclude that typical visual editors support only a reduced level of consistency, which we will refer to as *effective consistency level*, and that the effective consistency can be typically expressed by using multiplicity constraints (in addition to the basic consistency constraints).

2.2.2 User-level Edit Operations

Previous requirements analyses with industrial partners, e.g. as collected in a cooperation with ETAS GmbH [216], have virtually always led to the requirement that users should be able to configure the result of differencing tools if there are meaningful alternatives. The most important variation point is the set of edit operations which are available to modify models of a particular type. Edit operations occur in the descriptions of model differences, are offered as editing commands in editors etc. As usual, one should distinguish three aspects of an operation, namely *i)* the operation *signature* including the operation name and a list of formal parameters, *ii)* the operation *specification* which (formally or informally) specifies the effect of executing the operation and the conditions under which the operation can be applied, and *iii)* the *implementation* of the operation.

Operations can only be implemented if a (runtime-) representation of models is available. In this section, we will abstract from representation details and consider a model as an encapsulated data object which is accessible via an interface which defines a set of edit operations. All edit operations must be applicable to a model of the given model type in a meaningful way, i.e. their effect must be understandable by tool users. Understandability usually implies that an edit operation must transform a model from one consistent, editable state into another. This requirement obviously strongly depends on the properties of the standard editor. We refer to these kinds of edit operations as *user-level edit operations*. The set of user-level edit operations can be classified into elementary and complex edit operations.

Elementary edit operations. Elementary operations are the smallest edit operations from a user's point of view, i.e. they cannot be split into smaller operations being applicable to a model in a meaningful way. Edit operations restrictNavigability and createGeneralization of Example 2.1 are examples of this. The set of elementary edit operations roughly corresponds to the set of basic editing commands as offered by a standard modeling editor, but can also include operations which are not offered as editing commands. As shown in Figure 2.8, the availability of a complete set of elementary edit operations is mandatory such that every model modification can be expressed using edit operations available in this set.

Complex edit operations. Complex edit operations are typically composed of elementary edit operations. They are rarely offered as editing commands by standard editors but rather provided by complementary tools such as modern refactoring environments [35]. Moreover, [233] shows how GMF-generated editors can be extended by complex edit operations. In contrast to elementary edit operations, the availability of complex edit operations represents an optional feature for the design of model difference tools (s. Figure 2.8). In principle, one can construct arbitrarily many complex edit operations from elementary ones. However, most of them are not useful for our aim to make evolution of models better understandable. An important criterion for selecting a complex operation is that it is easier to understand than the single contained elemen-

tary edit operations. This criterion depends obviously a lot on the modelling language and in addition on design and usage rules and patterns, which may be project-specific. We can roughly distinguish two types of complex edit operations:

- **Refactorings:** Many complex edit operations are known as refactorings. Roughly speaking, refactorings are syntactic changes which do not change the meaning of the model. In cases, where a modeling language has a formal semantics, one can formally prove that a refactoring does not change the meaning of models. They are frequently applied in model quality management, which is an integral part of process models in many application domains. Edit operations pullUpAttribute and extractCompositeState are examples of this. Further suggestions for refactorings can typically be found in style guides and related publications, see e.g. [107, 231].
- **Evolutionary edit operations:** Edit operations which are not refactorings lead to “real” changes; they are called evolutionary. Many evolutionary edit operations facilitate frequently recurring editing tasks, e.g. moving a set of elements to a different subsystem or composite state, generating getters and setters for class attributes in an object-oriented design model etc. It is largely a matter of user preferences which evolutionary edit operations are considered useful as a means to better understanding model changes.

Further classification criteria for complex operations typically depend on the modeling language and particular use case. In the context of documenting the evolution of feature models, for instance, we have classified edit operations w.r.t. their effect on the set of valid feature combinations (s. Section 9.2.3), inspired by the edit categories proposed by Thüm et al. [239].

2.2.3 Difference Calculation and Presentation

A difference (or delta) can be informally defined as a description of how two models differ from each other. Given two models A and B of the same type, then a difference can be considered as a specification of how model A can be edited in a step-wise manner to become model B. Differences are calculated for various reasons, e.g., to explain the changes between two model versions (SC1), to describe the changes which are to be propagated from one variant to another (SC2-4), or to describe the repository changes which are to be propagated to a local workspace version (SC5). Thus, the calculation of a difference is one of the most basic tool functions in model version and variant management.

Difference presentation. In most of the scenarios presented in Section 2.1, in particular in SC1, SC4 and SC5, tool users are also interested in how a difference is being presented. A detailed analysis of this variation point can be found in [136], the available alternatives are summarized in Figure 2.8. A widely used presentation technique is to show both model versions side by side in an *integrated parallel display*. The *unified*

diagram combines two versions of a visual model in a single diagram, elements which are common in both model versions are shown only once. However, a model difference displayed in a unified diagram or in an integrated parallel display only shows which model elements are common or specific. Elementary edit steps can be implicitly derived from this information, while complex edit operations cannot be presented (cf. exclude relationships in Figure 2.8). A *textual notation* is more flexible in the sense that complex edit operations can be also presented. However, differences should be explained in terms of their external representation, i.e. visual models should be shown in their graphical syntax instead of some textual notation [25]. Thus, the *interactive list of edit steps* is an attractive presentation technique for both visual and textual models, especially when complex edit operations have to be presented. This interactive technique can be seen as a hybrid form of a textual notation and an integrated parallel display. A control window displays a list of edit steps. Selecting an edit step in the control window causes that arguments of this edit step to be highlighted in the respective external model representations which can be shown in a standard editor.

Executable differences. A requirement that applies to all scenarios in which changes shall be propagated (SC2-5) is that a calculated difference must be executable[2]. A difference is called executable if it comprises all necessary information such that the step-wise transformation from a model A to its revision B can be automated. In particular, an executable difference specifies a (partial) order in which the edit steps have to be executed. In our Example 2.1, generalization relationships between classes *Developer* and *Person* as well as *Manager* and *Person* have to be created before the common attribute *name* can be pulled up. Dependencies between edit steps are a phenomenon which does not occur in classical difference tools for texts and needs to be specifically addressed for visual models.

Tool users may be explicitly interested in dependencies between edit steps in scenarios SC3-5. Here, the application of an edit step can fail, dependent edit steps thus cannot be executed, too (cf. Section 2.2.4). Since the number of transitively dependent edit steps can be large, a user should be supported to find out which are root causes of the problem, i.e. which edit steps are required for (or depend on) some other edit step, and what is the reason for a dependency.

2.2.4 Change Propagation

There are a couple of versioning scenarios in which changes shall be propagated from one model variant to another. We informally refer to these changes as patch which is to be applied to a target model. As we have seen for scenario SC4, a patch must be sometimes edited by the patch creator before it is distributed to the patch applier, e.g. when unnecessary parts shall be removed from a patch. Ideally, this task is supported

[2]Please note that the requirement that calculated model differences must be executable in some cases is a *technical requirement* which is therefore not included in the diagram of user-visible features presented in Figure 2.8.

by a dedicated tool function (s. optional feature *Editing of Patches* in Figure 2.8).

When a patch is to be applied to a given target model, one has to resolve all references to model elements which occur as arguments of edit steps. References to model elements are data values which are usable for identifying model elements, e.g. path names, persistent identifiers or other data which "describe" elements within a model. Some realizations of references are reliable, i.e. if a model element can be located it is the right one, and if no related model element is found, then the referenced model element does not exist in the model. If only our simplest change propagation scenario SC2, the patching of a copy of the original model, needs to be supported, one can easily find reliable realizations of references. If everything is organized properly the resolution of references will never fail, and the patch can be applied without errors. Each edit step can be successfully executed on the intermediate state produced by the preceding edit steps. This observation does no longer hold if the patch is applied to another model different from the original version, e.g. a different variant (SC3-4) or a modified workspace copy (SC5). These change propagation scenarios potentially may lead to different kinds of errors and conflicts.

2.2.5 Errors and Conflicts

In this section, we analyze potential errors and conflicts which can occur when changes are to be propagated to a target model which differs from the original version from which the changes have been derived. Depending on the concrete application scenario, a patch application tool must offer different tool functions to properly detect these errors and conflicts (s. feature *Error and Conflict Detection* in Figure 2.8). We briefly sketch how each of these problems can be handled. Most often, a particular problem can be solved in an interactive or non-interactive way (s. alternatives *Interactive* and *Non-interactive* for the variation point *Error and Conflict Handling* in Figure 2.8).

We begin with errors, namely the failure of edit steps and wrongly chosen arguments, that can arise for our patching scenarios SC3 and SC4, respectively. These errors, in particular the failure of edit steps, can also occur as effects of conflicting modifications in the workspace update scenario (SC5). Here, another problem that can occur when repository changes are propagated to a local workspace version is that local modifications will be blindly overwritten.

Failure of Edit Steps

There are two main reasons why edit steps can fail; missing arguments and unfulfilled application conditions.

Missing arguments. Obviously, an edit step fails if a referenced element used as an argument cannot be resolved in the target model. In such a case, we have to distinguish whether reliable references or unreliable references on model elements are used:

1. If a reliable realization of references is used and if a search for a referenced model element fails, then the referenced model element does not exist at all in the target

model. Two reactions are possible in this situation: (a) The user intervenes and creates the missing model elements, and (b) the involved edit step and all dependent edit steps are not performed on the target model.

2. If an unreliable realization of references is used, then a search for a referenced model element can fail for two reasons; the element actually does not exist, or the element exists but the resolution procedure is unable to detect it. If the element actually does not exist, then the same reactions (a) and (b) as in case 1 are possible. If it exists, then the user can intervene and manually specify the correct model element.

In terms of our change propagation examples in Section 2.1, the problem of missing arguments occurs several times. For instance, the first change in the patching scenario of Example 2.3, the extraction of the class *Schedule* from class *Flight*, cannot be performed in the target model in the same way as in the original model because the attributes *boarding* and *gate* are missing. In the workspace update scenario, missing arguments typically occur if elements have been deleted in the workspace version. In our Example 2.4, operation deleteAttribute(Exhibition.artist) fails because in v_2 *Exhibition* does not contain an attribute *artist*.

In the examples above, missing arguments do not exist at all in the target model. The reference resolution will fail, regardless of a reliable or unreliable realization of references is used. Missing arguments caused by unreliable references obviously depend a lot on the implementation of a reference resolution procedure. In our patching Example 2.3, for instance, the resolution of class *Ticket* in variant B might fail if fully qualified names are used to identify model elements; the classes *Ticket* and *Voucher* obviously have different names, although they represent the same concepts in the original and target model of our example.

Unfulfilled application conditions. Unfulfilled application conditions of an edit operation used in an edit step are another reason why the execution of an edit step can fail. The third change of Example 2.3, the creation of the attribute *birthday* in class *Person*, is an example of this. This edit step can be successfully applied to the target model (assuming that *Person* has been identified as context for this change), at least if only syntactic consistency is required. However, this change leads to a violation of a constraint for design-level class models; the name of an attribute must be unique within the inheritance closure of its containing class. If this consistency constraint is explicitly checked by an application condition, the edit operation cannot be executed successfully.

In the workspace update scenario, this check can fail if local changes invalidate a precondition. In our Example 2.4, the edit step createAttribute(Person.name) fails because subclass *Customer* has already an attribute *name*.

Wrongly Chosen Arguments

So far, we have considered errors due to missing arguments, i.e. references to model elements which cannot be resolved in the target model. If unreliable references are used, we also have to consider the case of wrongly chosen arguments. If the search for a referenced model element succeeds, the found model element can actually be a wrong element; using this element as operation argument, the edit step would be performed at a wrong "position" in the target model. The only viable solution is to warn the user about potentially wrong arguments and to ask for a manual inspection.

For example, in case of our second change of Example 2.3, changing the type of the attribute *price*, it might be unclear where to perform it because this attribute occurs twice in the target model.

Blind Overwriting of Local Changes

In the workspace update scenario, the failure of edit steps is only one possible effect of conflicting modifications. For example, in conflict (1) of Example 2.4, the name of a model element was changed both in the repository and in the workspace version. The edit operation renameOperation(Ticket.getInfo, "getTInfo") will not fail, but would overwrite the change in the local workspace version. Of course, this cannot be allowed, and the user should be informed and warned about this conflict. More generally, a service is necessary which determines for each model element modified by an executable edit step whether it shall be considered as changed in v_2 compared to v_b. If so, the problem must be identified and handled somehow. A user can skip the respective edit step, take back the effect of this edit step later on or deliberately overwrite the local changes.

2.2.6 Consistency-preservation

Our analysis has shown that the propagation of changes to a given target model can fail for various reasons. It is no option to give up then and do nothing, the user must be enabled to analyze and resolve the problems somehow. The only solution is often to modify the target model, e.g. by creating missing model elements or modifying it in a way that a precondition of an edit step is fulfilled. Because the application of a patch can fail at any point of time, we conclude that the target model must be in a consistent, editable state after each successfully executed edit operation. In Figure 2.8, this requirement is represented by the mandatory feature *Consistency-preservation* of a change propagation tool.

This requirement holds both for textual and visual models. In the former case, it is typically fulfilled in a trivial way because even severely corrupted textual models can be corrected using standard text editors. In contrast to this, visual models are usually edited in their graphical representation, which can only be generated if the ASG which represents a model meets certain minimal consistency constraints. While it is possible in principle to represent ASGs textually, even if seriously incorrect, most users are either

incapable or unwilling to modify visual models in a textual representation. To that end, situations in which models cannot be edited using standard visual editors must be strictly avoided. The preservation of additional consistency constraints is desirable but not essentially required.

2.3 State of the Art

Version and variant management has been extensively researched since the advent of the first version control systems, e.g. SCCS [210] and RCS [241], in the 1970s and early 1980s. Besides basic storage services, version control systems essentially rely on a set of tool functions which are commonly referred to as differencing, patching and merging. We briefly describe and characterize these tool functions based on their input and output parameters. Here, we abstract from any technical details, notably the model representation. Thus, we consider each of these functions as an algebraic operator and state some basic properties which we expect from these operators, inspired by approaches from the related domain of model management [64, 179]. In the following, *Model*, *Difference* and *Matching* serve as basic data types; $m_1, m_2, \ldots, m_n$ are assumed to be models of the same type. If there is no confusion, indices are omitted.

The calculation of a difference between two versions of a model is the most fundamental function in model version and variant management:

$$\textbf{diff} : \mathit{Model} \times \mathit{Model} \rightarrow \mathit{Difference}$$

A difference $\Delta_{m_1 \rightarrow m_2}$ between two models m_1 and m_2 is a specification of how to transform m_1 into m_2. In the literature, this notion of a difference is usually referred to as *asymmetric difference* because $\Delta_{m_1 \rightarrow m_2}$ specifies changes only in one direction, i.e. from m_1 to m_2. In order to specify the changes from m_2 back to m_1, a different asymmetric difference is required. In other words, the *diff operator* is not commutative:

$$\text{diff}(m_1, m_2) \neq \text{diff}(m_2, m_1)$$

Please note that a directed delta as defined in [77] (cf. Section 1.3) is a special kind of asymmetric difference.

In contrast, a *symmetric difference* between two models m_1 and m_2 identifies all elements which are specific to m_1 and m_2, respectively [77]. Applying notions from set theory loosely, a symmetric difference between m_1 and m_2 may be written as $m_1 \, \Delta \, m_2 = (m_1 \cup m_2) \setminus (m_1 \cap m_2)$, where $(m_1 \cap m_2)$ denotes corresponding elements which are considered “the same” in m_1 and m_2, respectively. Such a relationship is usually referred to as *matching* and typically determined by a *match operator*:

$$\textbf{match} : \mathit{Model} \times \mathit{Model} \rightarrow \mathit{Matching}$$

Matching a model with itself should produce an identity relationship. Moreover, the match operator should be commutative, i.e. the order in which two models m_1 and m_2 are passed to a match operator should be irrelevant:

$$\text{match}(m_1, m_2) = \text{match}(m_2, m_1)$$

Obviously, symmetric differences can be easily derived from a given matching. To that end, many matching algorithms are actually called difference algorithms, e.g., LaDiff [70], MH-Diff [69], XyDiff [75], X-Diff [258], SiDiff [138], DSMDiff [162], UMLDiff [264], iDiff [185] or GenericDiff [267] to name some of them. However, within the scope of this thesis, we consequently refer to these algorithms as matching algorithms (cf. Section 2.3.1).

A *patch operator* applies a difference to a target model in order to generate a revised model:

$$\textbf{patch}: \textit{Model} \times \textit{Difference} \rightarrow \textit{Model}$$

The difference which is to be applied as patch is usually obtained by comparing an original model m_1 with a revised version m_2. In this case, we expect that the calculated difference applied to m_1 reconstructs m_2:

$$\text{patch}(m_1, \text{diff}(m_1, m_2)) = m_2$$

Note that patches can also be created using other methods, e.g. manually or by logging edit operations in syntax-based editors in closed development environments (s. Section 2.3.2). Nonetheless, the resulting difference must have the above mentioned property.

Merging refers to the process of combining two models of the same type into a single unified model. As a first significant characteristic, we can distinguish 2-way merging and 3-way merging. Whereas the former one tries to combine two models without any further information, the latter one tries to combine two models which are based on a common ancestor revision:

$$\textbf{2-way-merge}: \textit{Model} \times \textit{Model} \rightarrow \textit{Model}$$

$$\textbf{3-way-merge}: \textit{Model} \times \textit{Model} \times \textit{Model} \rightarrow \textit{Model}$$

Both 2-way merging and 3-way merging should be commutative, i.e. the order in which two models m_1 and m_2 which are to be merged (with respect to a common ancestor m_0) are passed to a merge operator should be irrelevant:

$$\begin{aligned} \text{2-way-merge}(m_1, m_2) &= \text{2-way-merge}(m_2, m_1) \\ \text{3-way-merge}(m_1, m_2, m_0) &= \text{3-way-merge}(m_2, m_1, m_0) \end{aligned}$$

Finally, merging should be idempotent, i.e. merging a model m with itself (with respect to a common ancestor m_0) should yield the same model m:

$$\begin{aligned} \text{2-way-merge}(m, m) &= m \\ \text{3-way-merge}(m, m, m_0) &= m \end{aligned}$$

In the remainder of this section, we give a brief overview of the state of the art on how to implement these version and variant management operators. We begin with the matching operator in Section 2.3.1, the calculation of a difference is addressed in Section 2.3.2. Different approaches to model merging are considered in Section 2.3.3, while Section 2.3.4 finally distinguishes patching from traditional 3-way merging.

Please note that the signatures of concrete implementations of our version and management operators might slightly differ from the above definitions. For example, some approaches require additional input parameters such as configuration data or meta-data from external resources. Such parameters are deliberately omitted from the above definitions in order to be as general as possible.

2.3.1 Matching Algorithms

The calculation of a matching is a fundamental utility function to identify occurrences of the same conceptual model element in different versions of a model.

Line-based processing of textual documents. A traditional approach in the context of source code versioning is to treat the documents which are to be compared as plain text, i.e. as a sequence of lines of characters. Most often, a line is considered as an atomic unit of comparison, i.e. two lines of text are considered to be equal if and if only the lines are identical. A common approach to determine the corresponding lines of texts in two documents is to compute a longest common subsequence [46] among the sequentially arranged lines of text. The computation of a longest common subsequence can also be considered as a solution to the string-to-string correction problem [257]; it directly delivers how one document can be transformed into another by applying a minimal number of line deletions and insertions. Further string-to-string correction algorithms which are based on different kinds of edit operations have been proposed in the literature. In particular, the original string-to-string correction problem as formulated in [257] also permits a *change* operation, [242] additionally covers block moves.

However, it is commonly agreed that comparing textual representations of visual models does not produce usable results [41, 63] and that models should be compared on the basis of their conceptual graph-based representation, which is usually referred to as structural matching.

Structural matching. Many graph matching algorithms have been proposed in the traditional computer science literature, a survey can be found in [78]. At first sight, some of them seem to be appropriate for structural model matching, in particular those which generalize the basic principles of the line-based approach to arbitrary graphs. For instance, the idea of computing an LCS can be generalized to the computation of a maximum common subgraph, where either the number of common nodes or the number of common edges is maximized. Assuming a set of primitive graph edit operations and a cost function that assigns editing costs to these operations, one can also try to determine minimal edit distances, which is commonly referred to as error-tolerant

graph matching [68], or, in the special case of trees, as the tree-to-tree correction problem [236]. However, virtually all of these problems are known to be NP-complete or NP-hard. Efficient algorithms are only available for special kinds of graphs such as ordered trees.

Besides performance problems, several conceptual problems arise when generic graph or tree matching algorithms shall be applied to models or other kinds of structured documents [106, 162, 198]. The main problem is that types of model elements and their conceptual properties are often neglected, which usually leads to a high false positive rate, e.g. if *i)* two model elements paired by a correspondence are actually not "the same" from a user's point of view, or *ii)* two model elements paired by a correspondence have different types, which usually leads to technical problems when a matching is to be further processed in a differencing pipeline or any other tool chain.

Thus, more practical approaches are based on heuristics which are specifically tailored to a dedicated document type. Amongst others, structural matching algorithms being based on language-specific heuristics have been proposed for hierarchically structured documents such as Latex documents [69, 70], XML documents [75, 258] or program source code [104, 185]. Domain-specific approaches have been developed in the context of schema matching [206] and ontology matching [256].

In general, most of the proposed heuristics are not directly applicable to models. Nonetheless, techniques being specifically designed for model matching are often inspired by the basic ideas of the proposed heuristics such as a type-specific notion of similarity between model elements, the use of signatures in order to (uniquely) characterize a model element or the integration of external resources into the matching process, e.g. lexical databases that provide additional information about semantically related terms and synonyms. A large number of approaches for comparing models were proposed recently (see [106, 146, 223] for surveys), which can roughly be divided into two categories; language-specific approaches and generic approaches. *Language-specific approaches*, as e.g. proposed for class diagrams [264], state charts [184], architectural diagrams [142, 171] etc., can only be applied to models of one specific type and will not be discussed in any further detail here. On the contrary, *generic approaches* can, if supplied with appropriate configuration data, be applied to models of many types. Configurations can be rather complex; in extreme cases, a configuration is written in a domain-specific language for configuring custom matching rules and policies [148].

Signature-based matching. Signature-based approaches [162, 198, 207, 224] match only elements which are "identical" in the sense that they have the same signature. The signature of a model element is a highly specific value that characterizes this element. Signatures can incorporate conceptual properties, e.g. the fully qualified name of an element, or technical properties such as persistent identifiers of model elements. If persistent identifiers are unique, the matching procedure is often called **UUID-based** [22] or **static identity-based matching**[3] [146]. The matching algorithm is fairly

[3] In the literature, UUID-based and signature-based approaches to model matching are sometimes treated as separate categories. The reason for this is that, in general, a signature-based matcher has

simple and very efficient: For each model, the signatures of all elements are computed and then sorted. A parallel scan through both sorted sets finds all signatures which occur in both sets. If such a pair of signatures is found and if both signatures are unique in their sets, a correspondence of the involved model elements is produced. No correspondence is typically produced in case of duplicates, i.e. if a signature occurs two or more times in the same model. The execution time is in the order of $O(n * log(n))$, the most dominant part being the sorting of the signature values.

Similarity-based matching. Similarity-based matching algorithms [63, 138, 162, 209, 267] try to match the mutually most similar, but not necessarily equal model elements. The configuration data defines, for each model element type, a set of properties which are relevant for the similarity and, for each property, a function which computes the similarity of two values of this property. Another function is needed which computes the overall similarity of two model elements. Matching two sets of model elements on the basis of similarities basically involves 3 steps:

1. The similarity of each pair of model elements is computed. An iterative calculation of similarities of all pairs of model elements allows to propagate similarities, i.e. the similarity of the "neighborhood" of two compared model elements can be "added" to their basic similarity, which is generally referred to as "similarity flooding" [178].

2. Whenever a similarity exceeds a threshold, both elements involved are considered as *candidates* for a correspondence between them. For each model element, all candidates in the other model are collected in a *preference list*. This list is sorted by the similarity of the candidates.

3. A matching is finally computed on the basis of these preference lists. For example, [267] constructs a bipartite graph from the preference lists and selects an optimal matching using a stable-marriage algorithm which is known as Gale-Shapley algorithm [109].

Steps 1 and 3 can be extremely time-consuming if large sets of model elements are to be matched, which motivates approaches that deviate from the general proceeding described above.

The approaches presented in [63, 266], for instance, avoid to globally search for pairs of mutually most similar model elements and rather apply a *top-down local* matching strategy which exploits the usual tree-like containment structure of models. A tree is spanned by the compositional structures defined by the corresponding meta-model. The model (or diagram) is the root of the tree, and root nodes are matched first. Then, for each pair of corresponding elements, the sets of their direct children are matched, and so on recursively. Matching is performed "locally" in the sense that only locally

to deal with duplicate signatures, while UUIDs are unique. This distinction is of minor importance here.

accessible data can be used to decide about correspondences and that only children of corresponding parents can be matched. This strategy is very efficient because only small sets of direct children have to be compared. However, it cannot detect moved model elements, i.e. it cannot match elements whose parents are not in a correspondence relationship.

The iterative matcher of the SiDiff framework can be configured such that the computation of similarities and the generation of correspondences are not strictly separated [138]. The iterative, incremental matching can be regarded as a generalization of similarity propagation, since correspondences can be produced during the propagation process. This option is very useful if elements of some dedicated types can be matched reliably very early. One example are start states in UML state charts, which are only defined once per region and which cannot be moved from one region to another. Correspondences between these elements may serve as "anchor points" for the propagation of similarities with the effect that the similarity flooding process usually converges much faster towards some stable result [2].

A similar performance optimization is pursued by the combination of a signature- and similarity-based approach [2, 162]: At first, a signature-based matcher is used to establish an initial set of correspondences. Subsequently, a similarity-based matcher deals with all remaining elements that have not been matched by the signature matcher.

Finally, the search space for the compute-intensive calculation of similarities can be reduced by preselecting promising matching candidates. The initial set of candidates of a model element can be usually restricted to all elements of the same type in the other model. There are further approaches to candidates initialization, e.g. a high-dimensional search tree as presented in [245].

2.3.2 Calculation of a Difference

Two main approaches to obtain a difference between two models have been proposed in the literature; *logging*, which is also known as *operation-based* differencing, and *state-based* comparison.

Logging of editing commands. A first approach based on logging is [164] and dates back to 1992; this approach assumed an object-oriented database management system (ooDBMS) to be used for storing documents, and each edit command to be executed as a transaction on the object base. The obvious idea was to exploit the logging facilities of the ooDBMS to generate change logs. The same basic idea was later adapted to change logs maintained in syntax-based editors, e.g. Fujaba [218, 219] and elsewhere [58, 62, 125, 143, 144].

Logging-based approaches have the big advantage that logs can be obtained directly on the level of user editing commands. However, they are bound to the fixed set of edit operations being offered by the editing environment. Moreover, documents must not be modified by tools which do not carefully maintain the change logs, change logs must never be deleted etc.; thus logging-based approaches imply closed environments, they are not applicable if models shall be exchanged across tool boundaries [154]. Moreover,

these approaches cannot compare models which are not revisions of each other, e.g. re-engineered models or the result of model transformations.

Another disadvantage is that a log produced by a user can contain tentative editing steps which are later corrected or which are a suboptimal approach for the desired editing effect. A trivial example are edit operations that cancel each other out. [57] reports about an experiment in which several users had to perform the same change in a model; very different paths to achieve the same effect were observed. Logs of edit commands are therefore not necessarily a good description of the intended changes, i.e. the logs must be optimized [101]. Moreover, change logs are often produced on the level of internal edit operations or basic storage operations [212].

State-based comparison of models. State-based methods for model differencing are only based on the state of the models which are to be compared. Virtually all of them, see [106, 223, 229] for surveys, have a similar processing pipeline like the basic differencing pipeline (steps 1 and 2) shown in Figure 1.2:

1. Initially, a matching procedure searches for pairs of corresponding model elements which are considered "the same" in both models.
2. Subsequently, a low-level difference is derived; elements not involved in a correspondence are considered to be deleted or created, each non-identical local property of corresponding elements is considered to be updated.

As a consequence, the obtained differences can only be explained in terms of primitive edit operations on the internal model representation. The UNIX `diff` utility, for instance, considers all lines of text which are not included in the longest common subsequence to be deleted or inserted. State-based approaches to model differencing conceptually consider models as graphs and explain a difference in terms of primitive graph editing operations, i.e. they basically report insertions and deletions of nodes and edges, respectively.

In contrast to textual editing, graph-based edit operations can depend on one another. For example, a node has to be first created before it can be connected to other nodes via edges. If only primitive edit operations are available for modifying an ASG, dependencies between edit steps do not have to be explicitly calculated. All edit steps can be applied in a standardized order; insertions are performed first, followed by update operations and finally deletions (cf. [22]).

If a matching is being determined by computing a minimal edit distance (s. Section 2.3.1), the above pipeline steps 1 and 2 cannot be clearly separated, as a sequence of elementary edit steps is usually delivered as a direct result of the matching algorithm.

2.3.3 Model Merging

Most currently available merge tools are based on the principle of 3-way merging. In particular, virtually all existing approaches for updating workspaces of models are

based on this principle. The main challenges in 3-way merging are conflict detection and conflict resolution. Many approaches for these problems have been proposed in the literature; surveys can be found in [28, 106, 180]. The proposed approaches can be roughly categorized into *state-based merging* and *operation-based merging*.

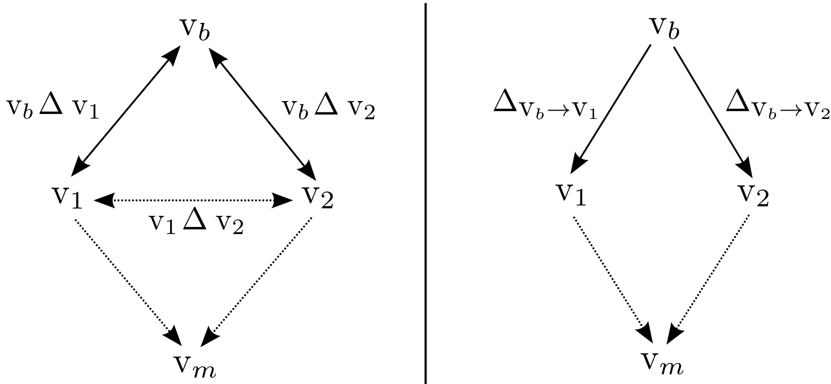


Figure 2.9: State-based (left) vs. operation-based merging (right)

State-based merging. State-based merging is the dominant approach in the field of classical versioning systems. Merging procedures such as `rcsmerge`, `SVN update` etc. are typically based on an adaptation of the `diff3` utility distributed with the UNIX diffutils package [169]. `diff3` merges two versions v_1 and v_2 with respect to a third version v_b, assuming v_b is the common ancestor version of v_1 and v_2. First of all, longest common subsequences are determined along three ways as shown on the left side of Figure 2.9; between v_b and v_1, between v_b and v_2, and between v_1 and v_2, indicated as symmetric differences in Figure 2.9. While symmetric differences $v_b \Delta v_1$ and $v_b \Delta v_2$ are explicitly calculated using the UNIX `diff` utility, $v_1 \Delta v_2$ is implicitly derived assuming transitivity of correspondences. Such a “diff3 parse” [141] determines the segments of lines that are (a) the same in all three versions, (b) the same in two versions, or (c) different in all three versions.

The merge result is well-defined for all segments of type (a) and for all segments of type (b) where either v_1 or v_2 is the differing version. In the latter case, the segment of lines that differs from v_b is included in the merge result v_m. All other cases indicate a conflict. Conflicts resulting from line segments which are different in all three versions, i.e. segments of type (c), are called overlaps. In case of conflicts, the default behavior of `diff3` is to include all possible variants into the final merge result, separated by special conflict markers. If `diff3` is adapted in the context of a version control system, non-overlapping conflicts are usually suppressed. In case of overlaps, only the segments of v_1 and v_2 are included as possible variants into the final merge result, the third variant originating from the common base version is usually omitted.

The basic principle of state-based merging has later been adopted to tree-based representations of different kinds of documents [31, 163] and to merge models based on their ASG representation, see e.g. [260]. Thus, the detection of conflicts is shifted

from a pure textual to a syntactical level. A key feature of syntactical merging is to avoid "pseudo-conflicts" that arise from different formattings of textual representations. Moreover, sophisticated merge procedures taking the conceptual structure of the underlying documents into account are also capable of detecting language-specific conflicts which are known as static semantic conflicts [180], context-sensitive conflicts [260], or violations [60]. Conflict ② of Example 2.4 presented in Section 2.1.5 is an example of this. Some approaches go one step further and aim at the detection of behavioral conflicts. To that end, reasoning techniques which are based on denotational semantics, program slicing or program dependence graphs have been applied to behavioral conflict analysis in the context of software merging [180, 225, 238]. Some of these techniques have been also adopted for model merging [27], which is, however, only possible for modeling languages with well-defined behavioral semantics.

2-way merging is obviously state-based and can be considered as special case of 3-way merging. However, the notion of a merge conflict varies significantly between both variants. While behavioral and static semantic conflicts can be sensibly transferred to 2-way merging [204, 213], the traditional notion of an overlapping change is obviously not applicable without having a common ancestor version.

Operation-based merging. The principle of operation-based merging has first been proposed in [164] and is illustrated on the right side of Figure 2.9. Here, two asymmetric differences $\Delta_{\mathrm{v}_b \to \mathrm{v}_1}$ and $\Delta_{\mathrm{v}_b \to \mathrm{v}_2}$ from the common base version v_0 to v_1 and v_2 are computed. These differences are analyzed for conflicting edit steps, then conflicts are resolved, and finally a merged version v_m is generated. In [164], the term conflict refers to directly contradicting edit steps, i.e. two operation invocations which do not commute. [164] leaves open how documents are represented internally and how edit operations are implemented. The basic proceeding may be applied to any abstract data type, provided that there is a decision procedure for testing the commutativity of two operation invocations.

The asymmetric differences $\Delta_{\mathrm{v}_b \to \mathrm{v}_1}$ and $\Delta_{\mathrm{v}_b \to \mathrm{v}_2}$ do not necessarily have to be obtained by logging editing commands, but can also be calculated by a comparison of v_b against v_1 and v_2. If only primitive edit operations such as inserting/elements and changing their basic properties are supported, as e.g. in [22, 63, 161], the boundary between state-based and operation-based merging somewhat disappears, because the changes induced by these primitive edit operations are also implicitly derived from the symmetric differences $v_b \,\Delta\, v_1$ and $v_b \,\Delta\, v_2$ of Figure 2.9 (left). In [180], the special flavour of operation-based merging which is based on primitive edit operations is also referred to as *change-based merging.* Although not intended (and actually not needed because all operations of a data type are assumed to maintain its data type invariants) in [164], the merge result obtained by a change-based (operation-based) merging procedure can be checked for violations of consistency constraints [60, 235].

Handling of Conflicts

Both for state-based and operation-based approaches we can further distinguish whether conflicts are resolved *interactively* or *non-interactively*. In the former case, merge decisions are delegated to the user during the merge process. In operation-based and change-based merging, the user can deliberately skip an edit step involved in a conflict [145, 164]. Moreover, 2-way merging is, in general, a highly interactive process [106]. Classical merge techniques known from RCS or SVN are non-interactive. This leads to the problem of what to do in case of conflicts. Two common solutions have already been implicitly addressed with the description of the `diff3` utility: Firstly, in the case of overlaps, all possible variants and some additional conflict markers are included in a preliminary merge result which has to be post-processed by the developer. Secondly, in the case of conflicts that result from non-overlapping changes, the variant that has been changed is being preferred. These basic principles, namely the creation of a unified document and automated conflict resolution, have been also adopted for models.

Creation of a unified document. A unified document [186, 198] is a "brute force" merger of v_1 and v_2 in the sense that all information contained in both versions is united. In case of conflict ① of Example 2.4, i.e. competing name changes, both names are shown side by side at this model element. Deleted model elements are not deleted, but only marked as deleted. These simple examples show that even if a unified document looks very similar to an original model it is actually a new type of model with a different meta-model which represents information about local differences between v_1 and v_2. In some cases, the graphical representation must be changed, too. As a result of this, several additional non-trivial new tools or tool modifications are needed for displaying and editing unified documents, for resolving conflicts and for converting the preliminary merge result into the original modeling language.

A similar approach in which overlay techniques are used to show parallel changes has been proposed in [177]. The approach presented in [55, 261] proposes to represent information about changes and conflicts in a unified diagram referred to as "tentative merge" using stereotypes and tagged values, i.e. by using UML profiles. This avoids some of the problems mentioned above, but creates new problems in case of DSMLs such as SysML [191] or MARTE [192], which are defined as profiles, too.

In sum, the unified document has an attractive GUI if differences are small. Larger differences will mostly overload the graphical representation. Complex edit operations with many parameters can only represented graphically by using an own node which is connected to other nodes which serve as arguments. Dependencies between changes are hard to represent.

Automated conflict resolution. One simple heuristic for taking standardized merge decisions is to assign the least priority to deletions [95]. If basic graph operations are used, anything which existed in v_1 or v_2 remains and it is impossible to identify the cancelled deletions in the merge result, unless the relevant information is conveyed using other means. This approach fails completely if edit operations are more complex

and replace graph fragments, i.e. they both delete and create nodes and/or edges of an ASG.

Another approach, which is proposed in [73], is to define conflict resolution patterns for pairs of conflicting edit operations. This leads to a very high specification effort for realistic modeling languages because there are many pairs of edit operations which potentially cause a conflict and for each such pair a solution must be developed. Moreover, it is unlikely that a conflict resolution pattern always guesses right what the user actually wants.

2.3.4 Patching vs. 3-Way Merging

The principle of document patching is sketched in Figure 2.10. Initially, a document version v_0 is compared with another document version v1. v_0 is referred to as *original document* of the patch, v_1 as *changed document*. The resulting patch $\Delta_{v_0 \to v_1}$ represents the changes between v_0 and v_1, it is a specification of how v_0 must be changed in oder to produce v_1. Patching in a narrow sense is the final step of repeating the changes specified in the patch on the *target document* v_t, which in general resides in a different repository.

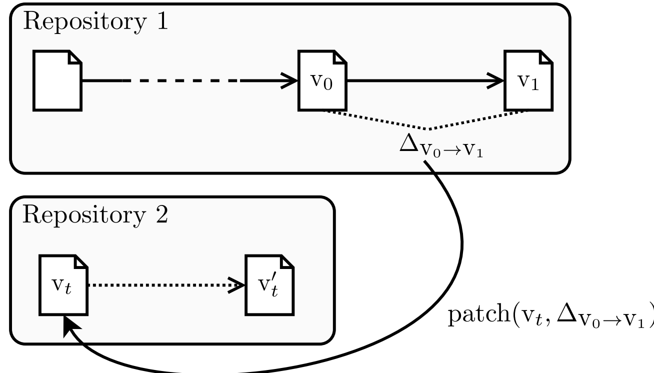


Figure 2.10: Basic principle of document patching

Misleadingly, patching is often referred to as “merging”, where merging is used in the sense of 3-way merging. However, patching differs from 3-way merging by the absence of a common base version. Versions v_0 and v_t are, in general, unrelated and managed in different repositories. Thus, traditional notions of a merge conflict (s. Section 2.3.3) cannot be applied immediately. Moreover, existing 3-way merge tools for models assume that all involved versions reside in one repository. They do not need to address the problems related to transferring and executing a patch on a different host.

Text-based patching. The combination of the UNIX standard tools `diff` and `patch` can be used to propagate changes if models are represented as textual files. The basic proceeding is as follows: First, a patch file is created by applying `diff` to the origin and the changed version of a document. A patch file represents an asymmetric difference

between the original and the changed version. It basically consists of a set of "hunks" [169], each hunk represents an edit step of the difference. A hunk describes which changes, i.e. insertions and deletions of lines of text, should be made to a certain block of text. In so-called "normal" patches [169], text blocks are identified by *absolute line numbers* in the base version. `patch` takes the patch file as input and applies the contained changes to the target document, producing a patched version which is put in place of the target document.

This traditional usage of patch is well-suited to apply differences to a copy of an original file as in scenario SC2 of Section 2.1. However, in scenarios SC4 and SC5 we have to apply a patch to a target model which differs from the original model. Even if the same model element appears in the original and the target model and if it is represented by the same block of text lines in both versions, these blocks can have different positions in both files. Absolute line number are thus no reliable identifiers for model elements.

The `diff/patch` tools offer a feature to cope with the problem of relocated text blocks. `diff` can add a specified number of context lines at each hunk of a patch. Such patches are commonly referred to as "context" patches. `patch` can be instructed to exploit these context lines when searching for the correct place to apply a hunk: rather than using the absolute line numbers, it searches for the context lines in the target file. If a context cannot be found, the respective hunk is put out to a reject file, i.e. `patch` can (partially) *fail now*.

There are two options to deal with this problem of non-applicable (or partly applicable) patches: Firstly, a patch can be adapted using any standard text editor, which is not a very attractive option for several reasons. A second option is to apply the patch and to reasonably ignore up to a specified number of context lines which are to be matched in the target document. However, the "fuzzification" of a patch application comes with a certain degree of uncertainty, i.e. a change might be applied at a wrong "position" in the target document. Thus, the result of the patch application has to be double-checked manually. While this approach is feasible in the case of textual documents, it is hardly applicable to visual models: At worst, the application of a patch can make a model so incorrect that it can no longer be processed by standard model editors.

ASG-based patching. In contrast to model merging, there are only a few approaches which directly address the patching of models.

The approach presented in [74] considers the application of a patch as *one* model transformation and aims at using a model transformation system (ATL) [131] as a technology to execute patches. The transformation which implements a patch is mechanically derived from the difference between two models. If a patch shall be applied to a model different from the original model, a so-called weaving model must be supplied, which is basically a matching between original and target model. It seems as though an application of a patch will never fail. Users are not supported in case of errors, e.g. violations of consistency constraints or references resolved wrongly. The controlled

editing of patches is not supported. The same observations apply to any approach based on the principle that a complete patch is implemented by *one*, possibly huge transformation rule: Model transformation systems were not designed to support the independent execution of parts of a transformation rule, the order in which primitive change actions of a rule are executed is unknown and cannot be influenced.

EPatch [85], which is available as part of the EMFCompare project [86] appears to support only our first application scenario, the patching of the original model using low-level changes. It is based on Xtext [92] and relies on persistent identifiers of model elements, which are pathnames derived from XMI identifiers. EMFCompare itself does not offer a patching facility.

2.4 Conclusions

Effective model version and variant management strongly depends on sophisticated tools supporting a variety of different scenarios. An important variation point in the design of versioning tools for models is whether models are externally represented and edited in a textual or visual editor, which in turn depends a lot on the modeling language. Traditional approaches to model versioning can be conveniently applied to all kinds of textual models. It is worth here to briefly recall why these tools have been successfully applied to all kinds of textual documents, notably source code, in the past:

(i) For the purpose of comparing, patching and merging, the documents can be regarded as pure texts without syntactic constraints. A simple text thus can be regarded as a low-level implementation of a source program, which is intended to represent a syntax tree.

(ii) Programmers are able and used to modify this low-level representation of a syntax tree when they actually want to change the syntax tree. For example, adding a line somewhere in the text can conceptually mean to insert a statement in a sequence of statements contained in the body of a method.

(iii) Representations of differences, conflicts or rejected edit steps of a patch on some output media can be constructed easily in the textual format with which the user is familiar.

(iv) Syntax errors produced by merging or patching can be removed manually by normal text editors.

These favorable and oftentimes simplifying conditions (i) - (iv) are no longer true in the case of visual models:

(i) Each model type has its own syntactic constraints which must be taken into account by difference functions.

(ii) The set of available user-level edit operations which shall be used to describe and specify model changes strongly depends on the model type.

(iii) Visual representations of differences, conflict resolution dialogues and further user interactions should be adapted to each model type, i.e. they should be based on the familiar representations of models.

(iv) Merge and patching results must conform to the effective consistency level imposed by model editors, there are no commonly used "low-level editors" which can remove significant syntax errors.

We conclude that each model type requires model management tools and difference functions being specifically tailored for each modeling language, development process, MDE environment, and individual user preferences. To that end, much research has been stimulated in the last decade in order to provide fundamental difference functions in the same quality as for textual documents. While sophisticated model matchers are becoming available and are actually used in industrial practice [216], differencing, patching and merge tools for visual models are not nearly as mature and widely used as their traditional counterparts for textual documents. The main deficiencies will be briefly summarized in the remainder of this section.

Understandability of low-level differences. State-based difference tools for models deliver poor results if applied to visual models. The main reason for that is that differences are based on primitive edit operations on the ASG instead of describing changes in terms of user-level edit operations.

Edit operations which appear to be elementary from a user's point of view can lead to many low-level changes on the ASG. For example, the creation of a generalization relationship in a UML class diagram leads to the creation of a node of type *Generalization* and three edges connecting this node to ASG nodes representing super- and subclass of the generalization relationship (cf. Figure 3.3). This simple example shows already that model differences can be hard to understand for modelers who are not familiar with meta-modeling and the related internal representation of models in terms of an ASG.

There are several other examples where a seemingly simple change in the external representation of visual models causes significant structural changes on the ASG; restricting the navigability of an association to one end (s. Figure 4.1), dragging an association end to another class, or changing the text which represents the multiplicity of an association end, a parameter list, or a list of stereotypes; [137] discusses further examples in state machines and similar types of models.

The above problems are mainly caused by different internal and external representations of models; the problem thus depends to some extent on presentation options of graphical modeling tools and might disappear, or be completely different, in a textual representation of models such as the Human-Usable Textual Notation (HUTN) [188] proposed by the OMG.

Restructurings caused by complex edit operations inherently lead to a large number of changes and are therefore another reason why low-level differences can become very confusing. For example, the extraction of a composite state in a UML state chart results

in many changes of the ASG which depend on the actual invocation parameters. In terms of our Example 2.2, these changes can be briefly summarized as follows:

- A new composite state called *Active* is created and embedded into the state machine.
- Several simple states (*DialTone*, *Dialing*, *Invalid*, *Connecting*, *Busy*, *Ringing*, and *Talking*) and the respective transitions between them are moved to the new composite state.
- Transition *lift* is conceptually split into two transitions; one from state *Idle* to the new composite state *Active*, and another one from the new initial state of *Active* to *DialTone*.
- Finally, transitions *hangup* originating from simple states *DialTone*, *Dialing*, *Invalid*, *Connecting*, *Busy*, *Ringing*, and *Talking* are folded to a top-level transition from *Active* to *Idle*.

Please note that each of the above change descriptions, which are still rather conceptually oriented, leads to a large number of low-level changes if we compare the ASGs of model versions v_1 and v_2 of Example 2.2.

Violation of the effective consistency level. The basic procedures of 3-way merging (s. Figure 2.9) and patching (s. Figure 2.10) do not depend on whether edit operations preserve the consistency of models. Most approaches to 3-way merging of models use primitive graph operations [22, 63], which can violate consistency constraints in ASGs. For example, creating a *Generalization* object without connecting it to nodes of type *Class* by edges *general* and *specific* leads to an inconsistent model which cannot be processed by code generators and, much worse, which cannot be visualized graphically or even textually. This causes serious problems in 3-way merging or patching scenarios: Conflict resolution in 3-way merging requires some edit steps involved in conflicts to be excluded from the final merge result; edit steps in patches cannot be executed if a referenced model element was deleted from the target model. Ultimately, only a subset of the edit steps contained in a directed delta is actually executed in these cases. Consequently, it is not guaranteed that the merge result can be displayed graphically.

This problem re-appears in the presentation of conflicts and user dialogues for resolving conflicts. Both must use a tree representation of models (as e.g. in [86, 161]), which is not very user-friendly and likely to cause mistakes. The problem to re-establish the consistency of a merged model using a tree representation is additional work imposed on users. In some common cases, e.g. the representation of associations between classes, a very good comprehension of the meta-model is needed, which can hardly be assumed for average users.

Complexity of merge tools. In principle, some sophisticated approaches to 3-way merging can be applied to combine visual models without violating the effective level of consistency. In particular, operation-based merging can be performed on the basis of user-level edit operations preserving the effective level of consistency. However, tools being based on this concept are hard to implement and difficult to use for the following reasons:

In principle, each edit step of $\Delta_{\mathrm{V}_b \rightarrow \mathrm{V}_1}$ can be in conflict with each edit step of $\Delta_{\mathrm{V}_b \rightarrow \mathrm{V}_2}$; either of them cannot be included in the merge result. Thus, the analysis, representation and resolution of conflicts is often very complex. A complete representation of all relevant information in 3-way merging comprises (a) the two asymmetric differences $\Delta_{\mathrm{V}_b \rightarrow \mathrm{V}_1}$ and $\Delta_{\mathrm{V}_b \rightarrow \mathrm{V}_2}$, (b) the model versions v_b, v_1 and v_2, and (c) all conflicts and their interrelations.

3-way merging forces a user to mentally return to the revision v_b and to partly or even completely repeat their own work since the last update. This repetition of work can be very tedious if there are many edit steps. If a conflict occurs in the middle of a long sequence of edit steps the current state of the model can be unclear. Repeating all edit steps between v_b and v_2 allows users to skip single, own edit steps which are in conflict with changes between v_b and v_1. This seems like a method for easy resolving conflicts with almost no effort. However, simply skipping own operations is mostly no acceptable solution. Experiences reported in [261] indicate that complex conflicts cannot be resolved in this way. A solution must often incorporate both changes, i.e. the effect of the skipped edit step must be manually re-established in a slightly modified form. Conflict ③ of our Example 2.4 is an example of this.

In fact, as revealed by the study conducted by Estublier et al. [99], structural and semantically enhanced 3-way merging techniques in the field of source code versioning had little to no impact on the industrial practice and were never adopted by commercial versioning systems. As argued in [99], the reasons of this can lie in the level of complexity required to master an idea or in the level of effort required by a customer to use a particular feature. According to our own experience with an early prototype of a 3-way merge tool [217], we believe that the same observations most likely apply to models, although there is yet no empirical evidence for this assumption.

Substantial lack of patch tools. Finally, Section 2.3.4 reveals that there is a substantial lack of patch tools for models. Tools available are not nearly as mature and flexible as the widely used UNIX standard tools diff and patch. In particular, the detection and handling of errors in the application of patches is not supported. Thus, existing approaches seem to support only the trivial patching scenario SC2. Moreover, conflict handling techniques, as proposed in the context of 3-way merging, can not be simply adopted for model patching due to the lack of a common base version.

CHAPTER 3

Representation and Editing of Models

Edit operations on models can only be specified precisely if a (runtime-) representation of models is available. To that end, the concept of *meta-modeling* has been established in the MDE community. A meta-model is basically a data model of the conceptual contents of a given class of models which is commonly referred to as *model type* or *modeling language* (cf. Section 2.2.1). Meta-models and ASGs can be implemented in various technical frameworks (cf. Section 8.3). In this section, we abstract from these technological details. To that end, Section 3.1 introduces our conceptual notion of an ASG which is formally considered as *typed, attributed, directed graph.* A meta-model defines the allowed types of nodes and edges of an ASG and is formally treated as a distinguished graph called *type graph.* All concepts can be transferred to technologies and frameworks which are based on the essential MOF (EMOF) standard being defined as part of the OMG MOF 2.0 specification [193].

In this thesis, we focus on monolithic models (or sub-models) which are usually treated as self-contained development documents by standard editors. In this regard, a model is typically considered to be syntactically consistent if it conforms to its meta-model (or a subset of a meta-model). Beyond basic typing correctness as defined in Section 3.1, models usually must meet a set of additional consistency constraints which will be considered in Section 3.2.

Our notion of a graph introduced in Section 3.1 is compatible with the graph model of the model transformation language and system Henshin [34]. Henshin is based on graph transformation concepts and is used in our approach to precisely specify the available edit operations for a given modeling language. Section 3.3 introduces the

basic principles of our rule-based approach to the specification of edit operations and briefly reviews the basic concepts of the Henshin model transformation language.

3.1 Graph-based Representation of Models

Figure 3.1 informally introduces the graph model being applied in this thesis. As usual in object-oriented modeling, graphs can be used at two levels which are commonly known as *instance level* (or object level) and *type level*. An (instance) graph basically consists of two disjoint sets containing the nodes and edges of the graph. Altogether, the nodes and edges are referred to as the elements of the graph. An edge is a directed connection between two nodes, which are called the source and target nodes of the edge. A type graph declares the available types of nodes and edges in an instance graph and thereby defines the allowed instance structures of graphs being typed over a fixed type graph. A type graph is a special graph which may contain further notational constructs to define type hierarchies, containment structures, and to declare types of edges which shall be conceptually regarded as bidirectional edges in an ASG. Moreover, multiplicities can be attached to edge types in order to restrict the allowed instance structures. Both type graphs and instance graphs may be attributed. Attributes defined by a type graph have to be considered as attribute declarations; we specify a name and a data type for a certain attribute. In an instance graph, we may assign any value from the data type's range of possible values to this attribute.

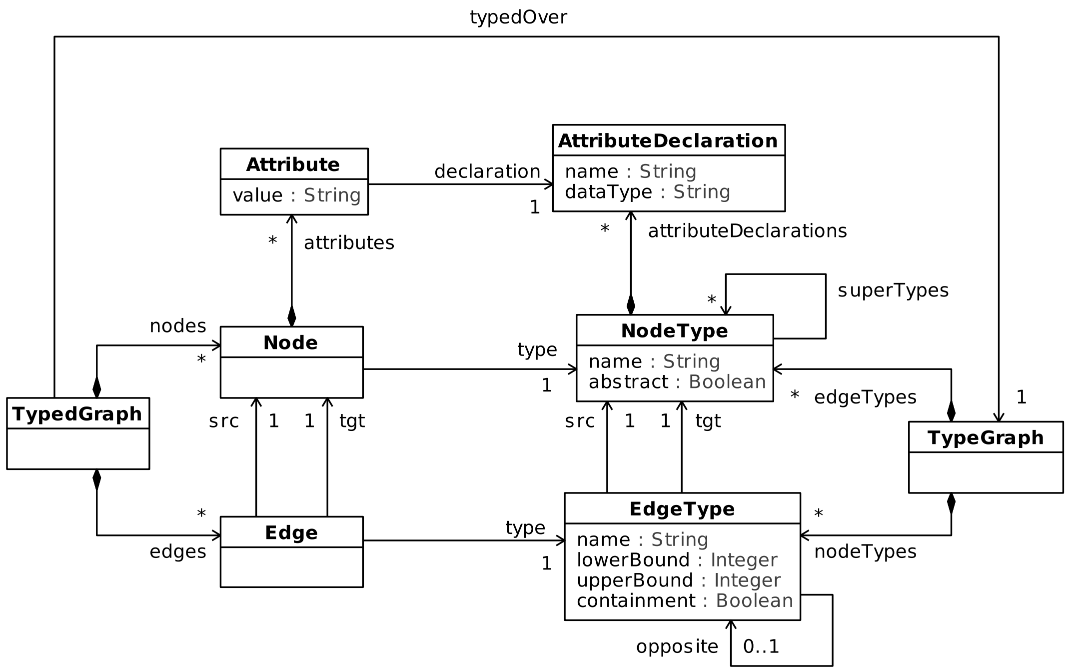


Figure 3.1: Attributed, directed graphs being typed over a fixed type graph with inheritance, containment, opposites and multiplicities

Please note that the graph model of Figure 3.1 also serves as a data model for the representation of graphs. In order to be precise, we will introduce formal definitions for representing MDE models as graphs in the remainder of this section. The mapping of the data model of Figure 3.1 to this "semantic domain" is mostly straightforward, hints will be given in case of ambiguities.

3.1.1 Graphs and Graph Mappings

Formally, we consider a graph as a directed multigraph in which source and target nodes of an edge are defined by dedicated functions.

Definition 3.1 (Graph)
A graph G is a 4-tuple $G = (G_N, G_E, src_G, tgt_G)$ which consists of a set G_N of nodes, a set G_E of edges, as well as source and target functions $src_G, tgt_G : G_E \to G_N$. If any of the functions src_G and tgt_G is a partial function, we say that G is a partial graph (or graph fragment) which intuitively differs from a graph in that it has so-called dangling edges.

We use an object-oriented infix notation to refer to the source and target nodes of an edge $e \in G_E$, i.e. we write $e.src$ instead of $src_G(e)$ and $e.tgt$ instead of $tgt_G(e)$, respectively (note that we omit indices for src_G and tgt_G in the object-oriented notation). In addition, we use functions $in, out : G_N \to \mathcal{P}(G_E)$ to refer to incoming and outgoing edges of a particular node. For each $n \in G_N$,

- the set of *incoming edges* is defined by

$$in(n) = \{e \in G_E \mid e.tgt = n\}. \tag{3.1.1}$$

- the set of *outgoing edges* is defined by

$$out(n) = \{e \in G_E \mid e.src = n\}, \tag{3.1.2}$$

Definition 3.2 (Graph mapping)
Given two graphs G and H, a pair of functions (f_N, f_E) with $f_N : G_N \to H_N$ and $f_E : G_E \to H_E$ forms a (total) graph mapping $f : G \to H$. A mapping is *edge-preserving* if it maps the nodes and edges of G to those of H in a structure-compatible way, i.e. for each edge $e_G \in G_E$ there is a corresponding edge $e_H = f_E(e_G) \in H_E$ such that

- $e_G.src$ is mapped to $e_H.src$, and
- $e_G.tgt$ is mapped to $e_H.tgt$.

An edge-preserving graph mapping is usually called *graph (homo-)morphism.* If the mapping is injective, the morphism is called a *graph monomorphism*; each node/edge in

G is mapped to a distinct node/edge in H, whereas H may have additional nodes/edges. If the mapping is bijective, the morphism is called a *graph isomorphism*.

We write $G \rightsquigarrow H$ if if mappings f_N and f_E are allowed to be *partial*. Notations $dom(f)$ and $ran(f)$ are used to refer to the *domain* and *range* of a partial mapping. The mapping f is a partial graph morphism if $dom(f) = (dom(f_N), dom(f_E))$ is a graph.

Basic binary operations known from set theory can be applied to graphs if there is a precise specification which elements of the input graphs are to be considered "identical". Such a precise specification can be, for instance, provided by a partial injective mapping.

Definition 3.3 (Intersection, union and difference of graphs)
Let G and H be graphs and $f : G \rightsquigarrow H$ be a partial injective mapping, then the "identical" elements of G and H are defined as follows:

- $n_G = n_H$ with $n_G \in G_N$ and $n_H \in H_N \Leftrightarrow f(n_G) = n_H$
- $e_G = e_H$ with $e_G \in G_E$ and $e_H \in H_E \Leftrightarrow f(e_G) = e_H$

Thereupon, the *intersection*, *union* and *difference* of G and H are defined componentwise for the sets of nodes and edges, respectively:

$G \cup H := (G \cup H)_N = G_N \cup H_N$ and $(G \cup H)_E = G_E \cup H_E$

$G \cap H := (G \cap H)_N = G_N \cap H_N$ and $(G \cap H)_E = G_E \cap H_E$

$G \setminus H := (G \setminus H)_N = \{n \mid (n \in G_N) \wedge (n \notin H_N)\}$ and $(G \setminus H)_E = \{e \mid (e \in G_E) \wedge (e \notin H_E)\}$

The result of the above operations is well-defined if $G \cap H$ is a graph. In this case, $G \cup H$ constitutes a graph, too, while $G \setminus H$ may result in a graph fragment.

Further operations which can be derived from the basic operations above are defined as usual. In particular $G \subseteq H$ denotes that G is a *subgraph* of H. In this case, the mapping f is an *inclusion*. Having defined the subgraph relationship, we can finally introduce our notion of a boundary graph which we will need later in this thesis.

Definition 3.4 (Boundary graph)
Let G be a graph, then the boundary graph of a fragment $F \subseteq G$ is the smallest graph $B \subseteq G$ completing F to a graph.

3.1.2 Meta-models as Type Graphs

In MDE, the allowed types of nodes and edges are defined by a meta-model: Types of nodes are specified by classes, while types of edges are specified by association relationships between classes. In this context, classes (associations) are often called metaclasses (meta-associations) to point out that they are used here for the purpose of language engineering, in contrast to their traditional usage in application engineering.

A meta-model can be formally treated as a distinguished graph called type graph whose nodes and edges represent node types and edge types, respectively [79]. MOF-based meta-models use several concepts which are known from object-oriented analysis and design, notably inheritance, containment, bidirectional associations and multiplicities[1]. To that end, we adopt here the approach presented in [51] which introduces so-called type graphs with inheritance and containment. Our formalization of multiplicities is based on [96].

Definition 3.5 (Type graph)
A type graph $TG = (T, I, A, C, OE, mult)$ consists of a graph $T = (T_N, T_E, src_T, tgt_T)$ representing the node types and edge types defined by a meta-model. In addition, TG consists of the following components:

$I \subseteq T_N \times T_N$ is an acyclic relation that formalizes the *inheritance* relationships defined by a meta-model. For each $x, y \in T_N$ and $(x, y) \in I$, x refers to the *subtype* and y to the *supertype* induced by an inheritance relationship.

$A \subseteq T_N$ identifies *abstract node types* in the inheritance hierarchy.

$C \subseteq T_E$ introduces a distinguished kind of edge type called *containment edge type* in order to represent meta-associations that are declared to be composite relations.

$OE \subseteq T_E \times T_E$ is a binary relation being introduced in order to represent bidirectional meta-associations with the help of *opposite edge types*, i.e. pairs of edge types (et_1, et_2) with $et_1.src_T = et_2.tgt_T$ and $et_2.tgt_T = et_1.src_T$. The relation OE is anti-reflexive and symmetric. Furthermore, OE is unique, i.e. each edge in a type graph can have at most one opposite edge.

$mult : T_E \rightarrow Mult$ denotes a function that maps each edge type to a *multiplicity* invariant. The range $Mult$ denotes the set of multiplicities, and a multiplicity is a pair $[lb, ub] \in \mathbb{N} \times (\mathbb{N} \cup \{*\})$ with $lb \leq ub$ or $ub = *$.

Multiplicities. Please note that a multiplicity attached to an edge type refers to the target end of an edge type, i.e. it specifies the number (more precisely, an integer interval defined by lower and upper bounds lb and ub, respectively) of nodes which may be connected to one source node via edges of a certain edge type. Given an edge type $et \in T_E$, we write $et.lb$ and $et.ub$ to refer to the lower bound and upper bound defined by $mult(et)$. In contrast to [96], multiplicities that refer to the source end of an edge type are not allowed here.

[1]Note that we deliberately abstained from formalizing some MOF concepts such as property redefinition, property subsetting and association specialization for two reasons. Firstly, the OMG standard [193] lacks a precise semantics for some aspects of these concepts, especially w.r.t. their interrelations [24, 52]. Secondly, these concepts are of minor practical importance since they are typically not supported by available reference implementations of the MOF standard (cf. Section 8.3).

mult	abbr.	**many**	**required**	**bounded**	**fixed**
0..1				✓	
0..x		✓		✓	
0..*	*	✓			
1..1	1		✓	✓	✓
1..x		✓	✓	✓	
1..*		✓	✓		
x..x	x	✓	✓	✓	✓
x..y		✓	✓	✓	
x..*		✓	✓		

$1 < x < y < *$

Table 3.1: Multiplicity properties of different multiplicity invariants.

It is often useful to classify edge types according to the upper bound (*ub*) and lower bound (*lb*) of its multiplicity invariant. An edge type $et \in T_E$ is called

- *required* if $et.lb > 0$
- *bounded* if $et.ub \neq *$
- *fixed* if $et.lb = e.ub$
- *many* if $(et.ub > 1)$ or $et.ub = *$.

Note that these properties are not mutually exclusive, i.e. an edge type can have several of these properties. Table 3.1 illustrates these properties for different multiplicity invariants.

Please also note that upper bounds of multiplicities are represented as *Integer* values in our data model of Figure 3.1. The special symbol $*$, which is used to specify an *unbounded* multiplicity ($\neg$*bounded*), is represented as negative *Integer* value.

Super- and subtypes. We use functions $super, allsuper, sub, allsub : T_N \rightarrow \mathcal{P}(T_N)$ to refer to the supertypes and subtypes of a node type. For each $x \in T_N$,

- the set of direct supertypes is defined by

$$super(x) = \{y \in T_N \mid (x, y) \in I\}, \tag{3.1.3}$$

- the set of all supertypes is defined by

$$allsuper(x) = \{y \in T_N \mid (x, y) \in I^+\}, \tag{3.1.4}$$

- the set of direct subtypes is defined by

$$sub(x) = \{y \in T_N \mid (y, x) \in I\}, \tag{3.1.5}$$

- the set of all subtypes is defined by

$$allsub(x) = \{y \in T_N \mid (y, x) \in I^+\}, \tag{3.1.6}$$

where I^+ is the transitive closure of the inheritance relation I. Obviously, we have $super(x) \subseteq allsuper(x)$ and $sub(x) \subseteq allsub(x)$.

Furthermore, we use functions $src_I, tgt_I : T_E \to \mathcal{P}(T_N)$ which, given an edge type $et \in T_E$, deliver the source (target) node type $et.src$ ($et.tgt$) of et and all subtypes of $et.src$ ($et.tgt$). For each $et \in T_E$, we have

$$src_I(et) = \{et.src\} \cup allsub(et.src), \tag{3.1.7}$$

and

$$tgt_I(et) = \{et.tgt\} \cup allsub(et.tgt). \tag{3.1.8}$$

We write $et.src_I$ ($et.tgt_I$) when using function src_I (tgt_I) in an infix notation.

Moreover, we use functions $in_I, out_I : T_N \to \mathcal{P}(T_E)$ to refer to all incoming and outgoing edge types (including inherited edge types) of a particular node type. For each $nt \in T_N$, we have

$$in_I(nt) = in(nt) \cup \big(\bigcup_{nt' \in allsuper(nt)} in(nt')\big), \tag{3.1.9}$$

and

$$out_I(nt) = out(nt) \cup \big(\bigcup_{nt' \in allsuper(nt)} out(nt')\big). \tag{3.1.10}$$

Finally, we slightly adapt the usual notion of a *path* in a directed graph to type graphs. When we say that there is a path $(et_1, et_2, ..., et_n)$ of edge types in a type graph, we mean that there would be a path in the corresponding flattened type graph.

3.1.3 Models as Typed Graphs

We formalize the typing relation between models and meta-models, which is often called *instanceOf*-relation, by a special graph morphism relating a typed graph with its associated type graph.

Definition 3.6 (Typed graph and typing morphism)
An instance graph G is typed over a fixed type graph TG if there is a typing relation $type_G : G \to T$ which maps the nodes and edges of G to those of T in a structure-compatible way, i.e. for each edge $e_G \in G_E$ there is a corresponding edge $e_T = type_G(e_G) \in T_E$ such that

- $e_G.src$ is mapped to $e_T.src$ or any subtype $x \in allsub(e_T.src)$, and
- $e_G.tgt$ is mapped to $e_T.tgt$ or any subtype $x \in allsub(e_T.tgt)$.

The relation $type_G$ is called typing morphism.

A typing relation associates each node and each edge of a typed graph with exactly one type to which we refer to as *node type* and *edge type*, respectively. For brevity, we write $n.type$ instead of $type_G(n)$ to refer to the node type of a node $n \in G_N$. Analogously, $e.type$ refers to the edge type of an edge $e \in G_E$. We adapt our basic definition of a graph mapping to typed graphs. In particular, we distinguish between *type-preserving* and *type-compatible* mappings of typed graphs. The latter notion is based on the object-oriented principle of polymorphism.

Definition 3.7 (Type-preserving and type-compatible mapping)
Let G and H be graphs which are typed over the same type graph TG, then a mapping $f : G \to H$ is called type-preserving if it maps the nodes and edges of G to those of H such that

(1) $n.type = f_N(n).type$ for each $(n, f_N(n)) \in f_N$, and

(2) $e.type = f_E(e).type$ for each $(e, f_E(e)) \in f_E$.

The mapping f is called type-compatible if it allows types of nodes in G to be more general than types of the corresponding nodes in H. To that end, the above condition (1) is relaxed such that only one of the following conditions must hold:

- $n.type = f_N(n).type$, or
- $n.type \in allsuper(f_N(n).type)$.

An edge e with $e.type \in C$, where $C \subseteq T_E$ denotes the containment edge types defined by the type graph, is called *containment edge*. Its source and target nodes are referred to as *parent* (or *container*) and *child*, respectively. Typed graphs are called *rooted*, if there is a distinguished node, the *root node*, which contains (transitively) all other nodes of the graph. The target node of a non-containment edge, i.e. an edge e with $e.type \notin C$, is called *neighbor* of the respective source node $e.src$.

Sets of incoming and outgoing edges of a particular node in a typed graph can be partitioned by edge types. Consequently, functions *in* and *out* according to (3.1.1) and (3.1.2) are redefined for typed graphs. Let G be an instance graph over a type graph TG, then we have functions $in, out : G_N \times T_E \to \mathcal{P}(G_E)$ with the following semantics: For a node $n \in G_N$,

- the set of *outgoing edges* of a particular type $et \in T_E$ is defined by

$$out(n, et) = \{e \in G_E \mid e.src = n \wedge e.type = et\}, \quad (3.1.11)$$

- the set of *incoming edges* of a particular type $et \in T_E$ is defined by

$$in(n, et) = \{e \in G_E \mid e.tgt = n \wedge e.type = et\}. \quad (3.1.12)$$

3.1.4 Attributed Graphs

As shown in Figure 3.1, nodes in an instance graph may have attributes. Each attribute can be assigned a data value from the domain of possible values defined by the data type of the attribute declaration to which this attribute refers to. Our formal notion of an attributed graph is based on [122]. Here, type graphs and instance graphs are modeled as a pair $AG = (G, A)$ of a graph G and a data type algebra A. A *data type* is formally represented by a data sort defined by the algebra, while *data values* are represented by elements of the carrier set of a data sort.

In an *instance graph*, data values are represented as nodes which are referred to as *value nodes* in order to distinguish them from ordinary *object nodes*. Object nodes are linked to value nodes by attributes, i.e. an attribute is a special edge connecting an object node with a value node. Value nodes must not have outgoing edges. Furthermore, only nodes may have attributes, i.e. edges cannot be linked to value nodes.

The same principle can be applied to *type graphs*. Here, data types are represented as nodes of a type graph, and ordinary node types can be linked to data types by special edge types representing attribute declarations. Finally, the typing morphism of Definition 3.6 can be extended to an *attributed graph morphism* in order to define typed attributed graphs, a formal treatment can be found in [122].

Please note that our notion of a typed attributed graph can be regarded as a special kind of an ordinary typed graph. For simplicity, we assume attributes to be single-valued, `null`-values are not allowed (cf. Figure 3.1), which is the usual case in common meta-models and modeling frameworks. Formally, a multiplicity of [1..1] is implicitly assigned to each attribute declaration in a type graph. Concerning the approach presented in this thesis, this simplification is without loss of generality; all concepts can be easily transferred to multi-valued attributes.

3.1.5 Visual Representation of Type and Instance Graphs

In this section, we illustrate how to formally treat meta-models and models as defined in the context of OMG-related standards. Thereby, we introduce the visual syntax used in this thesis in order to externally represent type and instance graphs in a compact manner.

Example 3.1 (UML meta-model of class diagrams as type graph)
Meta-models as defined in the context of the OMG can be translated to our notion of a type graph in a straightforward way. The left part of Figure 3.2 shows an excerpt of the UML meta-model for class diagrams as defined in the UML Superstructure Specification [190]. On the right, we illustrate how this excerpt is mapped to an attributed type graph with inheritance, containment, opposites and multiplicities. Basically, each meta-class is represented by a node in the corresponding type graph. Each unidirectional meta-association corresponds to a normal edge in the type graph. Bidirectional meta-associations are modeled by two separate edges that are declared to be opposite to each other (cf. opposite relation OE). Node and edge types are shown as rectangles,

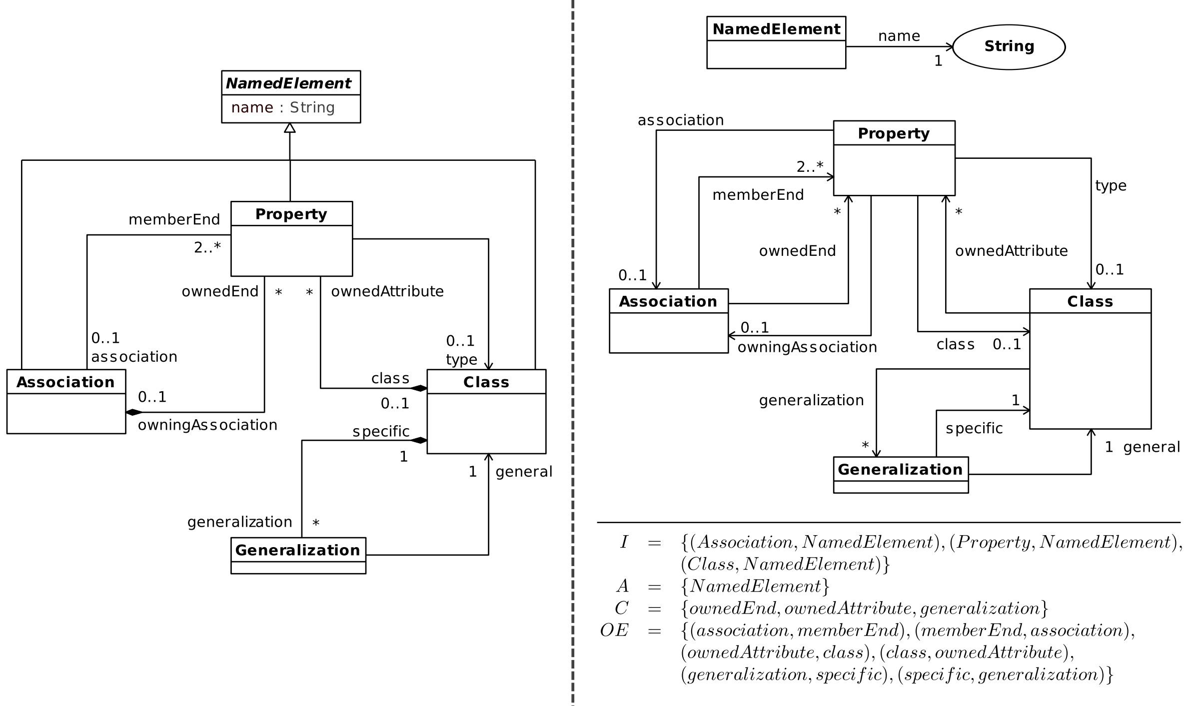


Figure 3.2: Subset of the UML meta-model as type graph with inheritance, containment, opposites and multiplicities: UML-like syntax (left) vs. formal syntax (right)

a data type DT being used by an attribute declaration $a : DT$ is shown as ellipse. Multiplicity invariants are shown in the usual UML-like $[lb..ub]$ notation; 1 and $*$ are used as shorthand notations for $[1..1]$ and $[0..*]$. Inheritance relationships, abstract types, and containment edge types are defined by relation I as well as sets A and C, respectively.

In the remainder of this thesis, we will use the more compact UML-like syntax. In particular, attribute declarations are shown as an integral part of the definition of a node type, and pairs of edge types being opposite to each other are shown as one visual edge without arrowheads.

Example 3.2 (ASG representation of a UML class diagram as typed graph)
Figure 3.3 shows parts of the ASG representation of our class diagram example of Figure 2.1, namely the classes *Person* and *Developer* as well as the generalization relationship between these classes. We use here the concrete syntax of UML object diagrams; the notation $n : T$ represents a node n of type T; n is used as symbolic identifier (which can actually be omitted) and T refers to a node type of the corresponding type graph. Edge types are attached to edges without the use of symbolic identifiers. On the left, the notation $a = v$ is used to express that a value v is assigned to attribute a. On the right, the formal representation is shown with attribute values being modeled as value

nodes of the graph.

In the remainder of this thesis, we will use the UML-like syntax on the left where attribute values are modeled inside a dedicated compartment of object nodes. For the sake of readability, we also indicate containment edges in an ASG using the UML-like syntax, i.e. a diamond is attached to the source end of a containment edge.

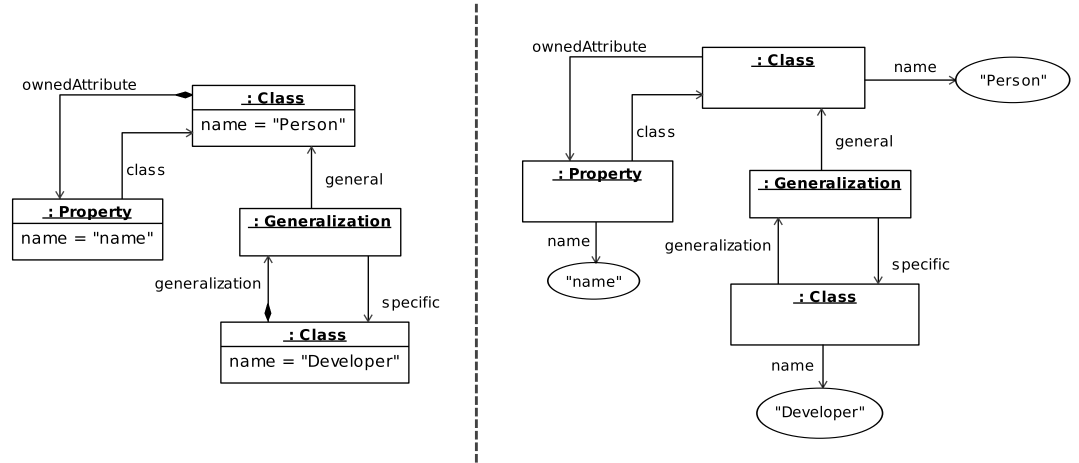


Figure 3.3: ASG representation of a sample UML class diagram as typed graph: UML-like syntax (left) vs. formal syntax (right)

3.2 Consistency of Models

A model is considered (syntactically) consistent if it is properly typed (s. Section 3.1.3), and if it meets additional consistency constraints which will be considered in the remainder of this section. In general, we distinguish among *basic consistency constraints*, *multiplicity invariants* and further, "arbitrary" *well-formedness rules*. In accordance with the requirements revealed by our domain analysis (cf. Section 2.2), our main objective is to identify those conditions which are required to produce external representations of models. In turn, these conditions obviously depend on the properties of standard editors of an MDE environment. In order to formally document these properties, we finally introduce our notion of an *effective meta-model*.

Basic consistency constraints. Basic consistency constraints correspond to fundamental conditions imposed by EMOF-based modeling frameworks. A formal treatment of basic consistency constraints can be found in [51]; they can be summarized as follows:

- **at-most-one-container** and **no-containment-cycles**: The concept of containment leads to two constraints to which we refer to as *at-most-one-container* and *no-containment-cycles*: Each ASG node must have at most one container and cycles of containment edges must not occur.

- **no-parallel-edges**: A consistent ASG must further adhere to the *no-parallel-edges* constraint, i.e. it must not contain two edges of the same type linking the same source and target node. The reason behind this constraint is that ASGs are typically implemented by a set of objects and references between them. Mainstream object-oriented languages do not assign internal identifiers to references (we cannot reference a reference). Consequently, two references of the same type linking the same source and target object cannot be uniquely distinguished.
- **all-opposite-edges**: For every pair of edge types (et_1,et_2) that are declared to be opposite to each other, the *all-opposite-edges* constraint must be satisfied by an ASG: For all edges of type et_1 there must be also an edge of type et_2 linking the same nodes in the opposite direction.

Multiplicity invariants. Multiplicity invariants define another kind of consistency constraint which restricts the allowed instance structures in an ASG. Although they are typically not enforced by standard modeling frameworks such as EMF, many of them are required for generating external representations of visual models in standard editors. A multiplicity attached to an edge type refers to its target end (s. Definition 3.5) and specifies the number of nodes which may be connected to a source node via edges of the given edge type. In other words; for each node in an ASG, a multiplicity invariant restricts the number of outgoing edges of a particular type:

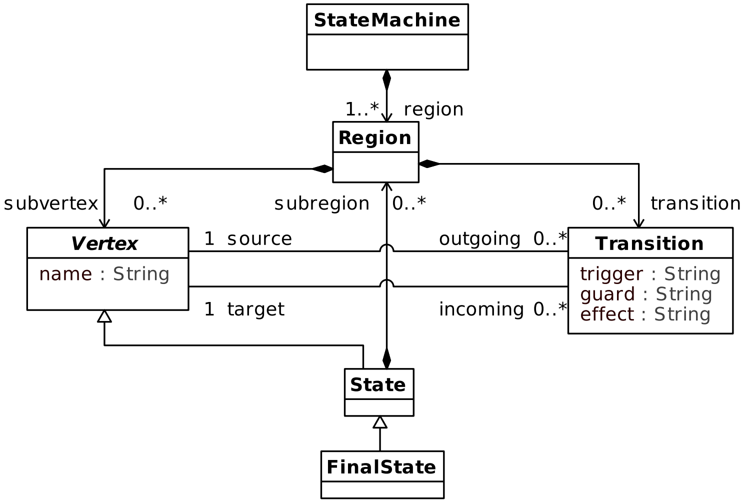
Let G be a model represented as ASG being typed over a type graph TG, then for each edge type $et \in TG$ and for each node $n \in G_N$ two conditions must hold to which we refer to as **no-lower-bound-violation** and **no-upper-bound-violation**, i.e.

$$et.lb \leq |out_{et}(n)| \leq et.ub \text{ or } et.ub = *, \tag{3.2.1}$$

where $|out_{et}(n)|$ denotes the cardinality of the set of edges of type et outgoing from node n.

In particular, edge types having multiplicity property *required* lead to mandatory neighbors and mandatory children [224]: Let $et \in T_E$ be an edge type with multiplicity property *required* (i.e. $et.lb > 0$). Then, in an instance graph G, each node $n \in G_N$ of type $et.src$ (or any of its subtypes in $allsub(et.src)$) must be connected to a number of $et.lb$ nodes of type $et.tgt$ (or any of its subtypes in $allsub(et.tgt)$) via edges of type et. If et is a non-containment edge type, we refer to these nodes as **mandatory neighbors** of n. Otherwise, if et is a containment edge type, we say that node n has $et.lb$ **mandatory children**.

Well-formedness rules. Meta-models such as the UML Superstructure Specification [190] typically define further advanced well-formedness rules beyond basic consistency constraints and multiplicity invariants. For example, Table 2.1 lists several well-formedness rules for UML state machines which are specified in natural language. In the context of MOF-based meta-models, formal specifications of these rules are usually expressed using the Object Constraint Language (OCL) [195]. Figure 3.4 shows OCL expressions for well-formedness rules (a), (b) and (q) of Table 2.1.



Additional well-formedness rules:

```
context FinalState
inv (a): self.outgoing->size() = 0
inv (b): self.region->size() = 0

context Region
inv (q): self.subvertex->forAll( memb |
          self.subvertex->excluding(memb)->forAll(other |
            memb.name <> other.name))
```

Figure 3.4: Effective meta-model of simple UML state machines according to typical visual UML editors

Effective level of consistency and effective meta-models. The degree of consistency achieved by satisfying all of the above mentioned constraints can be called "perfect" because models meeting these constraints can be interpreted or correctly translated to source code. However, as analyzed in Section 2.2.6, typical visual editors do not enforce all consistency constraints. In particular, only a small subset of the well-formedness rules is actually required. The effective level of consistency can be represented by starting from the perfect meta-model and by dropping all constraints except the relevant ones. Moreover, certain multiplicity invariants may be relaxed. We will refer to the resulting meta-model as the *effective meta-model* of a dedicated editing environment.

Definition 3.8 (Effective meta-model)
An effective meta-model precisely defines the effective level of consistency which is required by standard editors in order to generate an external representation of an ASG. An effective meta-model may differ from its corresponding perfect (or standard)

meta-model in two aspects; an effective meta-model may *i)* contain only a subset of the set of well-formedness rules defined by the standard meta-model, and *ii)* multiplicity invariants may be relaxed, i.e. lower bounds may be decreased while upper bounds may be increased.

Example 3.3 (Effective meta-model of UML state machines)
As an example, the effective meta-model of UML state machines according to typical visual UML editors is shown in Figure 3.4. Note that only a subset of the set of well-formedness rules defined by the UML Superstructure Specification is explicitly listed. Here, none of the multiplicity invariants has been relaxed as the specified lower bounds are of particular importance: Transitions can only be visually displayed if they are properly connected with their mandatory neighbors of type *Vertex*; regions of a state machine are typically used as "canvas" in visual editors, thus a *StateMachine* node has one mandatory child of type *Region*.

Based on our notion of an effective meta-model, consistency of models can be finally defined according to Definition 3.9.

Definition 3.9 (Consistency of models)
A model is consistent if it is properly typed, adheres to all basic consistency constraints, and satisfies all multiplicity invariants and well-formedness rules defined by the effective meta-model.

3.3 Rule-based Specification of Edit Operations

Meta-models are just data models of models and do not directly specify editing behavior. One obvious option to modify a model is to use *primitive graph editing operations* such as creating/deleting a node/edge of an ASG or setting an attribute value. These edit operations are *atomic* insofar as they are the smallest possible changes of an ASG. Primitive edit operations are *generic* because they have arguments which designate types, i.e. they can operate on ASGs with arbitrary meta-models.

In simple cases, an elementary user-level edit operation (cf. Section 2.2.2) can be implemented by one primitive operation on an ASG. However, many elementary edit operations can only be implemented by grouping several primitive ASG operations in a single transaction. For example, creating a generalization relationship in a UML class diagram boils down to four primitive graph operations: The creation of a node of type *Generalization* and the creation of three edges which connect the *Generalization* node to *Class* nodes via edges of type *general*, *specific* and *generalization* (cf. Figure 3.3).

An important characteristic that distinguishes user-level operations from primitive operations on an ASG is that their effect always leads to a consistent state of an ASG, i.e. an external representation can be created in a standard editor (cf. Definition 3.9). Thus, from a technical point of view, we refer to such an operation as *consistency-preserving edit operation* (CPEO). Please note that primitive edit operations are, in

general, not consistency-preserving. For example, if only a subset of the above primitive edit operations is executed, the effect might lead to a "dangling generalization relationship" in the visual representation of a class diagram. Typical visual editors will not create a diagram representation at all (s. Section 2.2.1)

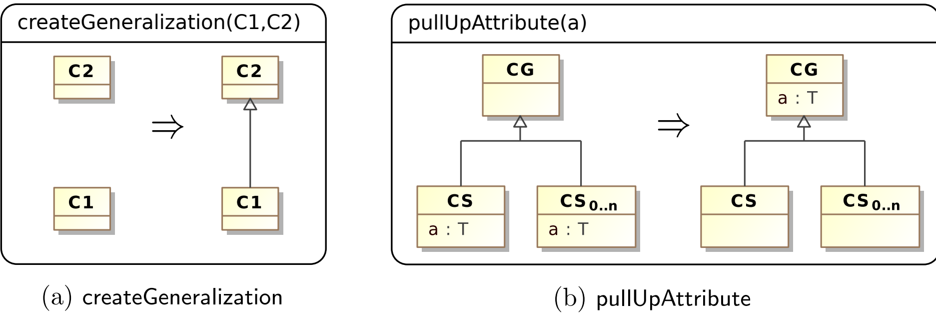
Definition 3.10 (Consistency-preserving edit operation)
A consistency-preserving edit operation (CPEO) is an edit operation that can be applied to any consistent model and, if executed successfully, transforms the model into another consistent model.

Obviously, the effect of complex edit operations such as refactorings or other high-level evolutionary operations (cf. Section 2.2.2) cannot be achieved by single primitive edit operations, too. We can conclude that implementations of user-level edit operations depend on the effective meta-model of a given modeling language, i.e. they are not generic but have to be individually engineered for each modeling language and editing environment. To that end, user-level edit operations have to be specified in a suitable way. Section 3.3.1 shows how edit operations can be informally specified in an intuitive way. Section 3.3.2 explains how edit operations can be implemented based on the model transformation language Henshin. Henshin is based on graph transformation concepts [94] which can be exploited to reason about potential conflicts and dependencies between edit operations as illustrated in Section 3.3.3.

3.3.1 Informal Specifications of Edit Operations

Informally, edit operations can be specified using natural language. Such specifications can be illustrated using the concrete syntax of the given modeling language. If required, the concrete syntax can be enriched with additional notational constructs in order to clarify the semantics of edit operations. Two examples of this for UML class diagrams are shown in Figure 3.5. The left side of such a specification defines the context in which an edit operation can be applied. The right side represents the new state of the diagram after the edit operation has been applied. An application of the operation can be thought of as replacing the diagram pattern on the left side with that on the right. Figure 3.5 (a) serves as an informal specification of the edit operation createGeneralization(C1,C2) that establishes a generalization relationship between classes C1 and C2 such that C2 becomes the superclass of C1.

Another example is the refactoring operation pullUpAttribute(a) shown in Figure 3.5 (b). In contrast to createGeneralization(C1,C2), pullUpAttribute(p) specifies a variable change pattern on UML class diagrams. It is variable in the sense that it is applied to all common attributes in the set of all direct subclasses of a class. The term "common" here means that all attributes must have the same name and the same UML data type. To that end, pullUpAttribute(a) is defined to take one dedicated attribute a as input which is bound to class CS and which is to be "pulled up" to the superclass of CS (called CG in Figure 3.5). Moreover, an arbitrary number of common attributes is to be deleted from all other direct subclasses of CG (indicated as $CS_{0..n}$).



(a) createGeneralization (b) pullUpAttribute

Figure 3.5: Informal specifications of edit operations on UML class diagrams

3.3.2 Implementing Edit Operations in Henshin

In-place model transformations are well-suited to define model refactorings, see e.g. [49, 147, 175, 181, 203, 231]. Of course, other edit operations can be specified by in-place model transformations as well, as shown e.g. in [233]. We use the model transformation language Henshin [34] for this purpose. This enables us to specify and at the same time implement edit operations as declarative transformation rules, to which we refer as *edit rules*. Arguably, model transformation rules are "executable specifications", the usual distinction between specification and implementation does not apply here.

Roughly spoken, a Henshin rule specifies *i)* the *conditions* under which the rule is applicable and *ii)* a set of *change actions* which are to be performed on a model when the rule is applied. Each change action corresponds to a primitive graph operation, i.e. the creation/deletion of a node/edge in the ASG, or the change of an attribute value. Please note that we introduce Henshin here as a visual language for the formal specification of graph transformation rules. Abstract syntax and execution semantics of Henshin rules are based on graph transformation concepts [94].

Definition 3.11 (Henshin transformation rule)
A Henshin transformation rule r is defined on the ASG and consists of the following components:

L The *left-hand side* is a graph pattern which specifies the pattern that has to be found in a model for applying r. L may include checks of attribute values which are specified by constants or variables.

R The *right-hand side* specifies a graph pattern which replaces L if r is applied. R may include changes of attribute values specified by expressions including declared variables.

Cxt_r is the intersection of L and R. It identifies that part which shall remain unchanged when r is applied.

$r : L \rightsquigarrow R$ is used as a shorthand notation for the above components. The arrow symbolizes a partial injective mapping from L to R which is type- and edge-preserving, i.e. a subgraph of L is mapped to R.

PAC_r, NAC_r are sets of positive and negative application conditions; each condition $c \in PAC_r \cup NAC_r$ is an extension of L. In general, conditions c are not graphs, but graph fragments. They also may contain attribute conditions over declared variables.

In addition, we use the following notations for components which are implicitly defined by the partial injective mapping $r : L \rightsquigarrow R$:

Del_r $= L \setminus R$ is the fragment of L which is to be deleted when r is applied.

Cre_r $= R \setminus L$ is the fragment of R which is to be created when r is applied.

Note that in general, Del_r and Cre_r are not graphs, but just fragments. We introduce boundary graphs B_{Del} and B_{Cre} completing Del_r and Cre_r to graphs:

$B_{Del} \subseteq L$ is the boundary graph for Del_r.

$B_{Cre} \subseteq R$ is the boundary graph for Cre_r.

Analogously to B_{Del} and B_{Cre} above, we introduce boundary graphs for application conditions. Given a condition $c \in PAC_r \cup NAC_r$, then B_c refers to the boundary graph completing c to a graph. Note that an application condition $c \in PAC_r \cup NAC_r$ can be translated to a *postcondition* of the same rule r if its boundary graph B_c is preserved by rule r [94].

Application of Henshin rules. The left-hand side of a rule r can have several matches ("occurrences") in a model M. A *match* is an injective mapping $m : L \rightarrow M$ assigning a concrete value to each declared variable. Moreover, m must be edge-preserving and type-compatible, i.e. the type of a node in a rule graph (or rule graph fragment) may be more general than the type of the corresponding model node. Note that rule graphs and rule graph fragments used in the left-hand side or in application conditions may contain nodes which are typed over abstract node types. In contrast, nodes which are to be created must be concretely typed.

A rule r is *applicable* at match m if m can be extended such that each $pac \in PAC_r$ can be matched and no $nac \in NAC_r$ can be matched. In order to guarantee that an application of r is free of side effects, we additionally require the so-called *dangling condition* of the *double-pushout approach* (DPO) to algebraic graph transformation [94] to be fulfilled; each edge in M which is incident to a deleted node in $m(Del_r)$ must have an origin in Del_r. This means that the context of a node which is to be deleted must be specified completely. Thus, dangling edges are not deleted implicitly, as opposed to the *single-pushout approach* (SPO) to algebraic graph transformation (cf. [211]).

The *effects* of applying an applicable rule r using match m in M can be described as follows (a full treatment can be found in [94]):

1. The fragment $m(Del_r) \subseteq M$ is deleted from M.

2. The fragment Cre_r is inserted into M as a fresh copy and connected with $m(Cxt_r)$.

3. Attribute values are changed according to defined expressions.

We write $M1 \overset{r,m,n}{\Longrightarrow} M2$ to express that model $M1$ is transformed into model $M2$ by applying rule r using match m. The *Co-match* $n : R \rightarrow M2$ shows how the right-hand side is part of model $M2$.

Visual syntax of the Henshin transformation language. Henshin offers an intuitive visual syntax to specify model patterns to be found and preserved, to be deleted, to be created, to be required and to be forbidden [34]. The idea is to integrate all components of a transformation rule r in a unified diagram such that the effect of r can be easily seen. Context elements in Cxt_r are colored in grey and annotated with the stereotype *preserve*, elements in Del_r are colored in red and annotated with the stereotype *delete*, elements in Cre_r are colored in green and annotated with the stereotype *create*. A positive application condition $pac \in PAC$ is specified as a model pattern which is annotated with the stereotype *require*, while a negative application condition $nac \in NAC$ is specified as a model pattern annotated with the stereotype *forbid*. Application conditions of the same type are distinguished using symbolic identifiers attached to the elements of the respective model patterns, i.e. elements sharing the same identifier belong to the same application condition.

Example 3.4 (Implementing edit operation createGeneralization in Henshin)
As an example, edit rule createGeneralization(c1, c2, gen) is shown in Figure 3.6. The example illustrates that a Henshin rule can define variables serving as input or output parameters. Here, input parameters c1 and c2 determine sub- and superclass between which a generalization relationship is to be created. The created node of type *Generalization* is returned as output parameter gen when the rule is applied.

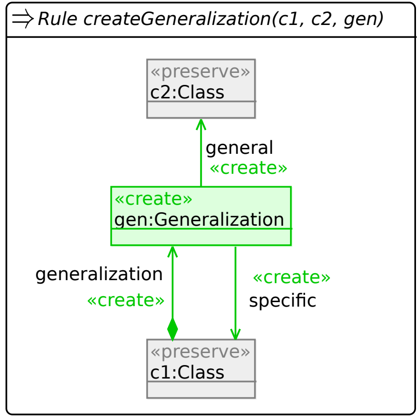


Figure 3.6: Implementing edit operation createGeneralization(c1, c2, gen) in Henshin

Rule schemes. Although "simple" Henshin transformation rules can combine a number of primitive graph operations into one transaction, the variability which is described by our informal specification of the edit operation pullUpAttribute of Figure 3.5 cannot be formally specified on that basis. We need an additional concept to express recurring model patterns, which we call *multi-object structures*.

Henshin offers so-called "nested rules" (a.k.a. rule schemes or interaction schemes) to specify transformations of multi-object structures which are based on the concept of amalgamation [50]. A *rule scheme* contains a *kernel rule* and an arbitrary number of *multi-rules*. A kernel rule is a simple transformation rule which can be equipped with a set of multi-rules which include the kernel rule. Each multi-rule specifies one multi-object structure and its transformation. A rule scheme is applied as follows: The kernel rule is applied once. This match is used as a common partial match for each multi-rule, which are matched as often as possible. Thus, multi-object structure transformations are performed as often as corresponding structures occur in a given model.

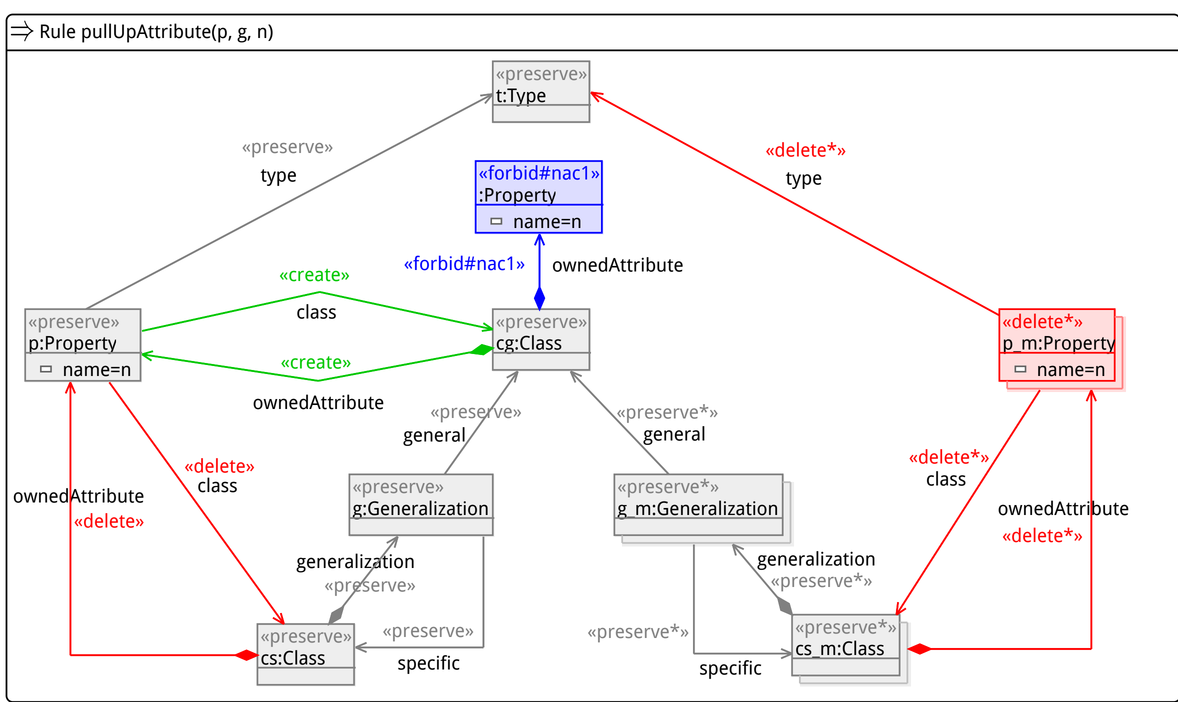


Figure 3.7: Refactoring pullUpAttribute(p,g,n) as rule scheme

Example 3.5 (Refactoring pullUpAttribute as rule scheme)
The effect of the refactoring operation pullUpAttribute can be achieved by a rule scheme, the basic idea is shown in [34] for a simplified UML meta-model; its kernel rule moves one of the "redundant" attributes to the common superclass, one multi-rule is needed to delete the remaining attributes from all sibling classes in the inheritance tree. The rule scheme pullUpAttribute(p,g,n) shown in Figure 3.7 adopts this basic principle, while all type definitions are based on the UML Superstructure Specification [190]. Note that

in the visual syntax of the Henshin transformation language, all multi-rule elements are indicated by the star symbol * and rendered as stacked rectangles.
The *kernel rule* takes three parameters p, g, and n as input. Note that p and g are *object parameters* which are to be bound to ASG nodes, while n is a *value parameter* which is to be bound to a primitive data value. Property p with name n is the attribute which shall be pulled up from class cs to superclass cg, but only if cg does not yet own an attribute with name n (s. nac1 in Figure 3.7). As UML classes may have multiple superclasses, the concrete superclass cg of cs is determined by the generalization relationship g given as input parameter. Variables cs, cg and t are bound implicitly when the kernel rule is applied. All attributes named n having the same type t as p are deleted from all other direct subclasses of cg by applying the *multi-rule* as often as possible. Variables cs_m, cg_m and p_m are bound implicitly when the multi-rule is applied.

3.3.3 Potential Conflicts and Dependencies

A set of transformation rules can be statically analyzed for potential conflicts and dependencies using the technique of critical pair analysis (CPA). Critical pair analysis has been initially developed in the context of term rewriting and is used there to check if a term rewriting system is confluent [37]. The theory of critical pairs and confluence has been transferred to transformation systems for typed attributed graphs [94, 122]. A critical pair provides a minimal example of a conflicting situation, or of a situation in which two rule applications depend on each other.

Two rule applications $G \overset{r_1,m_1,n_1}{\Longrightarrow} G1$ and $G \overset{r_2,m_2,n_2}{\Longrightarrow} G2$ are in conflict if the first rule application invalidates the second one. There are four reasons why rule applications can be conflicting:

- **delete/use:** The application of rule r_1 deletes a model element that is used by the match of r_2 including its PACs.
- **change/use:** The application of rule r_1 changes the value of an attribute that is used by the match of r_2 including its PACs.
- **create/forbid:** The application of rule r_1 creates a model element that a NAC of r_2 forbids.
- **change/forbid:** The application of rule r_1 changes the value of an attribute that is checked by a NAC of r_2.

Two subsequent rule applications $G1 \overset{r_1,m_1,n_1}{\Longrightarrow} G2$ and $G2 \overset{r_2,m_2,n_2}{\Longrightarrow} G3$ are dependent on one another if the first rule application enables the second one. The following four kinds of dependencies are possible if a rule r_2 is applied after a rule r_1:

- **create/use:** The application of rule r_1 produces a model element that is used by the match of r_2 including its PACs.
- **change/use:** The application of rule r_1 changes the value of an attribute that is used by the match of r_2 including its PACs.

- **delete/forbid:** The application of rule r_1 deletes a model element that a NAC of r_2 forbids.
- **change/forbid:** The application of rule r_1 changes the value of an attribute that is checked by a NAC of r_2.

Definition 3.12 (Critical pair)
A critical pair is a pair of rule applications that are conflicting (dependent) in a minimal context. A critical pair represents a potential conflict (potential dependency) between two rules. If there is no potential conflict (potential dependency) between two rules, they are said to be parallel independent (sequentially independent).

Without considering application conditions, minimal critical graphs to which two rules r_1 and r_2 can be applied *in a conflicting way* are computed by overlapping L_1 (the left-hand side of rule r_1) with L_2 (the left-hand side of rule r_2) in all possible type- and structure-compatible ways. Each critical graph G together with conflicting rule applications $G \overset{r_1,m_G,n_{G1}}{\Longrightarrow} G1$ and $G \overset{r_2,m_G,n_{G2}}{\Longrightarrow} G2$ constitutes a critical pair which represents a potential conflict between the two rules r_1 and r_2.

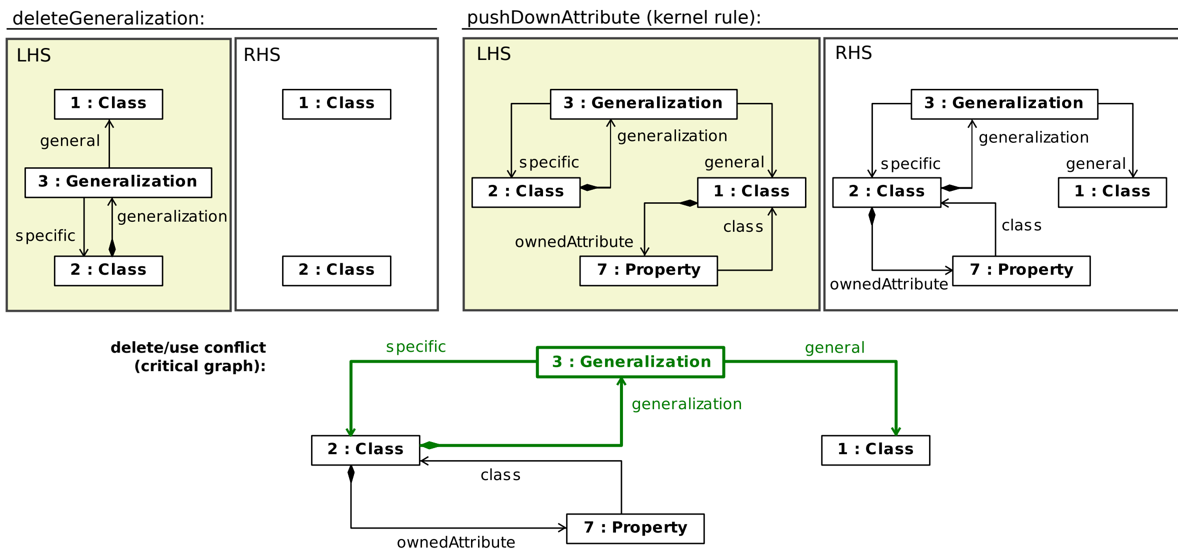


Figure 3.8: Potential delete/use conflict between edit rules deleteGeneralization and pushDownAttribute.

Example 3.6 (Potential delete/use conflict)
Figure 3.8 shows transformation rules implementing edit operations deleteGeneralization and pushDownAttribute. These rules are inverse to our edit rules of figures 3.6 and 3.7, respectively. Note that only the kernel rule of pushDownAttribute is shown in Figure 3.8. In contrast to the visual syntax offered by Henshin, left-hand side (LHS) and right-hand side (RHS) of a rule are shown separately in Figure 3.8 in order to indicate which

components have to be overlapped by the CPA (colored in light green). For each of the rules, the partial mapping between LHS and RHS is symbolized by identifiers attached to nodes, mappings between edges are given implicitly by their corresponding source and target nodes. The same identifiers are used to show the minimal critical graph in which deleteGeneralization and pushDownAttribute are conflicting. The conflict is indicated by elements in the critical graph which are colored in dark green: deleteGeneralization deletes the generalization relationship (represented by node 3 and its incident edges) between classes 2 and 1 which is required by pushDownAttribute to push down attribute 7 from class 1 to 2.

Minimal context graphs such that two rules r_1 and r_2 can be applied in a way that the application of r_2 *depends on* the application of r_1 are computed by overlapping R_1 (the right-hand side of rule r_1) with L_2 (the left-hand side of rule r_2) in all possible type- and structure-compatible ways (again, without considering application conditions). Each critical graph $G2$ together with sequentially dependent rule applications $G1 \overset{r_1, m_{G1}, n_{G2}}{\Longrightarrow} G2$ and $G2 \overset{r_2, m_{G2}, n_{G3}}{\Longrightarrow} G3$ constitutes a critical pair which represents a potential dependency between the two rules r_1 and r_2.

Example 3.7 (Potential create/use dependency)
A minimal example in which edit rules createGeneralization and pullUpAttribute may depend on one another is shown in Figure 3.9. Again, only the kernel rule of pullUpAttribute is shown here. As indicated by the minimal critical graph, the generalization relationship (represented by node 3 and its incident edges) between classes 2 and 1 has to be created by rule createGeneralization before attribute 7 can be pulled up from 2 to 1.

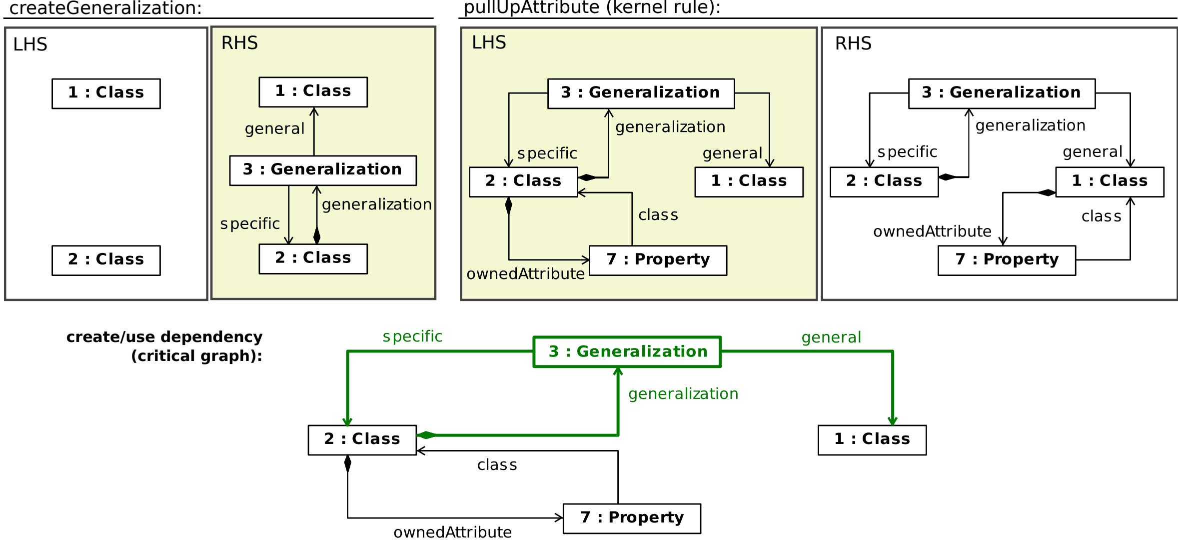


Figure 3.9: Potential create/use dependency between edit rules createGeneralization and pullUpAttribute

We will later see in Chapter 5 how we utilize the critical pair analysis in the context of edit script generation. Basically, all pairs of edit rules are statically analyzed for potential dependencies in order to reduce the set of candidate edit steps that have to be checked for actual dependencies in a model difference.

CHAPTER 4

Semantic Lifting of Model Differences

In this chapter, we introduce our approach to the semantic lifting of model differences. We begin with a motivating example which will serve as running example throughout the chapter.

Motivating example. Section 2.4 has outlined several reasons for the low quality of model differences being delivered by currently available ASG-based differencing tools. The biggest problem is that the external and internal representations of a model and related changes can differ quite substantially. Reconsider our example of Figure 2.1. If available difference tools compare versions v_1 and v_2 of our sample class diagram, they will find the following set of low-level changes as a result of the first edit step which restricts the navigability of association *worksFor* in one direction.

- Reference *ownedAttribute* from class *Person* to property *employer* has been removed.
- Reference *ownedEnd* from association *worksFor* to property *employer* has been added.
- Reference *class* from property *employer* to class *Person* has been removed.
- Reference *owningAssociation* from property *employer* to association *worksFor* has been added.

The reason why difference tools report these low-level changes can be explained by considering the UML meta-model [190]. The excerpt in Figure 3.2 shows all type definitions which are relevant for our example, namely the three meta-classes *Class*, *Association* and *Property*, together with their various relations. As we can see in Figure 3.2, properties being association ends (called *memberEnd* in the UML) may be owned by classes (via *ownedAttribute*) or associations (via *ownedEnd*). An association end is navigable if it is owned by a class[1]. To restrict the navigability of an association to one end, the corresponding edit operation restrictNavigability(p1) can be informally specified as shown in the upper part of Figure 4.1. Variable p1 serves as input parameter which shall be bound to the association end to be restricted in navigability. Variables a, p2, C1, and C2 will be automatically bound to the association, the opposite association end and the adjacent classes.

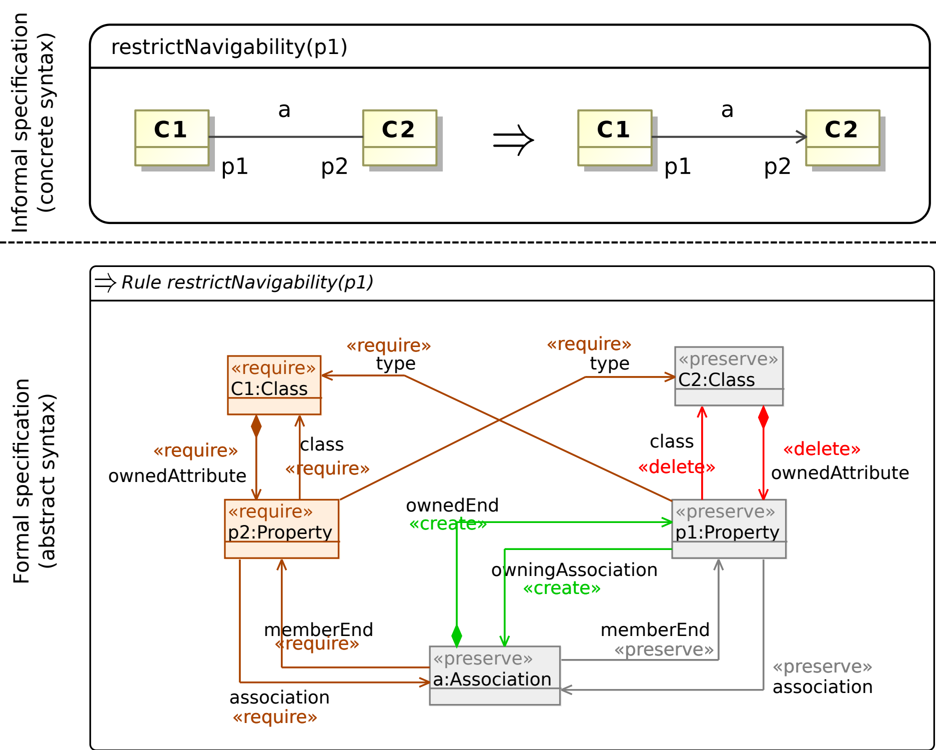


Figure 4.1: Edit operation restrictNavigability: Informal and formal specification

Although the edit operation appears to be rather simple, the necessary changes to the ASG are quite complex, as shown by edit rule restrictNavigability(p1) in the lower part of Figure 4.1. Property p1 representing the association end to be restricted in navigability, its owning class C2 as well as the association a having p1 as member end have to be found. After the rule application, the matched property p1 is no longer

[1] The UML meta-model specifies a second option how association ends can be marked as navigable. With this option, an association end is navigable if it is a *navigableOwnedEnd* which is owned by the association. In this thesis, we follow the more common interpretation and assume navigable ends to be owned by classes and non-navigable ends to be owned by associations.

connected to class C2, but belongs to the association. Note that a positive application condition (graph pattern annotated with “stereotype” *required* in Figure 4.1) ensures that there is a property p2 which represents the opposite association end of p1 and which is navigable from C1 to C2.

The effects of applying this rule to the base version v_1 of Example 2.1 correspond to the set of low-level changes listed above. Initially, property *employer* is owned by class *Person*, the ownership changes to association *worksFor*. Thus, ASG edges of type *ownedAttribute* and *class* are deleted, edges of type *ownedEnd* and *owningAssociation* are created. Obviously, these low-level changes are not understandable for normal tool users who are not familiar with meta-modeling and the related internal representation of models in terms of an ASG.

The approach in a nutshell. The example shows that a potentially large unstructured set of low-level changes should be grouped in such a way that model differences are explained in terms of user-level edit operations. We use the term *semantic lifting of differences* to refer to this transformation of low-level changes to more conceptual descriptions of model modifications. In this chapter, we present a technique for designing and implementing tool components which can semantically lift model differences. We assume the usual structure of state-based differencing algorithms as shown in Figure 1.2: Initially, a matching algorithm identifies corresponding model elements and relations in both models, i.e. corresponding nodes and edges in their ASGs. The low-level difference derived from a matching is further processed by a “semantic lifting” component which identifies sets of low-level changes (called *semantic change sets*) that implement an edit operation. In our approach, difference information is structurally represented on the level of the ASG. Thus, finding groups of related low-level changes is basically a pattern matching problem. We assume a matching engine being part of an interpreter for Henshin transformation rules to be readily available in order to solve this problem. The main task to adapt a semantic lifting component to a given modeling language is thus to “program” recognition rules which find groups of related low-level changes and which annotate these groups accordingly. In accordance with MDE principles, we automatically derive these rules from their corresponding edit rules rather than to manually program them. The effort required to configure a semantic lifting component is significantly reduced in this way.

The rest of this chapter is structured as follows. In Section 4.1 we introduce our representation of model differences which is later extended by the definition of semantic change sets. Section 4.2 presents our approach to specify instances of semantic change sets by recognition rules, while Section 4.3 explains how recognition rules are to be executed. Finally, Section 4.4 discusses restrictions of our approach.

4.1 Low-level Differences

State-based differencing of two models A and B starts by looking for the “same parts” in A and B (s. Figure 1.2). The result of this first processing step of a model differencing

pipeline is commonly referred to as *model matching*.

Definition 4.1 (Model matching)
Let A and B be models which are represented as ASGs being typed over the same type graph TG, then a matching between A and B is a partial, injective, edge- and type-preserving mapping $M_{A,B} : A \rightsquigarrow B$ identifying the nodes and edges which are considered to be "the same" in A and B. A pair of elements $(a, b) \in M_{A,B}$ is called a correspondence, the elements a and b are said to correspond to each other.

There are various methods to identify corresponding model elements, a brief overview is given in Section 2.3.1. Please note that most matching algorithms which have been specifically designed for the comparison of models deliver only correspondences between the nodes of two ASGs, their edges are not explicitly matched. In such a case, two edges can be considered as corresponding if they have the same type and if their source and target nodes are corresponding. This derivation of correspondences between edges is unique if both A and B adhere to the *no-parallel-edges* constraint (s. Section 3.2), which we assume here in any event.

Matching algorithms are largely out of the scope of this thesis. In principle, any of the available model matchers can be adopted, as long as it fulfills some basic requirements which follow directly from Definition 4.1:

- A first requirement is that the matcher must deliver *one-to-one correspondences* (matchings are injective). There are only a few model matching approaches, e.g. [152], which potentially deliver one-to-many or many-to-many correspondences and which therefore cannot be adopted.

- Secondly, due to our design decision to use Henshin as a specification language for implementing edit operations, we require a matching to be *type-preserving*, i.e. corresponding elements must have the same type. The reason behind this constraint is that Henshin rules do not allow a re-typing of graph elements.

Note that the use of consistency-preserving edit operations leads to further requirements which will be discussed in Chapter 7.

Low-level changes. Given two models A and B and a matching $M_{A,B} : A \rightsquigarrow B$, one can derive a directed delta from A to B as follows (cf. Section 2.3.2): Each element of A (or B) not involved in a correspondence leads to a change action which deletes (or creates) this element. Each non-identical attribute value of two corresponding nodes yields a change action overwriting this attribute with the value in model B. Each change action in such a directed delta corresponds to a *low-level change* which can be *observed* between the models (the modification could actually have been caused in a different way). We refer to these low-level changes and the corresponding matching as *low-level difference* $\delta_{A,B}$.

Definition 4.2 (Low-level model difference)
A low-level difference $\delta_{A,B} = (M_{A,B}, C)$ is a description of how two models A and B differ in terms of their ASGs. It consists of a matching $M_{A,B}$ and a set C of low-level changes from A to B which can be uniquely derived from $M_{A,B}$:

- For each node $n_A \in (A \setminus B)$, there is a distinct change $rmv_{n_A} \in C$ representing the deletion of n_A.
- For each node $n_B \in (B \setminus A)$, there is a distinct change $add_{n_B} \in C$ representing the creation of n_B.
- For each edge $e_A \in (A \setminus B)$, there is a distinct change $rmv_{e_A} \in C$ representing the deletion of e_A.
- For each edge $e_B \in (B \setminus A)$, there is a distinct change $add_{e_B} \in C$ representing the creation of e_B.
- For each pair of corresponding nodes $(n_A, n_B) \in M_{A,B}$ and for each pair of attributes $(n_A.a, n_B.a)$ sharing the same attribute declaration $a : DT$, there is a distinct change $avc_{n_A,n_B,a} \in C$ if and only if $n_A.a \neq n_B.a$. The change $avc_{n_A,n_B,a}$ represents the respective attribute value change.

Graph-based representation of low-level differences. Definition 4.2 leaves open how a low-level difference is represented. In order to be further processable in the model transformation system Henshin, we choose a graph-based approach for this purpose. Figure 4.2 introduces our *difference model* which defines the conceptual types and which can be formalized as a type graph in a straightforward way. Two models A and B that are being compared are conceptually represented as typed graphs over a fixed type graph derived from the common meta-model of A and B. Their difference consists of a set of correspondences representing the common parts of A and B, and a set of changes from model A to model B. A correspondence links a node of model A to a node of model B (s. invariant `iv1` in Figure 4.2). Correspondences between edges are given implicitly by their corresponding source and target nodes. We distinguish the following types of changes:

- A change of type *AddNode* represents the insertion of a new node, i.e. a node that is contained in model B but does not have a corresponding node in model A (s. invariant `iv2`). Analogously, changes of type *AddEdge* represent edges that have been inserted (s. invariant `iv4`).
- Changes of types *RemoveNode* and *RemoveEdge* represent the inverse of changes of types *AddNode* and *AddEdge*, respectively.
- An *AttributeValueChange* represents a value change of an attribute of corresponding nodes (s. invariant `iv6`).

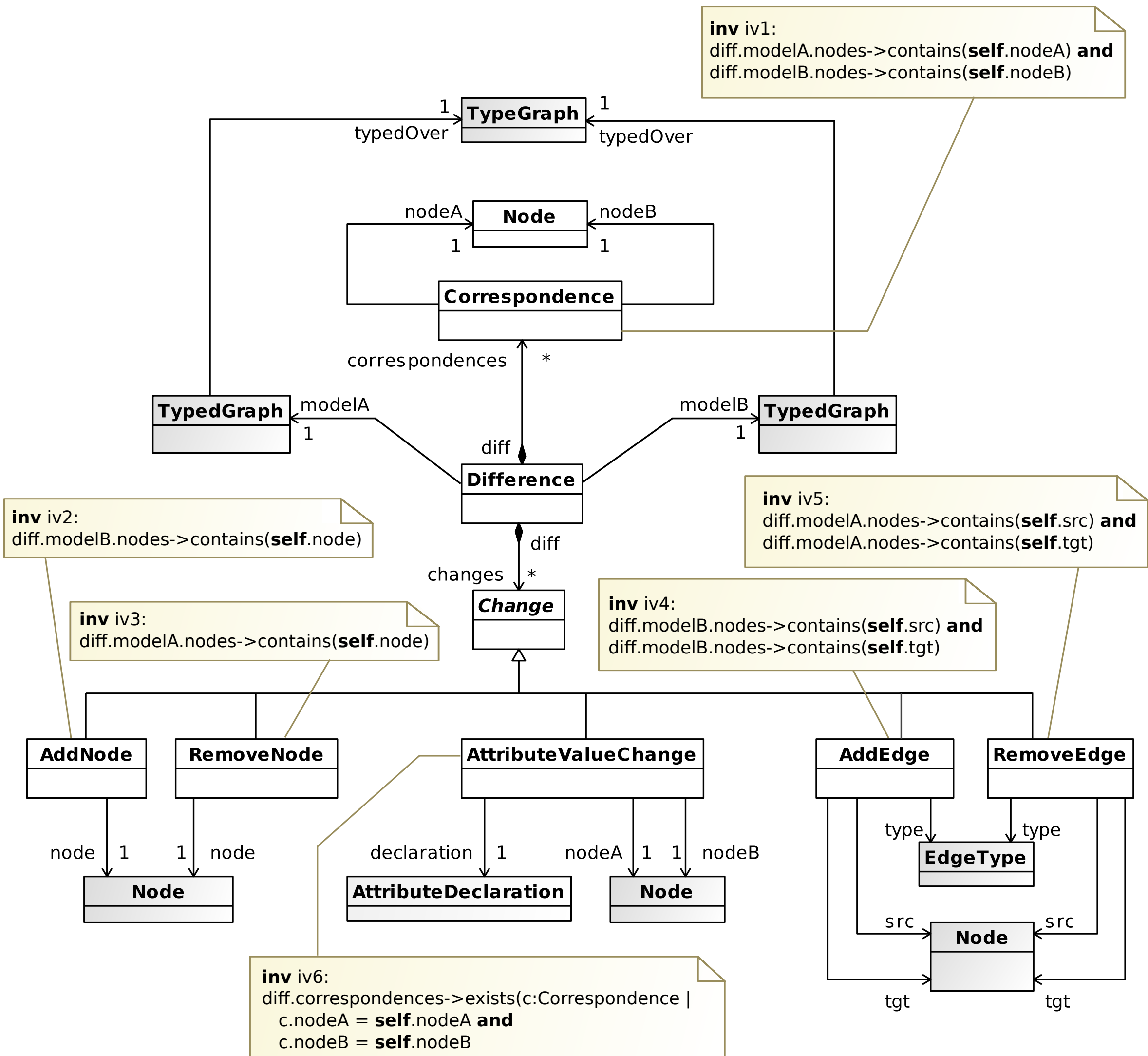


Figure 4.2: Graph-based representation of low-level differences

We introduce the following notation to describe a change c in a compact manner:

$$c = \langle changeType, context \rangle \tag{4.1.1}$$

The *changeType* denotes the type of change, i.e. the concrete subclass of *Change*, while the *context* denotes the set of ASG elements to which the change has been applied.

Example 4.1 (Low-level Changes)
In terms of our running example of Figure 2.1, restricting the navigability of association end *employer* leads to the following four low-level changes which were already introduced informally in the introductory part of this chapter.

$$\begin{aligned}
\text{rn1} &= \langle\text{"}RemoveEdge\text{"}, \\
&\quad \{src = \text{"}Person\text{"}, tgt = \text{"}employer\text{"}, \ type = \text{"}ownedAttribute\text{"}\}\rangle \\
\text{rn2} &= \langle\text{"}AddEdge\text{"}, \\
&\quad \{src = \text{"}worksFor\text{"}, tgt = \text{"}employer\text{"}, \ type = \text{"}ownedEnd\text{"}\}\rangle \\
\text{rn3} &= \langle\text{"}RemoveEdge\text{"}, \\
&\quad \{src = \text{"}employer\text{"}, tgt = \text{"}Person\text{"}, \ type = \text{"}class\text{"}\}\rangle \\
\text{rn4} &= \langle\text{"}AddEdge\text{"}, \\
&\quad \{src = \text{"}employer\text{"}, tgt = \text{"}worksFor\text{"}, \ type = \text{"}owningAssociation\text{"}\}\rangle
\end{aligned}$$

Changes $rn1$ and $rn2$ represent the “move” of property *employer* from class *Person* to association *worksFor*. Changes $rn3$ and $rn4$ are typical examples of conceptually irrelevant pseudo changes [140]; $rn3$ results from the fact that the edge of type *class* is a redundant information here. Change $rn4$ occurs since edge types *ownedEnd* and *owningAssociation* are defined as opposite to each other (s. Figure 3.2).

Let us now consider the second edit step of our Example 2.1. Here, the creation of a generalization relationship between classes *Developer* and *Person* leads to low-level changes $cgd1$, $cgd2$ and $cgd3$, which can be easily derived from the ASG representation shown in Figure 3.3. Note that we use the notation *Developer* → *Person* as a symbolic reference to the “anonymous” node of type *Generalization*. Again, the pseudo change which represents the creation of edge *generalization* being opposite to the edge of type *specific* is deliberately omitted here.

$$\begin{aligned}
\text{cgd1} &= \langle\text{"}AddNode\text{"}, \\
&\quad \{obj = \text{"}Developer \rightarrow Person\text{"}\}\rangle \\
\text{cgd2} &= \langle\text{"}AddEdge\text{"}, \\
&\quad \{src = \text{"}Developer \rightarrow Person\text{"}, tgt = \text{"}Person\text{"}, \ type = \text{"}general\text{"}\}\rangle \\
\text{cgd3} &= \langle\text{"}AddEdge\text{"}, \\
&\quad \{src = \text{"}Developer\text{"}, tgt = \text{"}Developer \rightarrow Person\text{"}, \ type = \text{"}specific\text{"}\}\rangle
\end{aligned}$$

Analogously to $cgd1$, $cgd2$ and $cgd3$, we obtain low-level changes $cgm1$, $cgm2$ and $cgm3$ for the third edit step of Example 2.1, the creation of a generalization relationship between classes *Manager* and *Person*.

$$\begin{aligned}
\text{cgm1} &= \langle\text{"}AddNode\text{"}, \\
&\quad \{obj = \text{"}Manager \rightarrow Person\text{"}\}\rangle \\
\text{cgm2} &= \langle\text{"}AddEdge\text{"}, \\
&\quad \{src = \text{"}Manager \rightarrow Person\text{"}, tgt = \text{"}Person\text{"}, \ type = \text{"}general\text{"}\}\rangle \\
\text{cgm3} &= \langle\text{"}AddEdge\text{"}, \\
&\quad \{src = \text{"}Manager\text{"}, tgt = \text{"}Manager \rightarrow Person\text{"}, \ type = \text{"}specific\text{"}\}\rangle
\end{aligned}$$

Finally, let us consider the fourth edit step of Example 2.1. Here, the attribute *name* is moved from subclasses *Developer* and *Manager* to their superclass *Person* by means of the well-known refactoring operation pullUpAttribute [107]. Let us assume that our matching contains correspondences between all classes in v_1 and v_2 having equal names, and a correspondence between the attributes *Manager.name* in v_1 and *Person.name* in

v_2. The following low-level changes are derived then. We omit several low-level changes which are pseudo changes. They are similar to those which occurred in the previous edit steps.

$$\begin{aligned}
\text{pua1} &= \langle \text{"}RemoveNode\text{"}, \\
&\quad \{obj = \text{"}Developer.name\text{"}\}\rangle \\
\text{pua2} &= \langle \text{"}RemoveEdge\text{"}, \\
&\quad \{src = \text{"}Manager\text{"}, tgt = \text{"}Manager.name\text{"}, \ type = \text{"}ownedAttribute\text{"}\}\rangle \\
\text{pua3} &= \langle \text{"}AddEdge\text{"}, \\
&\quad \{src = \text{"}Person\text{"}, tgt = \text{"}Person.name\text{"}, \ type = \text{"}ownedAttribute\text{"}\}\rangle
\end{aligned}$$

Change $pua1$ represents the deletion of the attribute *Developer.name*, while changes $pua2$ and $pua3$ represent the move of attribute *Manager.name* from class *Manager* to class *Person*.

4.2 Semantic Change Sets

Our example has already shown that user-level edit operations often lead to many low-level changes which are hard to understand for normal users. This problem is further aggravated by the fact that the set of low-level changes can be listed in an arbitrary order and that low-level changes stemming from different edit operations can be mixed randomly. The objective of semantically lifting a model difference is thus to partition the set of low-level changes into subsets, each subset containing the changes belonging to exactly one application of an edit operation. We call these subsets *semantic change sets*. A difference in which the set of low-level changes is partitioned into semantic change sets is called a *semantically lifted difference*.

Definition 4.3 (Semantically lifted difference and semantic change sets)
A semantically lifted difference $\Delta_{A,B} = (\delta_{A,B}, CS)$ is a description of how two models A and B differ in terms of semantic change sets. It consists of a low-level difference $\delta_{A,B} = (M_{A,B}, C)$ and a set of semantic change sets $CS = \{cs_1, ..., cs_n\}$ with $n \in \mathbb{N}$ and the following properties:

- $n \leq |C|$, where $|C|$ is the overall number of low-level changes,
- $cs_i \subseteq C$ for all i, $1 \leq i \leq n$,
- $\bigcup_{i=1}^{n} cs_i = C$, and
- $cs_i \cap cs_j = \emptyset$ for all $1 \leq i < j \leq n$.

Example 4.2 (Semantic change sets)
In our example of Figure 2.1, the set of all change sets is

$$\begin{aligned} CS_1 = \; & \{\{rn1, rn2, rn3, rn4\}, \\ & \{cgd1, cgd2, cgd3\}, \\ & \{cgm1, cgm2, cgm3\}, \\ & \{pua1, pua2, pua3\}\}. \end{aligned}$$

The first change set contains the result of the edit operation restrictNavigability, the second and third change set are results of edit operation createGeneralization in different contexts, and the last one contains the result of edit operation pullUpAttribute. As already mentioned, we omit several pseudo changes here for the sake of readability.

Representation of semantic change sets. In order to represent semantic change sets, we must extend our difference model of Figure 4.2. These extensions are shown in Figure 4.3. Nodes of type *Change*, which represent the low-level changes, are grouped by *SemanticChangeSet* nodes, each of these nodes represents the effect of an edit operation application. The name of a change set corresponds to the name of the respective edit operation.

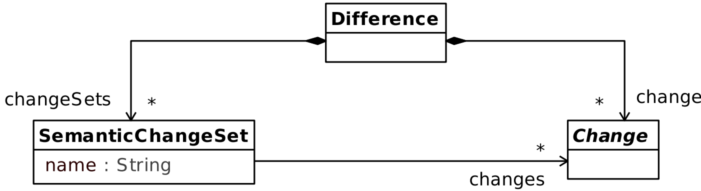


Figure 4.3: Difference model extension: Representation of semantic change sets

4.2.1 Rule-based Specification of Semantic Change Sets

The application of a specific type of edit operation results in a change pattern on our graph-based difference representation which is characteristic of this type of edit operation. Thus, we can make use of Henshin transformation rules

1. to specify change patterns that have to be recognized in a graph-based difference representation and

2. to specify how to group low-level differences contained in a change pattern.

A transformation rule which handles a specific type of edit operation is referred to as *change set recognition rule*, or recognition rule, for short. Basically, the LHS of a recognition rule specifies an instance of a change pattern which is characteristic of the handled type of edit operation (1.). The RHS is responsible for extending the difference with a semantic change set node which groups the low-level changes related to the change pattern (2.).

Example 4.3 (Change set recognition rule rr-restrictNavigability)
As an example, an excerpt of the recognition rule rr-restrictNavigability recognizing edit operation restrictNavigability is depicted in Figure 4.4. Although the concrete identifier names are technically irrelevant here, we use the variable names of Figure 4.1 to build identifier names. That way, rule graph elements in rr-restrictNavigability can be mentally related to those of the corresponding edit rule restrictNavigability.

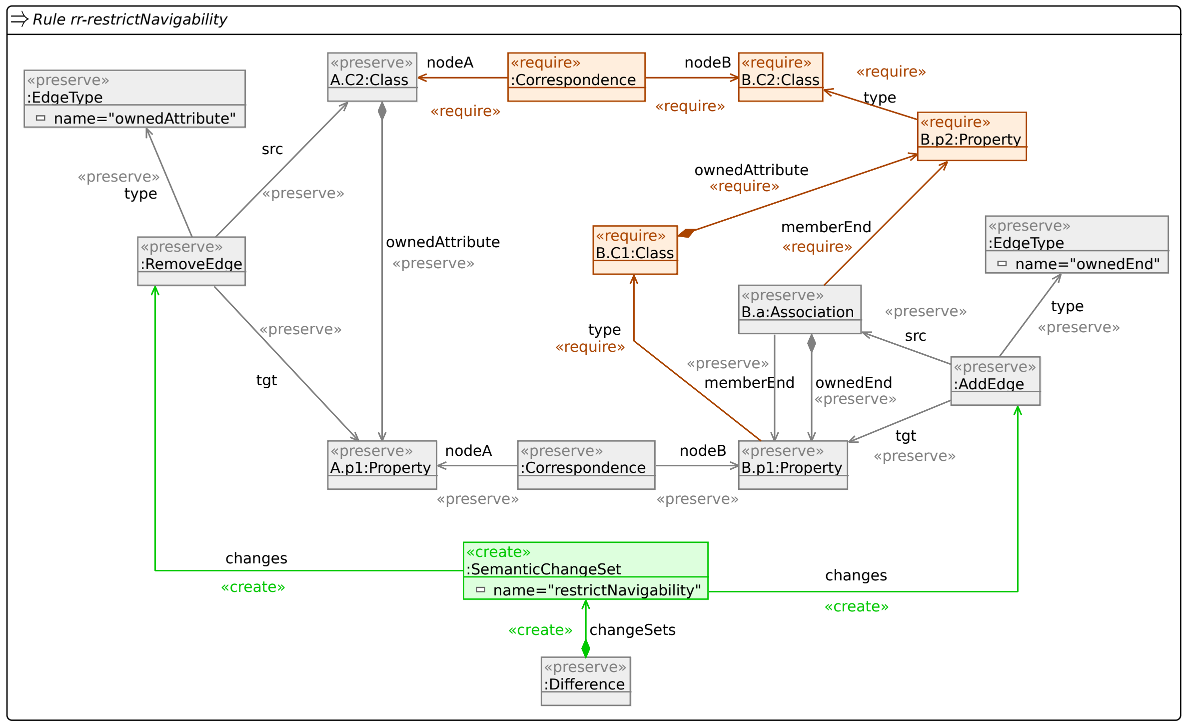


Figure 4.4: Recognition rule rr-restrictNavigability

Our sample recognition rule rr-restrictNavigability contains four change patterns (two *remove edge patterns* and two *add edge patterns*) representing changes $rn1$ to $rn4$ (s. Example 4.1). Figure 4.4 shows patterns for changes $rn1$ and $rn2$ only. They can be easily identified in Figure 4.4 by means of their respective change types and contexts.

A *SemanticChangeSet* node is created by each application of rr-restrictNavigability; this node groups all related low-level changes. Additionally, the created change set will be inserted into the difference representation.

Certain elements being part of contexts of low-level changes, notably elements which are to be preserved by an edit rule, are searched on the original model version A *and* the revised version B. As these elements are to be considered the same in A and B, they are linked by correspondences in the difference representation. As we can see in Figure 4.4, an instance of such a *correspondence pattern* occurs for property p1 which is moved from association a to class C2. Please note that, according to our difference representation introduced in Figure 4.2, no direct correspondence mappings are established between preserved edges in the difference representation. Corresponding

edges are identified by their context, i.e. corresponding source and target nodes. In Figure 4.4, we generally omitted edges of type *owningAssociation*, *association* and *class* for the sake of readability.

As we can see in Figure 4.4, application conditions specified by an edit rule are also translated to the corresponding recognition rule. Application conditions that are to be interpreted as precondition must be fulfilled on model version A, while application conditions that are to be interpreted as postcondition of an edit rule must be fulfilled on model version B. As explained in Section 3.3.2, certain application conditions can be checked before or after the application of an edit rule; this is possible, if the boundary elements of the application condition are preserved by the rule. An application condition which can be interpreted as pre- *and* postcondition can be checked by a recognition rule in three different ways: It can be checked to be fulfilled *i)* on the original model version A, *ii)* on the revised model version B, or *iii)* on both model versions. In such a case, a concrete interpretation has to be chosen which meets the desired user preferences. In our example, the application condition is to be interpreted as postcondition; a navigability restriction in one direction, i.e. for an association end `p1` at class `C1`, shall only be detected if the opposite end `p2` at class `C2` is still navigable.

4.2.2 Generation of Recognition Rules

Change set recognition rules are getting complex very quickly. However, they are very schematic and can be automatically generated from their corresponding edit rule. The basic principle of this transformation is presented as function EDITR2RECOGNR in Algorithm 1. Let *er* be the edit rule which serves as input parameter and which will be transformed to a corresponding recognition rule *rr* (s. line 2 in Algorithm 1). The name of *rr* corresponds to that of *er* supplemented with a prefix “rr-” (line 3).

The further translation proceeds in three steps (lines 5-7): First, the *context* of the edit rule is translated. Next, all *application conditions* are translated to application conditions being checked by the recognition rule. Finally, *change actions* are translated such that low-level change patterns are created for each change action, and low-level changes are grouped to semantic change sets.

Algorithm 1 Edit rule to recognition rule translation (main program)

```
1: function EDITR2RECOGNR(Rule er) : Rule
2:     Rule rr = new Rule();
3:     rr.name = “rr-”∘ er.name;
4:     ▷ Delegate further translation to subroutines
5:     TRANSLATECONTEXT(er, rr);
6:     TRANSLATEAPPLICATIONCONDITIONS(er, rr);
7:     TRANSLATECHANGEACTIONS(er, rr);
8:     return rr;
9: end function
```

Context translation. The translation of the context of an edit rule *er* to a recognition rule *rr* is illustrated by function TRANSLATECONTEXT(Rule *er*, Rule *rr*) (s. Algorithm 2). We start with the translation of the nodes which are to be preserved by *er* and which are boundary nodes to edge deletion or creation actions:

(lines 5-7): Boundary nodes to edge deletions must occur in model version A, but not necessarily in version B. Thus, we search for them in model version A.

(lines 8-10): Boundary nodes to edge creations must occur in model version B, but not necessarily in version A. Thus, we search for them in model version B.

(lines 11-13): Preserved nodes which are boundary to edge deletions and edge creations must occur in model versions A and B and have to be connected by a correspondence link in our difference representation.

Nodes which are to be preserved by *er* and which are neither boundary to edge deletions nor to edge creations can be conceptually treated like nodes of a positive application condition. Therefore, the translation proceeds similar to PAC nodes as explained below and is not explicitly shown in Algorithm 2. The same applies to context edges, i.e. edges which are to be preserved by *er*; they can be conceptually treated like PAC edges as explained below.

Algorithm 2 Context translation

```
 1: function translateContext(Rule er, Rule rr)
 2:     ▷ Translation of context nodes
 3:     for all Node n_er ∈ Cxt_er do
 4:         Node n_A, n_B;
 5:         if n_er ∈ B_Del_er then                    ▷ n_er must occur in model version A
 6:             n_A = translateToCxt(rr, "A", n_er)
 7:         end if
 8:         if n_er ∈ B_Cre_er then                    ▷ n_er must occur in model version B
 9:             n_B = translateToCxt(rr, "B", n_er)
10:         end if
11:         if n_er ∈ B_Del_er ∧ n_er ∈ B_Cre_er then  ▷ n_er must occur in versions A and B
12:             createCorrespondencePattern(n_A, n_B)
13:         end if
14:     end for
15:     return ;
16: end function
```

Note that subroutines TRANSLATETOCXT(Rule *rr*, String *version*, Node n_{er}) and CREATECORRESPONDENCEPATTERN(Node n_{rr_A}, Node n_{rr_B}) are not explained in detail here but can be implemented straightforward according to the explanation of the example recognition rule of Figure 4.4.

Application condition translation. Application conditions of edit rule *er* are translated to application conditions being checked by the recognition rule *rr* as illustrated by function TRANSLATEAPPLICATIONCONDITIONS(Rule *er*, Rule *rr*) of Algorithm 3. For each application condition c_{er} of *er* we first create an empty application

condition c_{rr} in rr (line 3). Next, we translate each node of the edit rule application condition to a corresponding node of the recognition rule application condition (lines 5-7).

Algorithm 3 Translation of application conditions

```
1: function TRANSLATEAPPLICATIONCONDITIONS(Rule er, Rule rr)
2:     for all ApplicationCondition c_er ∈ PAC_er ∪ NAC_er do
3:         ApplicationCondition c_rr = new ApplicationCondition(rr);
4:         ▷ Translate "normal" nodes
5:         for all Node n ∈ c_er do
6:             TRANSLATETOAC(c_rr, n);
7:         end for
8:         ▷ Create "missing" boundary nodes
9:         for all Node b ∈ B_c do
10:            if ISTOBEFULLFILLEDONA(c_er) and ¬(b ∈ B_Del_er) then
11:                Node n_B = TRACEB(b);
12:                Node n_A = TRANSLATETOAC(c_rr, b);
13:                CREATECORRESPONDENCEPATTERN(n_A, n_B);
14:            end if
15:        end for
16:        for all Node b ∈ B_c do
17:            if ISTOBEFULLFILLEDONB(c_er) and ¬(b ∈ B_Cre_er) then
18:                Node n_A = TRACEA(b);
19:                Node n_B = TRANSLATETOAC(c_rr, b);
20:                CREATECORRESPONDENCEPATTERN(n_A, n_B);
21:            end if
22:        end for
23:        ▷ Translate edges
24:        for all Edge e ∈ B_c do
25:            translateToAC(c_rr,e)
26:        end for
27:    end for
28:    return ;
29: end function
```

Further on, we check whether nodes of c_{rr} have to be created in the case of "missing" boundary nodes: Given a boundary node $b \in B_{c_{er}}$ for the edit rule application condition c_{er}, the corresponding boundary node for c_{rr} is missing if the condition must be fulfilled on model version A but the node b is not searched in version A (line 10). Obviously, the translated node n_B, which is obtained via subroutine called TRACEB in Algorithm 3 (s. line 11), cannot serve as boundary node for the recognition rule application condition in this case because it is to be searched in model version B. Instead, c_{rr} is extended by translating b to node $n_A \in c_{rr}$ (line 12), and linking the created node n_A via a correspondence pattern with n_B (line 13). Analogously, a boundary node $b \in B_{c_{er}}$ is also missing if the application condition must be fulfilled on model version B but node b is not searched in this version (line 17). In this case, the translated application condition c_{rr} is extended (lines 18-20) according to the same principle as for missing boundary nodes in A. Finally, we translate each edge of the edit rule application condition to a corresponding edge of the recognition rule application condition in a straightforward

way (lines 24-26).

All subroutines used in Algorithm 3 are not discussed here in further detail. Subroutine TRANSLATETOAC(ApplicationCondition c_{rr}, Node n_{er}) is basically a simple copy operation, similar to TRANSLATETOCXT used in Algorithm 2. The utility functions TRACEA(Node n_{er}) and TRACEB(Node n_{er}) require that trace links between edit rule and recognition rule elements are explicitly managed (s. Section 5.2); the implementation of the look-up procedure is straightforward. The interpretation of an application condition is obtained by utility functions ISTOBEFULLFILLEDONA(ApplicationCondition c_{er}) and ISTOBEFULLFILLEDONB(ApplicationCondition c_{er}). In our implementation, this interpretation is attached as annotation to the respective edit rule application condition (s. Section 8.3.3).

Algorithm 4 Translation of change actions

```
 1: function translateChangeActions(Rule er, Rule rr)
 2:     Node cs = new Node(Cre_rr, "SemanticChangeSet");
 3:     cs.name = er.name;
 4:     ▷ Create low-level change patterns and specify grouping of changes
 5:     for all Node n ∈ Del_er do
 6:         RemoveNodePattern p = createRemoveNodePattern(rr, n);
 7:         linkChangeToChangeSet(p, cs);
 8:     end for
 9:     for all Edge e ∈ Del_er do
10:         RemoveEdgePattern p = createRemoveEdgePattern(rr, e);
11:         linkChangeToChangeSet(p, cs);
12:     end for
13:     for all Node n ∈ Cre_er do
14:         AddNodePattern p = createAddNodePattern(rr, n);
15:         linkChangeToChangeSet(p, cs);
16:     end for
17:     for all Edge e ∈ Cre_er do
18:         AddEdgePattern p = createAddEdgePattern(rr, e);
19:         linkChangeToChangeSet(p, cs);
20:     end for
21:     for all Node n ∈ Cxt_er do
22:         for all Attribute a of n do
23:             if n_L.a ≠ n_R.a then
24:                 AttributeValueChangePattern p = createAttributeValueChangePattern(rr,
    e);
25:                 linkChangeToChangeSet(p, cs);
26:             end if
27:         end for
28:     end for
29:     return ;
30: end function
```

Translation of change actions. The translation of change actions of an edit rule er to a recognition rule rr is illustrated by function TRANSLATECHANGEACTIONS

(s. Algorithm 4). We begin with the creation of a Node $cs \in Cre_{rr}$ representing the semantic change set which is to be created by rr (lines 2-3)

Subsequently, a low-level change pattern has to be found by a recognition rule for each change action which is defined by the corresponding edit rule. All subroutines creating instances of the respective low-level change patterns are not explained in detail here. They can be implemented straightforward according to the explanation of Example 4.3. Change patterns which do not occur at all in our example can be easily derived from the difference model of Figure 4.2. In case of an attribute value change action (s. lines 21-28), n_L and n_R refer to LHS and RHS representations of a context node n.

For each change action being translated, the *SemanticChangeSet* node *cs* is linked with all *Change* nodes being part of change pattern instances; these links are created by the subroutine LINKCHANGETOCHANGESET(ChangePattern p, Node *cs*).

4.2.3 Non-static Change Patterns

So far, we have considered edit operations which result in *static change patterns*, i.e. each edit operation can be specified by one transformation rule. However, there are also edit operations resulting in *non-static change patterns*. The refactoring operation pullUpAttribute is an example of this; it is applied to all common attributes in the set of all direct subclasses of a given class. Since the number of common attributes and the number of subclasses can vary, we need a concept to capture this variability.

Analogously to the implementation of multi-object structures in edit rules (s. Section 3.3.2), we adopt the concept of amalgamation to handle non-static change patterns in recognition rules. To that end, our procedure for recognition rule generation is extended by the function EDITRS2RECOGNRS(RuleScheme ers). A sketch of this function is listed in Algorithm 5. Parameter ers denotes the editing rule scheme and variable rrs denotes the recognition rule scheme which is to be created. The kernel rule of ers is transformed to the kernel rule of rrs according to the function EDITR2RECOGNR introduced in Section 4.2.2 (s. line 4). In the same way, each multi-rule of ers is transformed to a multi-rule of rrs properly embedding the kernel rule of rrs into each multi-rule (lines 5-8).

Algorithm 5 Translation of rule schemes

```
 1: function editRS2RecognRS(RuleScheme ers) : RuleScheme
 2:     RuleScheme rrs = new RuleScheme();
 3:     rrs.name = "rr-"∘ ers.name;
 4:     rrs.kernel = editR2RecognR(ers.kernel);
 5:     for all Rule em ∈ ers.multi do
 6:         Rule rm = editR2RecognR(em);
 7:         add(rrs.multi, rm);
 8:         embedKernel(rrs, rm, em);
 9:     end for
10:     return rrs;
11: end function
```

4.3 Edit Operation Recognition

Section 4.2 introduced our notion of a semantic change set and how change set recognition rules are derived from their corresponding edit rules. In this section, we explain how recognition rules are applied to a given low-level difference.

4.3.1 Rule Application Strategy

It is easy to see that all pairs of change set recognition rules are both sequentially and parallel independent: On the one hand, the change pattern that has to be found by a recognition rule, i.e. its left-hand side as well as sets of positive and negative application conditions, refers to the graph-based representation of a low-level difference which is to be semantically lifted. On the other hand, this low-level difference representation is not modified by a recognition rule which, if being applied successfully, only creates additional semantic change sets and leaves the low-level difference representation untouched. Consequently, the sequential order in which we apply the change set recognition rules is irrelevant. Additionally, recognition rules can be applied to a given difference representation at all possible matches in parallel.

However, the above rule application strategy can lead to too many change sets, i.e. there can be false positives. As an example, we extend our set of edit operations defined for UML class models by two further edit operations, namely the edit operation deleteAttribute, which removes an attribute from a class, and the edit operation moveAttribute, which moves an attribute from one class into another. In our running example of Figure 2.1, the recognition rules for deleteAttribute and moveAttribute will find the additional potential change sets $\{pua1\}$ and $\{pua2, pua3\}$, respectively (s. Example 4.1). In sum, all five recognition rules together will find the following change sets:

$$PCS_1 = CS_1 \ \cup \ \{\{pua1\}, \{pua2, pua3\}\},$$

CS_1 (s. Example 4.2) contains the change sets which represent the edit operations that have been actually applied. The change sets contained in PCS_1 are not mutually disjoint, therefore some of them must be discarded.

In general, let $\delta_{A,B} = (M_{A,B}, C)$ be a given low-level difference according to Definition 4.2, and PCS_C be the set of *potential change sets* which are created by applying all recognition rules on $\delta_{A,B}$ as often as possible. Thus, we have

$$\forall p \in PCS_C : \ p \subseteq C.$$

If some potential change sets overlap, i.e.

$$\exists p, q \in PCS_C : \ p \cap q \neq \emptyset,$$

then PCS_C must be postprocessed as described in the next section.

4.3.2 Postprocessing

The goal of the postprocessing phase is to determine a subset of set PCS_C of potential change sets which conforms to the conditions for sets of semantic change sets (s. Definition 4.3). The postprocessing of PCS_C results in a set partitioning problem, which is basically an optimization problem. Due to the lack of a clear optimization criterion [201], we assume that reporting a minimal number of edit steps reflects the user perception of a model difference most adequately. Thus, we are looking for a set $PCS_{min} \subseteq PCS_C$ of potential change sets such that the following conditions hold for $PCS_{min} = \{p_1, ..., p_k\}$:

- $p_i \cap p_j = \emptyset$ for all $1 \leq i < j \leq k$,
- $\bigcup_{i=1}^{k} p_i = C$, and
- k is minimal.

We employ the following heuristics in order to efficiently reduce PCS_C and finally determine PCS_{min}:

Firstly, we are looking for change sets that do not overlap with other change sets. Such change sets are very frequent. They must obviously be included in PCS_{min} and do not have to be dealt with further. In our example, this is the case for change set $\{rn1, rn2, rn3, rn4\}$.

Secondly, we search for each change set p which properly includes smaller change sets $q_1 \dots q_m$ which do not overlap with any other change set not included in p. In this case, change sets $q_1 \dots q_m$ will be discarded and are not included in PCS_{min} because, generally, an edit operation which covers a larger change set is preferred over edit operations which cover a smaller, included set of changes. In our running example, the change set $\{pua1, pua2, pua3\}$ representing the "pullUpAttribute" refactoring will thus be preferred over the smaller change sets $\{pua1\}$ and $\{pua2, pua3\}$, which can be discarded. These cases occur whenever a single operation is composed of a set of smaller operations or a core operation has one or more extensions.

In the special case where two change sets p and q are identical in the sense that they cover the same low-level changes, we prefer the change set representing the invocation of the "more specific" edit operation. An edit rule r_1 is considered to be more specific than edit rule r_2 if

1. the number of application conditions specified by r_1 is higher than the number of application conditions defined by r_2,
2. the types of rule nodes of r_1 are more specific than types of rule nodes of r_2 (we employ the well-known object-oriented metric "Depth in Inheritance Tree" (DIT) [71] and summarize DIT-values for the types of all rule nodes of r_1 and r_2, respectively),
3. the number of formal parameters specified by r_1 is higher than the number of formal parameters defined by r_2.

Note that conditions 1-3 are tested in sequential order, each condition is tested in mutual direction (r_1 against r_2 and vice versa). If none of the conditions evaluates to *true*, one of the change sets p, q will be selected randomly. This default processing can be overridden by manually assigning priority values to edit rules (s. Section 8.3.3).

Finally, the remaining change sets are partially overlapping. This reduced set partitioning problem has to be solved by combinatorial optimization. Our practical evaluation has shown that these cases are hypothetically as long as the set of available edit operations consists of elementary operations as they are offered by typical editors for graphical modeling.

4.4 Restrictions of the Approach

The information which can be exploited by our approach to operation detection is "restricted" to the information which is provided by a low-level difference, namely the model states A and B and the low-level changes which are observable based on these model versions and a given matching (s. Section 4.1). Thus, some editing effects may be transient in the sense that they are no longer visible at the end of a sequence of edit steps. Section 4.4.1 clarifies our notion of a *transient effect* and the resulting restrictions w.r.t. to the recognition of edit operations. Section 4.4.2 discusses the practical impact of transient effects.

4.4.1 Transient Effects

If a sequence of operation invocations was applied to a model and one operation invocation removes effects of an earlier one, we can observe two types of phenomena: Firstly, this effect does not appear in terms of low-level changes in a difference at all. Secondly, this transient effect leads to transient elements, i.e. nodes and edges of an ASG, that are neither contained in the original model version A, nor in the revised model version B. We can further distinguish two kinds of transient elements:

- **Positive transient elements** are created by one edit step and deleted in a subsequent edit step of an editing process.
- **Negative transient elements** are deleted by one edit step and re-created in a subsequent edit step.

Definition 4.4 (Transient effects)
A sequential pair of rule applications $t_1 = M1 \overset{r_1,m_1,n_1}{\Longrightarrow} M2$ and $t_2 = M2 \overset{r_2,m_2,n_2}{\Longrightarrow} M3$ leads to positive transient elements if $n_1(Cre_{r_1}) \cap m_2(Del_{r_2}) \neq \emptyset$. The sequence $t_1; t_2$ leads to negative transient elements if $m_1(Del_{r_1}) \cap n_2(Cre_{r_2}) \neq \emptyset$. Effects of sequential rule applications leading to transient elements are called *transient effects*.

Consequences. Transient effects lead to rule applications which cannot be detected by our approach to operation recognition. Consider a sequence of edit rule applications $t_1 = M1 \overset{r_1,m_1,n_1}{\Longrightarrow} M2$, $t_2 = M2 \overset{r_2,m_2,n_2}{\Longrightarrow} M3$ and $t_3 = M3 \overset{r_3,m_3,n_3}{\Longrightarrow} M4$. Let us assume that an effect of t_1 causes an application condition of t_2 to be fulfilled, while this effect is later on removed by rule application t_3. In such a case, none of the three rule applications t_1, t_2 and t_3 will be recognized by our operation detection approach for two reasons:

1. **Unobservable low-level changes:** The rule applications t_1 and t_3 can not be recognized because a recognition rule finds a match only if the difference contains *all* low-level changes which are specified as change actions by the corresponding edit rule. This restriction caused by unobservable low-level changes occurs regardless of whether the transient effect causes positive or negative transient elements.

2. **Transiently fulfilled application conditions:** Rule application t_2 can not be recognized because one of its application conditions is only transiently fulfilled. A recognition rule, however, finds a match only if *all* application conditions are fulfilled either on the original model version A, or on the revised version B; application conditions that are to be interpreted as preconditions must be fulfilled on model version A, while application conditions that are to be interpreted as postconditions must be fulfilled on model version B (s. Section 4.2).

   If t_1 and t_3 cause positive transient elements that are required to extend m_2 such that a $pac \in PAC_{r2}$ can be matched, then this pac is only transiently fulfilled. If t_1 and t_3 cause negative transient elements which must not occur in M1 such that a $nac \in NAC_{t2}$ is fulfilled at match m_2, then this nac is only transiently fulfilled.

4.4.2 Discussion

Transient effects which are not observable in a difference are welcome when an operation invocation is taken back completely, e.g., when a typical *undo* command was executed within an editing environment.

It is not desired, and not harmful either, if complex operation invocations interfere; in such situations, the compound effect can typically be represented by an alternative difference, notably a difference using elementary edit operations. Assume, for example, a three level inheritance hierarchy as shown in Figure 4.5). Starting from the base version v_1, we first pull up attribute *name* from class *Developer* to class *Person* (as in our running example). Next, the attribute is pulled up from *Person* to *Entity*. The second edit step leads to a transient effect: The creation of the containment edge from class *Person* to attribute *name* is not observable in the low-level difference δ_{v_1,v_2}. Thus, none of the two refactorings can be recognized. However, the same effect can also be described by one single attribute movement which directly shifts the attribute *name* from class *Developer* to class *Entity*.

From an editing point of view, we call such a transient effect *avoidable*, because the same effect can be reached without causing transient elements. From an operation

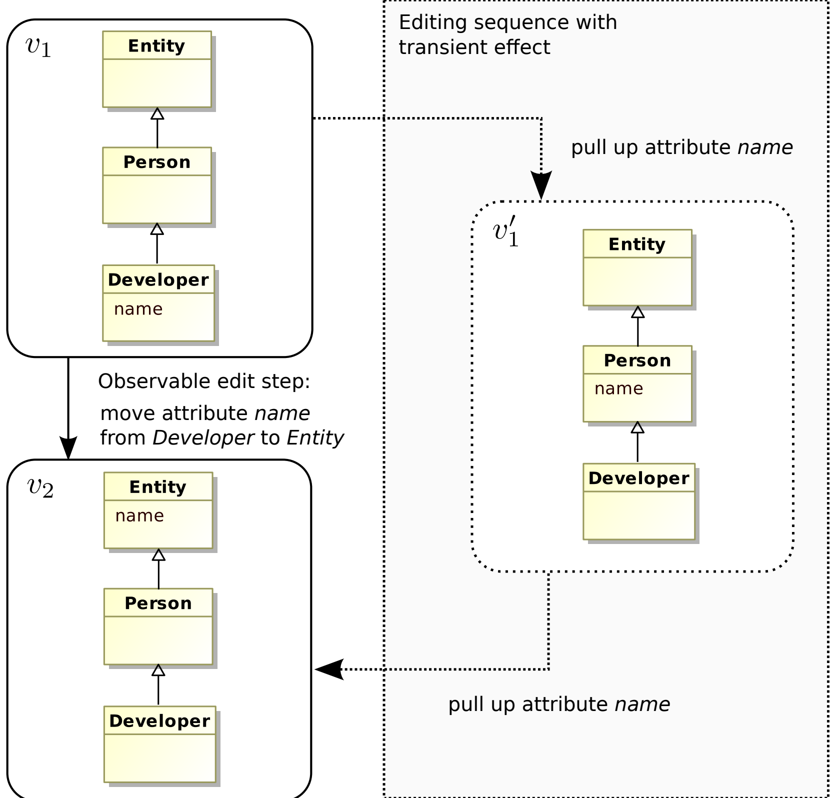


Figure 4.5: Example of a transient effect leading to unobservable low-level changes (right) and how this transient effect can be avoided (left)

recognition point of view, we call the transient effect *erasable*.

Transiently fulfilled application conditions are usually a good hint to not report the respective operation invocation which is missed due to the transient effect. Consider the example shown in Figure 4.6. On the right, we can see an editing sequence which, starting from the base model v_1, yields model version v_2 and which causes transiently fulfilled application conditions. Firstly, we introduce generalization relationships such that class *Person* becomes the common superclass of *Developer* and *Manager*. Secondly, the common attribute *name* is pulled up. In addition to our running example of Figure 2.1, the generalization relationships are finally deleted. Obviously, the operation contract for the refactoring pullUpAttribute is only fulfilled transiently on model version v_1'. It is neither fulfilled on model version v_1, nor on model version v_2. Thus, alternative edit steps, namely an attribute movement together with an attribute deletion, are reported by the operation detection algorithm[2]. Thus, the transient effect is erasable.

To sum up, transient effects are usually not harmful w.r.t. to our approach to edit operation recognition, provided that the effect of an operation sequence causing transient

[2]The given matching $M_{v_1,v_2} : v_1 \rightsquigarrow v_2$ determines which edit operations are actually detected in this example. If none of the *name* attributes are in a correspondence relationship, two attribute deletions and one creation are reported.

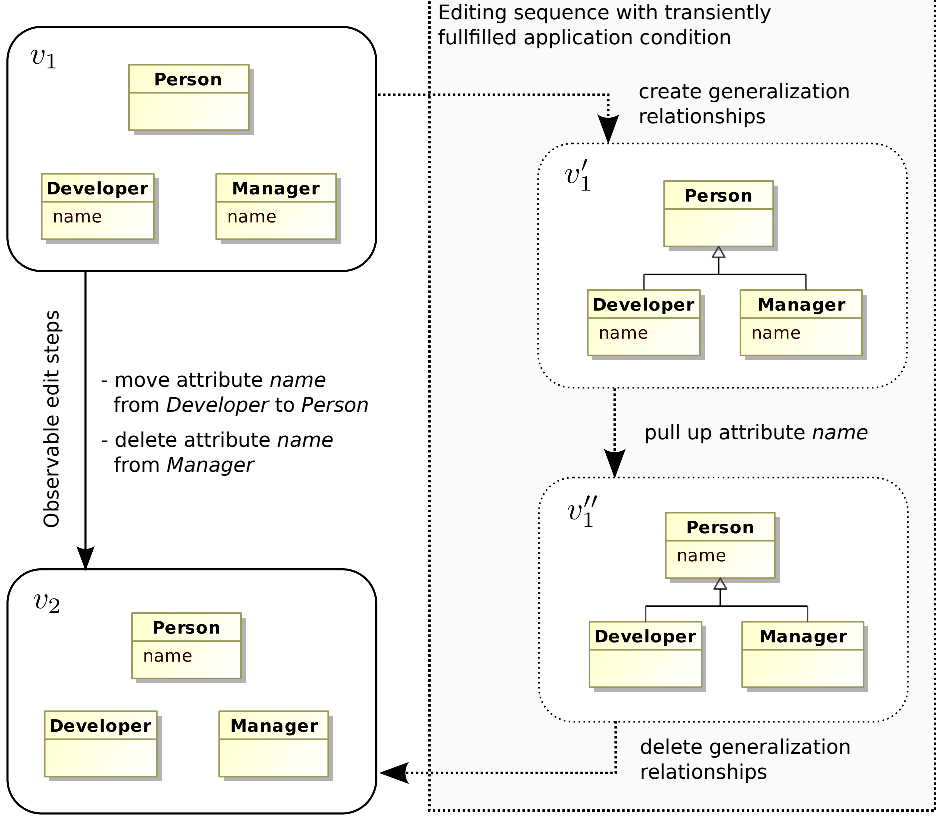


Figure 4.6: Example of a transient effect leading to a transiently fulfilled application condition (right) and how this transient effect can be avoided (left)

effects can be summarized by an alternative editing sequence. This is not absolutely necessary if differences are lifted for the sake of understandability, e.g. when certain evolution steps shall be documented. However, if we must guarantee a complete lifting of low-level differences, as it is the case for the generation of edit scripts (s. Chapter 5), the edit rule set for a given model type must be properly designed (s. Chapter 7).

CHAPTER 5

Generation of Edit Scripts

Chapter 4 addressed the question how to recognize executions of edit operations in a given low-level difference. However, this technique identifies edit operation executions only. This is useful for understanding changes, but not sufficient for replaying them in a change propagation scenario. In the latter case, the difference must be *executable*. Section 5.1 introduces our notion of an executable difference to which we refer to as *edit script*. Given a semantically lifted difference, edit scripts are generated in two subsequent processing steps of the differencing tool chain presented in Figure 1.2. The retrieval of actual edit operation parameters is described in Section 5.3, the analysis of dependencies between edit steps is considered in Section 5.4. A basic prerequisite for both processing steps is that trace links between edit and recognition rule elements are available. To that end, Section 5.2 precisely describes the tracing information being obtained from our recognition rule generation algorithm.

5.1 Edit Scripts

An edit script is a special kind of asymmetric difference which, in contrast to directed deltas according to [77, 180], additionally comprises information about dependencies of edit steps. In general, two edit steps depend on one another if they can be executed in one order and not in the other order, or lead to a different effect if executed in the reverse order, i.e. they do not commute. Section 5.4 will later show how to utilize critical pairs for the analysis of dependencies.

Definition 5.1 (Edit script)
Let A and B be models which are represented as ASGs being typed over the same type graph TG, then an edit script $\Delta_{A \Rightarrow B} = (S, D)$ is an executable specification of how model A can be edited in a step-wise manner to become model B. It comprises the following information:

S is a set of edit steps, each edit step invokes an edit operation and supplies appropriate actual parameters. Each edit operation must be applicable to models of type TG.

$D \subseteq S \times S$ is an acyclic relation that identifies pairs of edit steps which directly depend on each other. For each $s_1, s_2 \in S$ and $(s_1, s_2) \in D$, we say that s_2 directly depends on s_1; when $\Delta_{A \Rightarrow B}$ is applied, s_1 has to be executed first.

The transitive closure D^+ forms a binary partial order relation on S.

Note that Definition 5.1 leaves open which edit operations are available for modifying a model of a certain type. If edit scripts are used to propagate model changes, a particular requirement of our approach to model patching is that each edit operation used in an edit step must guarantee that the modified model remains displayable and thus editable in a standard editor for this model type (s. Chapter 6). We refer to this special kind of edit script as *consistency-preserving edit script*.

Definition 5.2 (Consistency-preserving edit script)
A consistency-preserving edit script $\Delta_{A \Rightarrow B} = (S, D)$ is an edit script in which each edit step $s \in S$ refers to a consistency-preserving edit operation.

$\Delta_{v_1 \Rightarrow v_2}$:
e_1: $restrictNavigability(in: \text{“}worksFor\text{”}, \text{“}employer\text{”})$
e_2: $createGeneralization(in: \text{“}Developer\text{”}, \text{“}Person\text{”};\ out: \text{“}Developer \rightarrow Person\text{”})$
e_3: $createGeneralization(in: \text{“}Developer\text{”}, \text{“}Person\text{”}; out: \text{“}Manager \rightarrow Person\text{”})$
e_4: $pullUpAttribute(in: \text{“}name\text{”}, \text{“}Developer \rightarrow Person\text{”})$
$e_2 \leftarrow e_4$
$e_3 \leftarrow e_4$

Figure 5.1: Edit script $\Delta_{v_1 \Rightarrow v_2}$ for Example 2.1 synthesized in a textual notation

Example 5.1 (Edit script obtained from two revisions of a UML class diagram)
Figure 5.1 shows a consistency-preserving edit script $\Delta_{v_1 \Rightarrow v_2}$ obtained from model versions v_1 and v_2 of our Example 2.1, synthesized in a textual notation. Edit steps

e_1 to e_4 are numbered as in Figure 2.1 and dependencies between them are explicitly listed: Generalization relationships "*Developer* → *Person*" and "*Manager* → *Person*" must first be created (e_2 and e_3) before attribute *name* can be pulled up (e_4). Note that we use symbolic identifiers to refer to model elements used as arguments of formal parameters, input and output parameters are indicated by key words *in* and *out*, respectively.

The conceptual data structure which is used to represent operation invocations, their invocation arguments and dependencies between operation invocations is shown in Figure 5.2. Note that an instance of this data structure is populated in a step-wise manner (s. Figure 1.2):

1. Initially, for each semantic change set which has been identified during operation recognition, an *OperationInvocation* that represents the invocation of a certain edit operation is being created (not shown in Figure 1.2). For our running example, four operation invocations are created; two of them, namely createGeneralization and pullUpAttribute, are shown in Figure 5.4.

2. Next, a complete operation invocation must be constructed, i.e. each formal parameter of an edit operation has to be bound to a concrete value.

3. Finally all operation invocations have to be analyzed for sequential dependencies.

Steps 2 (*parameter retrieval*) and 3 (*dependency analysis*) are explained in the remainder of this section. Note that these steps do not depend on each other; they can be performed in any order, symbolized by the parallel processing in Figure 1.2.

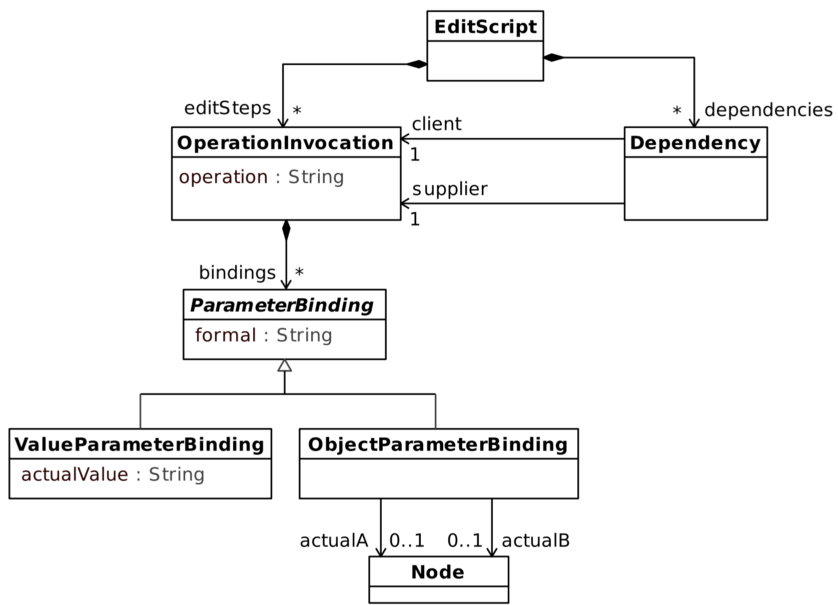


Figure 5.2: Conceptual data structure for the representation of edit scripts

5.2 Prerequisite: Tracing of Edit Rule Elements

During the translation of an edit rule to a recognition rule, trace links are established between edit rule elements and their respective recognition rule counterparts. These trace links are managed persistently. The set of trace links can be formally defined as a binary relation on the overall set of distinct graph elements defined by er and rr, respectively. Therefore, we will first introduce some basic notions to refer to sets of distinct graph elements defined by a Henshin rule.

Let r be a Henshin transformation rule according to Definition 3.11, then

$$\bigcup NAC_r = \bigcup_{i=1}^{|NAC_r|} nac_i \tag{5.2.1}$$

refers to the set of distinct graph elements being defined by negative application conditions of r. Analogously,

$$\bigcup PAC_r = \bigcup_{i=1}^{|PAC_r|} pac_i \tag{5.2.2}$$

denotes the set of all distinct graph elements being defined by positive application conditions of r. Thereupon, the set of all distinct graph elements being defined by the components of r can be defined as

$$E_r = L_r \cup R_r \cup \left(\bigcup NAC_r\right) \cup \left(\bigcup PAC_r\right). \tag{5.2.3}$$

Definition 5.3 (Trace relation and trace mappings)
Let er be an edit rule and rr its corresponding recognition rule, then the set of trace links being established during the translation of er to rr forms a binary relation $T \subseteq E_{er} \times E_{rr}$, where E_{er} and E_{rr} are defined according to (5.2.3). A trace link is represented by a pair of graph elements $(e_{er}, e_{rr}) \in T$. The trace relation can be partitioned to form a pair of partial mappings (t_A, t_B):

- $t_A : (E_{er} \setminus Cre_{er}) \rightsquigarrow E_{rr}$
- $t_B : (E_{er} \setminus Del_{er}) \rightsquigarrow E_{rr}$

Elements that have to be found on an original model version A (cf. Section 4.2.2) are mapped by t_A, while elements that have to be found on the revised version B are mapped by t_B. Note that these mappings can be utilized to implement the subroutines TRACEA(Node n_{er}) and TRACEB(Node n_{er}) of Algorithm 3.

Example 5.2 (Trace links between edit and recognition rule elements)
Consider the example shown in Figure 5.3. On top, the kernel part of edit rule pullUpAttribute (cf. Section 3.3.2) is shown, while the corresponding recognition rule rr_pullUpAttribute is shown in the middle of Figure 5.3. Note that we generally omitted edges of type *class* and *generalization* for the sake of readability. Trace links between

edit rule nodes and their respective recognition rule counterparts are indicated by dotted arrows in Figure 5.3. Note that trace links between edges are omitted for the sake of readability, they are implicitly given by traces between source and target nodes. Trace links illustrate how context, application conditions and change actions of an edit rule are translated to the respective recognition rule. Please note that NAC nac1 is to be interpreted as postcondition here. The same interpretation is chosen for node g which is to be preserved by the edit rule pullUpAttribute and which is neither boundary to edge deletions nor to edge creations. Thus, node g together with its incident edges can be conceptually considered as positive application condition (s. Section 4.2.2), which is to be interpreted as postcondition here: A pullUpAttribute operation shall be detected only if class cg, to which the attribute p is moved, is still a superclass of cs.

5.3 Retrieval of Actual Parameters

The parameter retrieval step in our differencing pipeline of Figure 1.2 has to bind actual parameters, i.e. operation arguments, to each formal parameter declared by the invoked edit operations. We illustrate the general proceeding for edit steps e_2 and e_4 of our sample edit script of Figure 5.1.

Value parameters can be retrieved by analyzing a match of the recognition rule. Each concrete value can be located in the revised model; it is bound to the respective recognition rule parameter. In our example, parameter n is a value parameter. The concrete value is "name". The respective value parameter binding is shown in Figure 5.4. It illustrates that value parameters are bound to the String representation of primitive data values.

In order to retrieve the actual values of *object parameters*, the occurrences of edit rule nodes in model versions A and B of a difference can be utilized. These occurrences are identified by composing two mappings which are illustrated in Figure 5.3:

1. The trace links, labeled t_A and t_B, between edit rule nodes and recognition rule nodes, which are created and maintained statically for all pairs of edit and recognition rules (cf. Section 5.2).

2. The matches of recognition rule nodes, labeled m, which are created during operation detection, i.e. when recognition rules are applied to a concrete low-level difference.

Thus, edit rule node occurrences in model version A of a difference are identified by the mapping $o_A = m \circ t_A$, while edit rule node occurrences in model version B of a difference are identified by the mapping $o_B = m \circ t_B$. Two node occurrences, namely $o_B(gen) = m(t_B(gen))$ and $o_B(g) = m(t_B(g))$, are explicitly shown for our running example in Figure 5.4, while the recognition rules themselves are omitted in this figure.

Having these edit rule node occurrences at hand, object parameter bindings can be created for formal parameters, e.g. gen of createGeneralization and g of pullUpAttribute. In

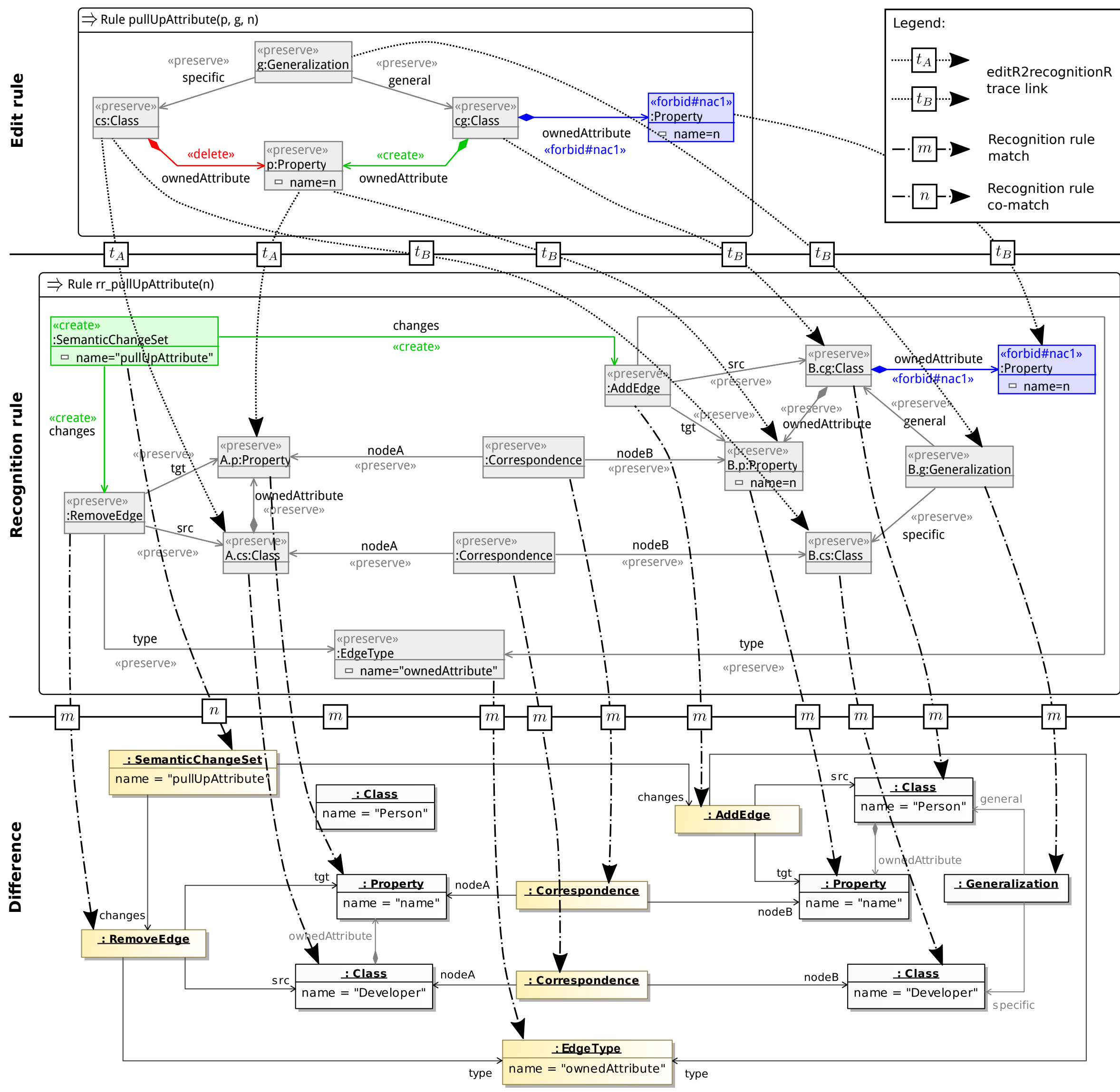


Figure 5.3: Relations between edit rule, recognition rule, and the occurrence of the semantic change set pullUpAttribute in the low-level difference δ_{v_1,v_2} of Example 2.1

the same way, object parameter bindings are created for all other formal parameters of the example rules, namely c1, c2, and p. Note that variables cg and cs are note declared as formal parameters by rule pullUpAttribute and thus no parameter bindings are created.

The concrete node which is bound to a formal object parameter can be located in the original model version A, or in the revised model B (referred to as *actualA* and *actualB* in Figure 5.2). Nodes being created exist only in model B, e.g. our created node of type *Generalization* of Figure 5.4. Nodes which are deleted exist only in the original model A. All other arguments have representations in both model versions, the respective nodes

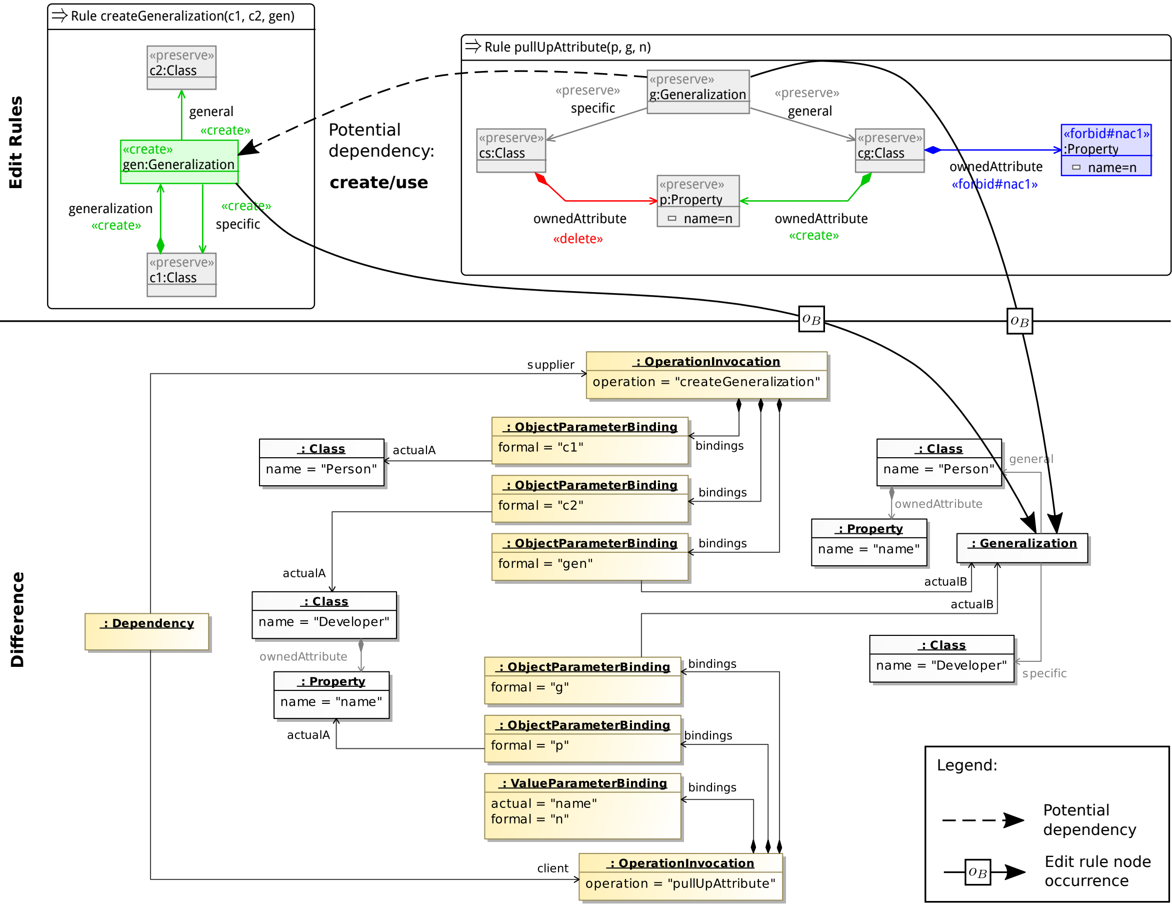


Figure 5.4: Extraction of an edit script for our example difference (excerpt)

are in a correspondence relationship. Nodes representing classes *Person* and *Developer* as well as attribute *name* are examples of this. Note that correspondence relationships between these elements and edges of type *actualB* are omitted in Figure 5.4 for the sake of readability.

Actual parameters can be used as *input* and/or *output* parameters (s. Section 3.3.2). For example, parameter g is an input parameter of operation pullUpAttribute. It determines the generalization along which the attribute represented by property p is to be pulled up. Parameter *gen* of createGeneralization is an output parameter; it returns the created node of type *Generalization*. An element used as actual output parameter of one operation invocation can also be used as actual input parameter of a second operation invocation. Such a case leads to a sequential dependency between both invocations.

5.4 Dependency Analysis

Sequential dependencies between edit operation invocations which are caused by input and output arguments are obvious. In the excerpt shown in Figure 5.4, the invocation of

pullUpAttribute sequentially depends on the creation of a generalization relationship by operation createGeneralization. However, not all model elements affected by an operation invocation are necessarily provided as operation arguments. Moreover, dependencies can also be caused by application conditions of edit rules. Thus, a complete dependency analysis cannot be reduced to the analysis of input and output arguments.

Obviously, testing all combinations of pairs of edit steps for dependencies is infeasible. Instead, we utilize our rule-based implementation of edit operations in Henshin in order to provide an efficient dependency analysis. To reduce the set of candidates for dependencies that have to be checked, all pairs of edit rules are statically analyzed for *potential dependencies* by using *critical pairs*.

Potential dependencies. As explained in Section 3.3.3, there are four possible kinds of dependencies if a rule r_2 is applied after a rule r_1; create/use, change/use, delete/-forbid and change/forbid.

In addition, the application of rule r_2 can have effects that would prevent r_1 from being successfully applied after r_2. According to the theory of algebraic graph transformation, these cases are referred to as conflicts of kinds delete/use, change/use, create/forbid, and change/forbid (cf. Section 3.3.3). As opposed to graph transformation theory, we treat these cases here as dependencies, too. In our context of edit script generation, a dependency relation shows how a conflict between two rule applications can be resolved by applying the rules in a specific order. Consequently, we refer to these kinds of dependencies as use/delete, use/change, forbid/create, and forbid/change in order to indicate which of the two rules has to be applied first.

Table 5.1 shows an overview of all kinds of potential dependencies. Note that rule r_2 potentially depends on r_1, i.e., if the potential dependency becomes an actual one, then r_1 has to be applied before r_2. All these different kinds of potential dependencies define partial mappings between rules r_1 and r_2 in the sense that a rule element is mapped to another one if the first one is dependent on the second one. These partial mappings are called *potential dependency mappings*, since they do not necessarily occur in actual rule applications.

Context/PAC		**NAC**	
Nodes/Edges	**Attributes**	**Nodes/Edges**	**Attributes**
create/use	change/use	delete/forbid	change/forbid
use/delete	use/change	forbid/create	forbid/change
Dependency relationship: $r_2 \rightarrow r_1$ Order of application: $r_1; r_2$			

Table 5.1: Possible kinds of dependencies in the context of edit script generation

Actual dependencies. The set of potential dependencies allows us to analyze *actual dependencies* of operation invocations efficiently. Section 5.4.1 explains our analysis

procedures for dependencies of kind create/use and use/delete, while delete/forbid and forbid/create dependencies are considered in Section 5.4.2.

Please note that dependencies which are caused by attribute changes are not discussed here in detail. The analysis proceeds almost analogously to the analysis of dependencies being caused by the creation (deletion) of structural graph elements. Analysis procedures for dependencies of kind change/use (use/change) include checks for create/use (use/delete) dependencies being caused by the creation (deletion) of edges to value nodes. Analysis procedures for dependencies of kind change/forbid (forbid/change) include checks for delete/forbid (forbid/create) dependencies being caused by the creation (deletion) of edges to value nodes.

5.4.1 Dependencies of Kind create/use and use/delete

Dependencies of kind create/use and use/delete are the most common kinds of dependencies. They typically occur when models are built (create/use) or reduced (use/delete) in a step-wise manner. We illustrate the analysis procedure using the sample editing sequences of Figure 5.5. A create/use dependency occurs in the editing sequence createGeneralization;pullUpAttribute from left to right, while a use/delete dependency occurs for the application of inverse edit rules pushDownAttribute and deleteGeneralization in reverse order.

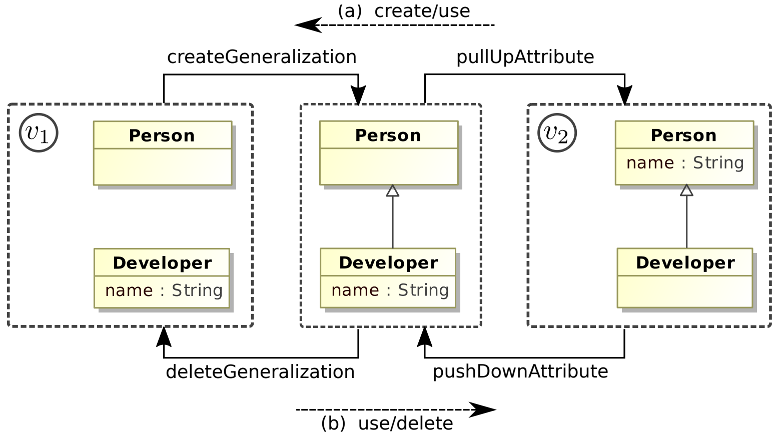


Figure 5.5: Sample editing sequence leading (a) to a create/use dependency, and (b) to a use/delete dependency in reverse order. Solid arrows represent edit steps, while dotted arrows express dependencies.

Dependencies of kind create/use. In general, a potential dependency of kind create/use between a critical pair of rules is an actual one between the applications of these rules if the minimal rule application of a critical pair can be *embedded* into the actual model changes: Given two rule applications $M1 \overset{r_1,m_1,n_1}{\Longrightarrow} M2$ and $M2 \overset{r_2,m_2,n_2}{\Longrightarrow} M3$, they are *actually* in a create/use dependency if there are at least two elements $e_1 \in r_1$

and $e_2 \in r_2$ such that there is a potential dependency mapping from e_2 to e_1 and $n_1(e_1) = m_2(e_2)$.

The application of this basic principle is illustrated in Figure 5.4. A potential dependency mapping of kind create/use is indicated by a dashed arrow in the upper part of Figure 5.4; the generalization gen has to be created first, potentially by an application of createGeneralization, before it can be used as required generalization g by pullUpAttribute. In order to test the above condition, we utilize again the occurrences of edit rule nodes in the graph-based representation of a low-level difference $\delta_{A,B}$. In our example, the applications of createGeneralization and pullUpAttribute are actually dependent because *gen* of createGeneralization and *g* of pullUpAttribute are mapped to the same model element, i.e. we have $o_B(gen) = o_B(g)$.

Dependencies of kind use/delete. The "embedding condition" for dependencies of kind use/delete is similar to the condition for dependencies of kind create/use: Given two rule applications $M1 \overset{r_1,m_1,n_1}{\Longrightarrow} M2$ and $M2 \overset{r_2,m_2,n_2}{\Longrightarrow} M3$, they are *actually dependent* in terms of a use/delete dependency if there are at least two elements $e_1 \in r_1$ and $e_2 \in r_2$ such that there is a potential dependency mapping from e_2 to e_1 and $m_1(e_1) = m_2(e_2)$.

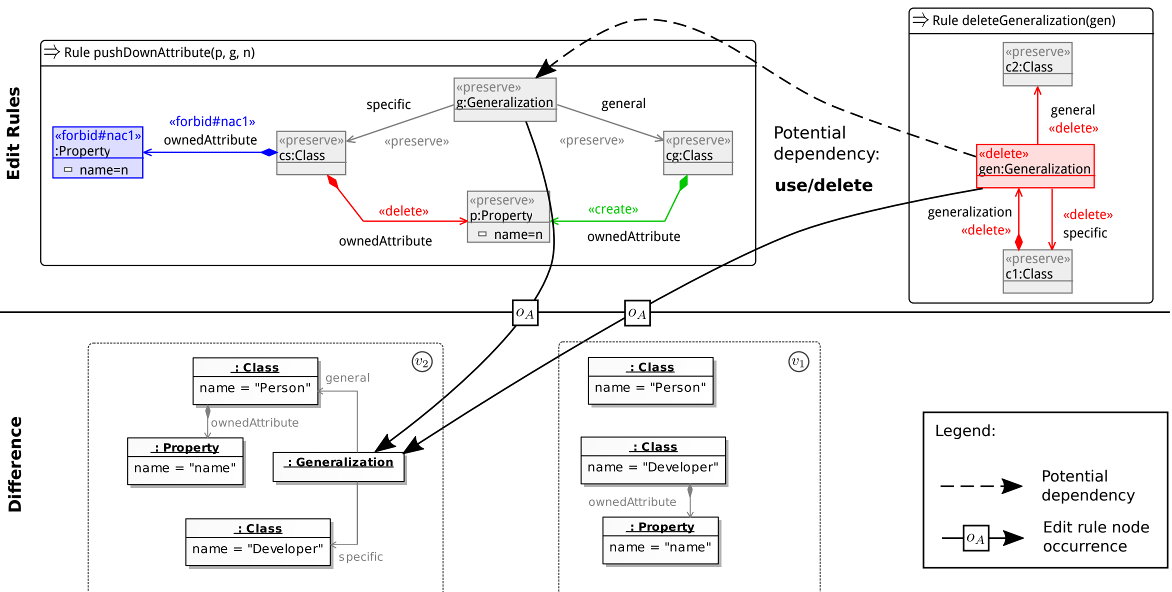


Figure 5.6: Example: Detection of dependencies of kind use/delete

A potential dependency mapping of kind use/delete is indicated by a dashed arrow in the upper part of Figure 5.6. The lower part shows ASG representations of model versions v_2 and v_1 of Figure 5.5. Note that we consider the sample editing sequence of Figure 5.5 from right to left, i.e. version v_2 is the original model while v_1 is the revised model of the difference calculation. Here, generalization g has to be used by pushDownAttribute before being deleted, potentially as generalization gen of deleteGeneralization. Testing the above condition proceeds analogously to dependencies of kind create/use.

In our example, the applications of pushDownAttribute and deleteGeneralization are actually dependent because *gen* of deleteGeneralization and *g* of pushDownAttribute are mapped to the same model element, i.e. we have $o_A(gen) = o_A(g)$.

5.4.2 Dependencies of Kind delete/forbid and forbid/create

Dependencies of kind delete/forbid and forbid/create typically occur if complex edit rules define negative application conditions as part of their operation contract. In order to illustrate our analysis procedure, we use editing sequences on a sample feature model to keep the example as small as possible (s. Figure 5.7).

From left to right, we apply the refactoring inlineOrGroup such that features *F2* and *F3* become two separate optional features. Next, we convert feature *F3* to a mandatory feature by applying edit operation setFeatureMandatory. The second step depends on the first one because we cannot convert a feature which is part of a group. A forbid/create dependency occurs for the application of inverse edit operations extractOrGroup and setFeatureOptional in reverse order. Here, solitary feature *F3* can only be converted to an optional feature as long as it is a solitary feature. Thus, operation setFeatureOptional has to be applied first, i.e. the second step extractOrGroup sequentially depends on the first one because it would prevent setFeatureOptional from being successfully applied.

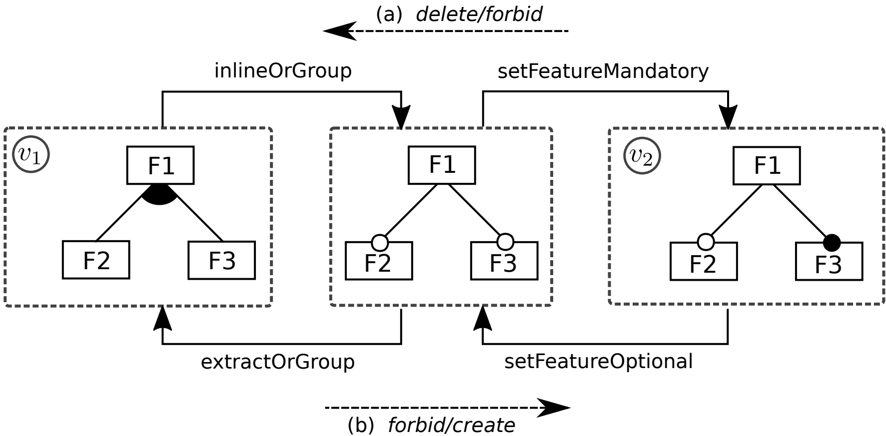


Figure 5.7: Sample editing sequence leading (a) to a delete/forbid dependency, and (b) to a forbid/create dependency in reverse order. Solid arrows represent edit steps, while dotted arrows express dependencies.

The detection of dependencies of kind delete/forbid and forbid/create differs from our previous cases of Section 5.4.1 because there are no occurrences of NAC elements in a low-level difference that can be utilized for that purpose. In the remainder of this section, we illustrate the analysis procedures to detect these kinds of dependencies using the sample editing sequences of Figure 5.7. To understand the Henshin rules implementing the edit operations which are used in our example of Figure 5.7 better, an excerpt of the meta-model for feature diagrams over which these rules are typed is shown in Figure 5.8: A *FeatureModel* basically serves as container. It contains a

distinguished *Feature* called root feature, which represents the root of a feature hierarchy. A feature contains an arbitrary number of child features such that features can be organized in a tree structure. A feature has a *name* and can be mandatory or optional, depending on the value of attribute *mandatory*. Features can be grouped and a *Group* comprises at least two features. Alternative groups are distinguished from or-groups by the value of the attribute *groupType*. The respective data type, called *GroupType*, is an enumeration type that defines two literals. Cross-tree relations and several invariants are omitted in Figure 5.8 as they are irrelevant for our running example. A full version of this meta-model for feature diagrams can be found in [20].

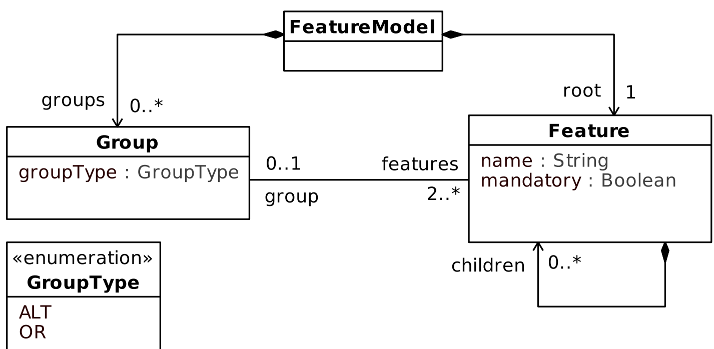


Figure 5.8: Subset of the meta-model for feature diagrams introduced in [20]

Dependencies of kind delete/forbid. In general, our approach to operation recognition implies that a NAC must be fulfilled on the original model version A or on the revised version B. Otherwise, the NAC is only transiently fulfilled and an application of a rule that specifies this NAC cannot be recognized by our operation detection approach at all (s. Section 4.4). W.r.t. to the occurrence of actual dependencies of kind delete/forbid, we can conclude from this principal restriction that an actual delete/forbid dependency can only be found in a difference if

- an element which is being deleted such that a potential delete/forbid dependency becomes an actual one must be contained by the original model version A, and
- the NAC which causes the potential dependency has to be fulfilled on the revised model version B.

In other words, the above conditions state that we have an actual delete/forbid dependency only if a NAC is fulfilled on the revised model version B but not on the original version A.

Thus, the idea here is to search on model version A for a particular NAC pattern which does not exist on version B. If such a NAC pattern is found, we can conclude that certain elements of this pattern must have been deleted such that the NAC is fulfilled on version B. This proceeding is illustrated in Figure 5.9. On top, we have edit rules implementing the edit operations inlineOrGroup and setFeatureMandatory. A

potential dependency mapping of kind delete/forbid is indicated by a dashed arrow. ASG representations of our sample model versions v_1 and v_2 of Figure 5.7 are shown in the lower part of Figure 5.9. The corresponding recognition rules as well as representations of correspondences, low-level changes, semantic change sets and operation invocations of our difference are omitted for the sake of readability.

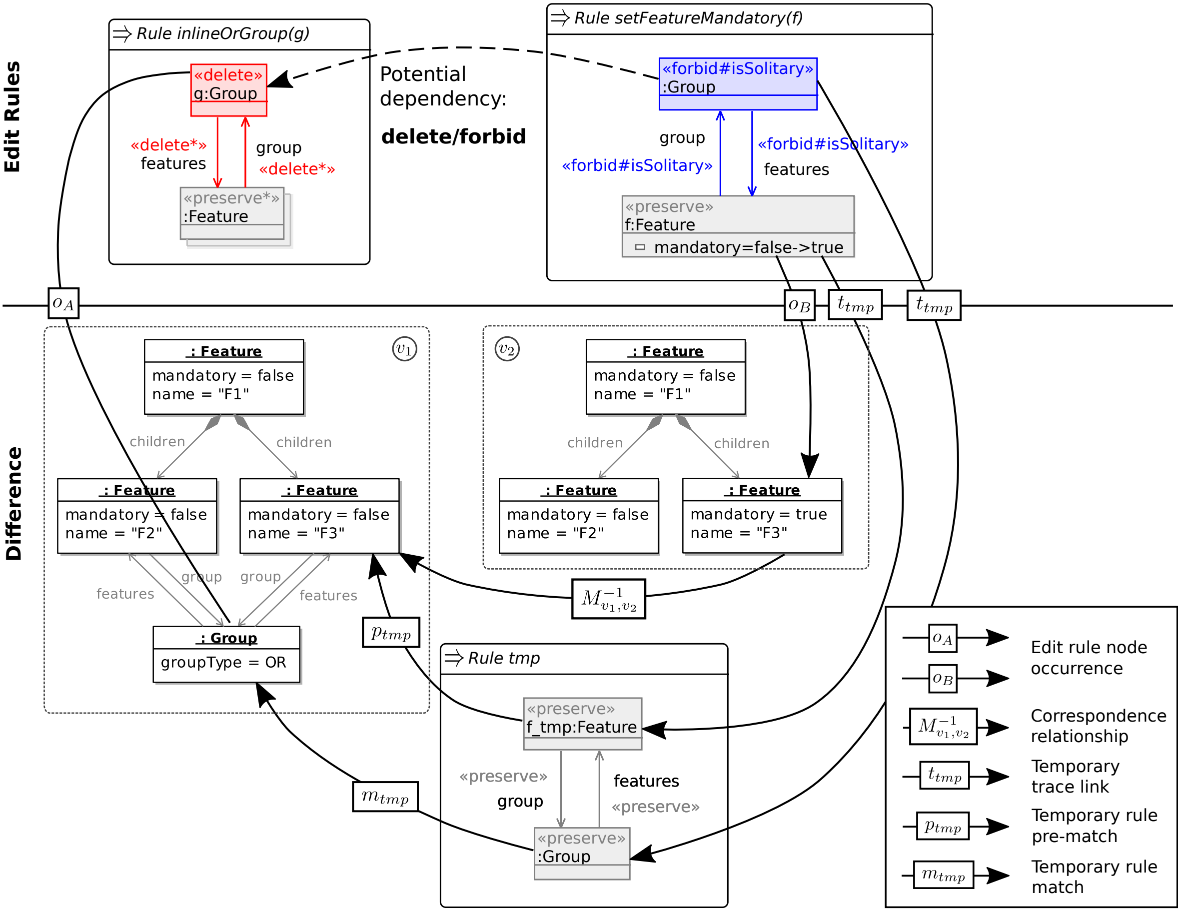


Figure 5.9: Example: Detection of dependencies of kind delete/forbid

In general, let r_1, r_2 be edit rules for which applications have been recognized in a given low-level difference $\delta_{A,B}$. Let further $nac \in NAC_{r_2}$ be a negative application condition which potentially causes a delete/forbid dependency w.r.t. rule r_1 such that r_1 has to be applied before r_2, i.e. r_2 potentially depends on r_1. The actual dependency analysis proceeds in four sequential steps:

1. For each occurrence $o_B(n)$ of a boundary node n of nac in the revised model B, we try to find the corresponding node of $o_B(n)$ in the original model version A. In order to find such a corresponding node in A, we utilize the inverse mapping $M^{-1}_{A,B}$ of matching $M_{A,B}$. If a corresponding node can be found for each occurrence of all boundary nodes of nac, we proceed with step 2.

In our example of Figure 5.9, there is only one boundary node f of NAC isSolitary, occurrence $o_B(f)$ refers to the ASG node representing feature *F3* in version v_2. We obtain the corresponding node of $o_B(f)$ in version v_1 as $M^{-1}_{v_1,v_2}(o_B(f))$.

2. The NAC pattern *nac* together with all its boundary nodes is translated to a temporary rule tmp. This temporary rule serves as a pattern specification which has to be found on model version A. Therefore, it contains only elements which are to be found and preserved. Similar to the translation of edit rules to recognition rules, this transformation leads to a trace mapping $t_{tmp} : (nac \cup B_{nac}) \rightarrow Cxt_{tmp}$.

3. Subsequently, we try to find all matches of temporary rule tmp in model A. Thereby, all boundary node occurrences in model A which have been obtained in step 1 serve as pre-match for this pattern matching procedure. The pre-match is denoted by a partial mapping $p_{tmp} : L_{tmp} \rightsquigarrow A$. If there is at least one complete match $m_{tmp} : L_{tmp} \rightarrow A$, we proceed with step 4.

   In our example, pre-match p_{tmp} maps the temporary rule node f_tmp to the ASG node representing feature *F3* in version v_1. This partial match can be extended to a complete match m_{tmp} as shown in Figure 5.9.

4. Finally, for each match $m_{tmp} : L_{tmp} \rightarrow A$, the test for an actual dependency proceeds similar to the analysis of actual dependencies of kind create/use and use/delete: A potential dependency mapping between rule elements $e_2 \in r_2$ and $e_1 \in r_1$ becomes an actual one between the applications of these rules if e_1 and e_2 are mapped to the same element in model A, i.e. we have $o_A(e_1) = m_{tmp}(t_{tmp}(e_2))$.

   In our example, the applications of setFeatureMandatory and inlineOrGroup are actually dependent because the NAC node of type *Group* of setFeatureMandatory and node g of inlineOrGroup are mapped to the same model element in version v_1.

Dependencies of kind forbid/create. Similar to the analysis of dependencies of kind delete/forbid, we can conclude from principal restrictions of our approach to operation detection that an actual forbid/create dependency can only be found in a difference if a NAC is fulfilled on the original model version A but not on the revised version B. Thus, the idea here is to search on model version B for a particular NAC pattern which does not exist on version A. If such a NAC pattern is found, we can conclude that certain elements of this pattern must have been created since the NAC is fulfilled on version A. This proceeding is illustrated in Figure 5.10. On top, we have edit rules implementing the edit operations setFeatureOptional and extractOrGroup. A potential dependency mapping of kind forbid/create is indicated by a dashed arrow. ASG representations of our sample model versions v_1 and v_2 of Figure 5.7 are shown in the lower part of Figure 5.10. Note that we consider the sample editing sequence of Figure 5.7 from right to left, i.e. version v_2 is the original model while v_1 is the revised model of the difference calculation. Again, the corresponding recognition rules as well as representations of correspondences, low-level changes, semantic change sets and operation invocations of our difference are omitted for the sake of readability.

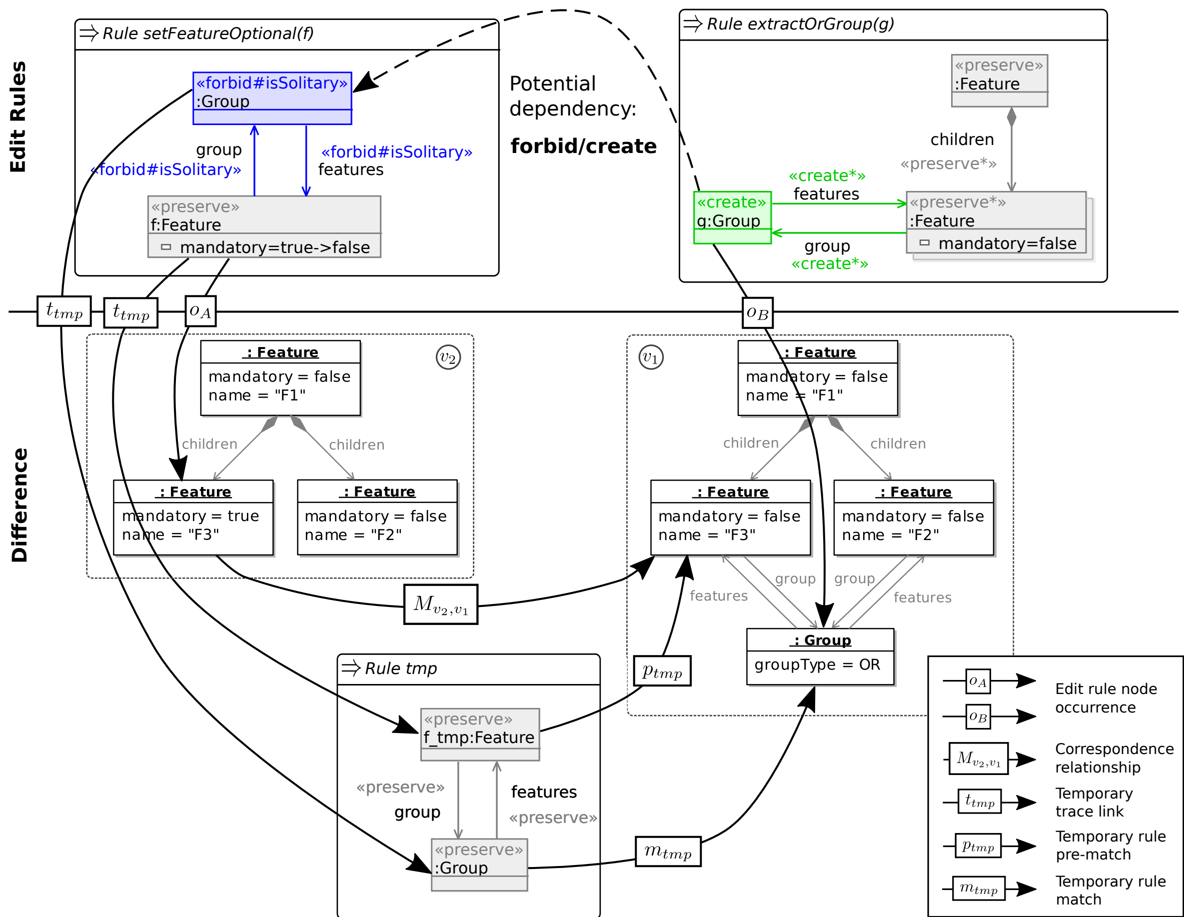


Figure 5.10: Example: Detection of dependencies of kind forbid/create

In general, let r_1, r_2 be edit rules for which applications have been recognized in a given low-level difference $\delta_{A,B}$. Let further $nac \in NAC_{r_1}$ be a negative application condition which is potentially invalidated by rule r_1 such that r_1 has to be applied before r_2, i.e. r_2 potentially depends on r_1. The actual dependency analysis proceeds in four sequential steps:

1. For each occurrence $o_A(n)$ of a boundary node n of nac in the original model A, we try to find the corresponding node of $o_A(n)$ in the revised model B utilizing matching $M_{A,B}$. If a corresponding node can be found for each occurrence of all boundary nodes of nac, we proceed with step 2.

   In our example of Figure 5.10, there is only one boundary node f of NAC isSolitary, occurrence $o_A(f)$ refers to the ASG node representing feature *F3* in version v_2. We obtain the corresponding node of $o_A(f)$ in version v_1 as $M_{v_2,v_1}(o_A(f))$.

2. The NAC pattern nac together with all its boundary nodes is translated to a temporary rule tmp. This temporary rule serves as a pattern specification which has to be found on model B. Therefore, it contains only elements which are

to be found and preserved. The transformation leads to a trace mapping $t_{tmp} : (nac \cup B_{nac}) \rightarrow Cxt_{tmp}$.

3. Subsequently, we try to find all matches of temporary rule tmp in model B. Thereby, all boundary node occurrences in model B which have been obtained in step 1 serve as pre-match for this pattern matching procedure. The pre-match is denoted by a partial mapping $p_{tmp} : L_{tmp} \rightsquigarrow B$. If there is at least one complete match $m_{tmp} : L_{tmp} \rightarrow B$, we proceed with step 4.

   In our example, pre-match p_{tmp} maps the temporary rule node f_tmp to the ASG node representing feature *F3* in version v_1. This partial match can be extended to a complete match m_{tmp} as shown in Figure 5.10.

4. Finally, for each match $m_{tmp} : L_{tmp} \rightarrow B$, the test for an actual dependency proceeds as usual: A potential dependency mapping between rule elements $e_2 \in r_2$ and $e_1 \in r_1$ becomes an actual one between the applications of these rules if e_1 and e_2 are mapped to the same element in model B, i.e. we have $o_B(e_2) = m_{tmp}(t_{tmp}(e_1))$.

   In our example, the applications of extractOrGroup and setFeatureOptional are actually dependent because the NAC node of type *Group* of setFeatureOptional and node g of extractOrGroup are mapped to the same model element in version v_1.

CHAPTER 6

Propagation of Changes based on Consistency-preserving Edit Scripts

Section 2.1 describes several versioning scenarios (SC2-5) which can be informally described as the propagation of model changes from one version of a model to another. We support these development tasks adopting the principle of document patching. Changes which are to be propagated are expressed as an edit script. We define two basic variants of a patch operator to which we refer to as *patch* and *update*, respectively.

Our *patch operator* is defined as usual and applies an edit script to a target model in order to generate a revised version of this model:

$$\textbf{patch} : \mathit{Model} \times \mathit{EditScript} \rightarrow \mathit{Model}$$

The basic principle of patching a model by applying an edit script is illustrated in Figure 6.1(a). Edit script $\Delta_{v_0 \Rightarrow v_1}$ specifies changes between an original model v_0 and its revision v_1. It is to be executed on a target model v_2. This proceeding can be applied to scenarios SC2-4, while Figure 6.1(a) illustrates the most general case where versions v_0/v_1 and v_2 are unrelated and managed in different repositories. Please note that references to model elements in an edit script $\Delta_{v_0 \Rightarrow v_1}$ are initially, when $\Delta_{v_0 \Rightarrow v_1}$ is being created, references to elements of the original model v_0. Depending on the implementation of an argument resolution procedure (s. Section 6.2), v_0 must be available at the patch appliers site and serves as additional input parameter of our patch operator:

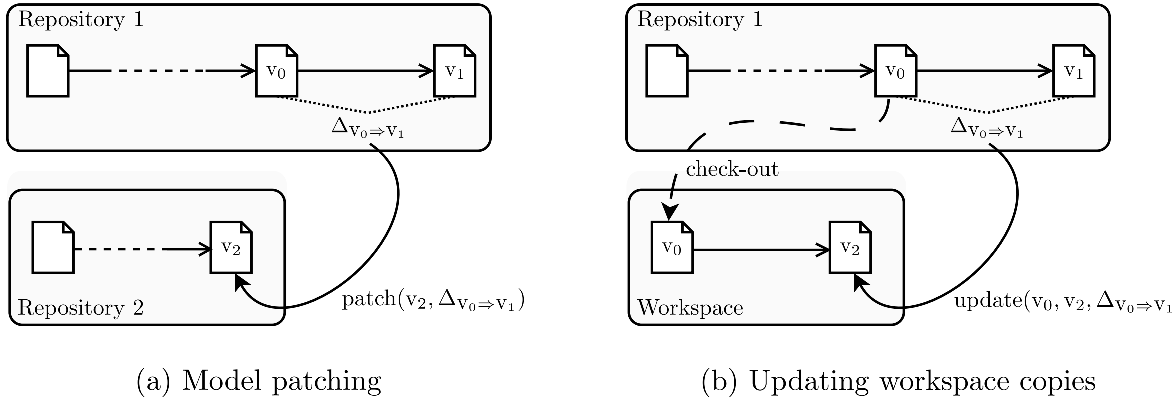
$$\textbf{patch} : \mathit{Model} \times \mathit{Model} \times \mathit{EditScript} \rightarrow \mathit{Model}$$

Note that this second variant of our patch operator is not shown in Figure 6.1(a).

We consider the updating of workspace copies (scenario SC5) as a special, extended form of model patching:

$$\textbf{update} : Model \times Model \times EditScript \rightarrow Model$$

As illustrated in Figure 6.1(b), the target model is the workspace version v_2 of a model, edit script $\Delta_{v_0 \Rightarrow v_1}$ specifies the repository changes between versions v_0 and v_1 which are to be propagated to v_2. Additionally, update takes the base version v_0 as input in order to determine for each model element in v_2 being modified by the application of $\Delta_{v_0 \Rightarrow v_1}$ whether it shall be considered as changed compared to v_0 . If so, a warning against blindly overwriting local workspace changes is issued.



(a) Model patching

(b) Updating workspace copies

Figure 6.1: Propagation of model changes based on edit scripts: Patching (a) *with* and (b) *without* common base version

In order to understand the effects of an edit script, a user must be able to mentally execute each step in the edit script in any order compatible with its partial order. The effect of edit operations can be explained on the basis of the external representation of models. This implies that *each edit step leads to a correct, displayable intermediate model.* In other words, each edit operation used in an edit step must guarantee that the modified model remains consistent in the sense that it can be displayed and further edited. Thus, one key idea of our approach to propagate model changes is to use consistency-preserving edit scripts as a basis for model patching. We use the approach presented in Chapter 5 to create an initial version of an edit script and assume the edit script generation algorithm to be configured by a complete set of consistency-preserving edit rules[1]. Section 6.1 describes how to use and distribute edit scripts as model patches and how to modify edit scripts in a controlled way. Error and conflict detection procedures are discussed in Section 6.2, while Section 6.3 finally presents a method and graphical user interface which enables developers to apply consistency-preserving edit scripts to a target model in a controlled, interactive way.

[1] Note that a semi-automated approach to create a complete set of consistency-preserving edit rules for a given modeling language will be presented in Chapter 7.

6.1 Using Edit Scripts as Model Patches

A consistency-preserving edit script serves as initial version of a patch. As revealed by our domain analysis in Chapter 2, this initial version of the edit script must sometimes be reduced to a subset of edit steps which are considered as relevant by the patch creator. In case of scenario SC4, for instance, edit steps which are not related to the improvement which is to be propagated must be excluded from the edit script before it is distributed to the patch applier (s. Section 2.1.4). Thus, edit scripts become an *editable document* which must be represented in a suitable way. Section 6.1.1 presents an extension of our conceptual data model for the representation of edit scripts. Our approach to the guided editing of edit scripts is introduced in Section 6.1.2. Finally, the modified edit script must be prepared to be used by the patch applier at the patch applier's site. Depending on the application scenario, the edit script must be distributed with or without the original model and with or without implementations of the used CPEOs (s. Section 6.1.3).

6.1.1 Representation of Edit Scripts

A basic conceptual data model for the representation of edit scripts has been already introduced in Section 5.1. If edit scripts shall be used as patches, some further information is required. To that end, Figure 6.2 presents an extension of our data model of Figure 5.2 and serves as a central design artifact for patch tool implementors.

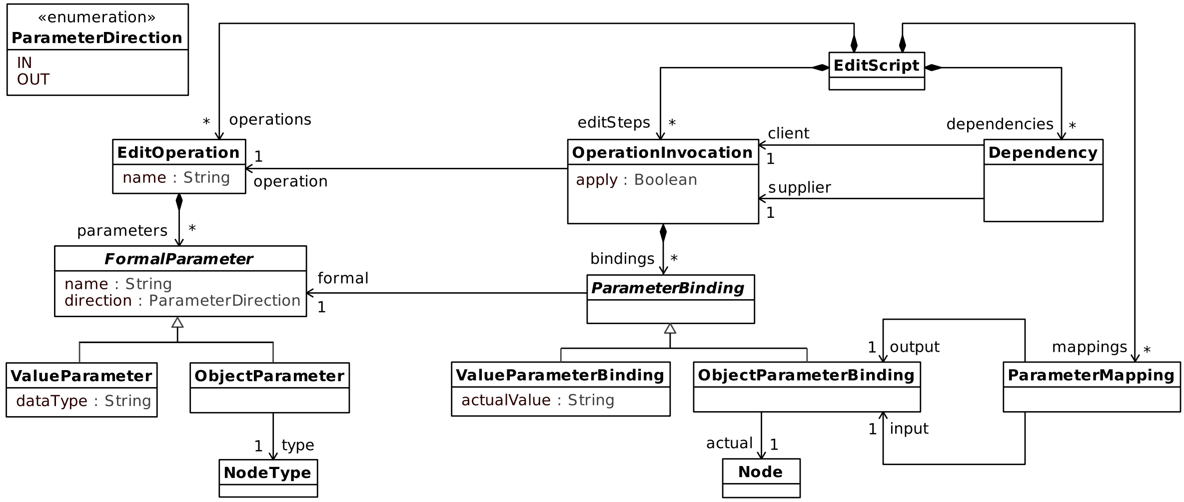


Figure 6.2: Representation of edit scripts being used as patches

In contrast to Figure 5.2, signatures of edit operations are also represented. Thus, proper CPEO implementations can be reliably selected at the patch applier's site, as we assume signatures of edit operations to be unique within the set of available operations for a given model type. As shown in Figure 6.2, an edit operation declares a set of formal parameters which are bound to actual parameters, i.e. arguments, of

an operation invocation. Moreover, output arguments of one edit step which are used in another sequentially dependent one are mapped onto each other via an instance of *ParameterMapping*. Thus, an edit script basically becomes a data flow graph of edit steps. Consequently, references to model elements in the revised model v_1 of the two model versions v_0 and v_1 of which an edit script $\Delta_{v_0 \Rightarrow v_1}$ has been created can be omitted. Finally, a boolean-valued attribute *apply* is added to class *OperationInvocation*. Thus, each edit step in an edit script can be "masked" (*apply=false*), i.e. the respective edit step is to be skipped when the edit script is applied. From a patch creator's point of view, the edit step (and all its dependent edit steps) is "logically" deleted from the edit script, but the deletion can be reverted in a patch editor (cf. Section 6.1.2). Masked edit steps can be "physically" deleted when the edit script is distributed to the patch applier's site (cf. Section 6.1.3).

Figure 6.3 illustrates how the excerpt of the edit script of Figure 5.4 is converted to an initial version of an edit script which can be used as patch. Note that we omitted types of formal parameters for the sake of readability.

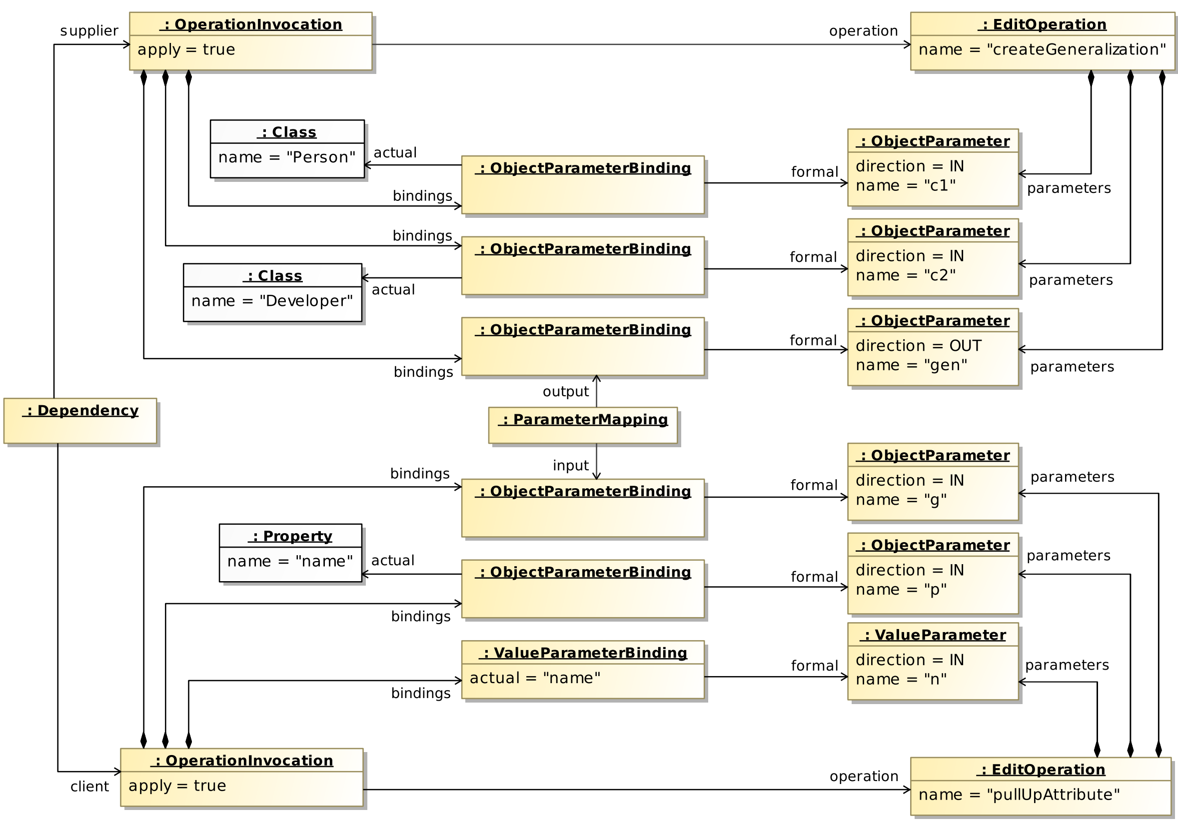


Figure 6.3: Excerpt of a sample edit script serving as initial version of a patch

6.1.2 Guided Editing of an Initial Version of an Edit Script

The manual modification of an edit script is a challenge because edit steps can depend on one another. Thus, a modified edit script cannot be executed successfully if an edit step is deleted without deleting all dependent edit steps. Consequently, edit scripts must be modified with caution. This should be supported by a dedicated "patch editor". Such an editor must present an edit script in a user-readable way. A particular challenge is the presentation of operation arguments which are references to "nameless" model elements, i.e. elements which have no user-readable identifier. Visual modeling languages such as the UML define many anonymous elements, e.g. pseudo-states in state charts, input pins of an action node in activity diagrams, etc. The only viable solution to show these arguments to a user is to use the visual representation of these model elements.

Figure 6.4 shows the graphical user interface of our patch editor. It is based on the difference presentation concept which is commonly known as *interactive list of edit steps* (s. Section 2.2.3). We choose this presentation technique because of its flexibility: Edit steps which are executions of complex edit operations can be interactively inspected. Moreover, the original model and the changed model are displayed in two separate native editor windows on the right. Thus, models are displayed in the notation users are familiar with. The example in Figure 6.4 shows two revisions of a class diagram (modeled in EMF Ecore) which correspond to the original and the changed version of Example 2.3.

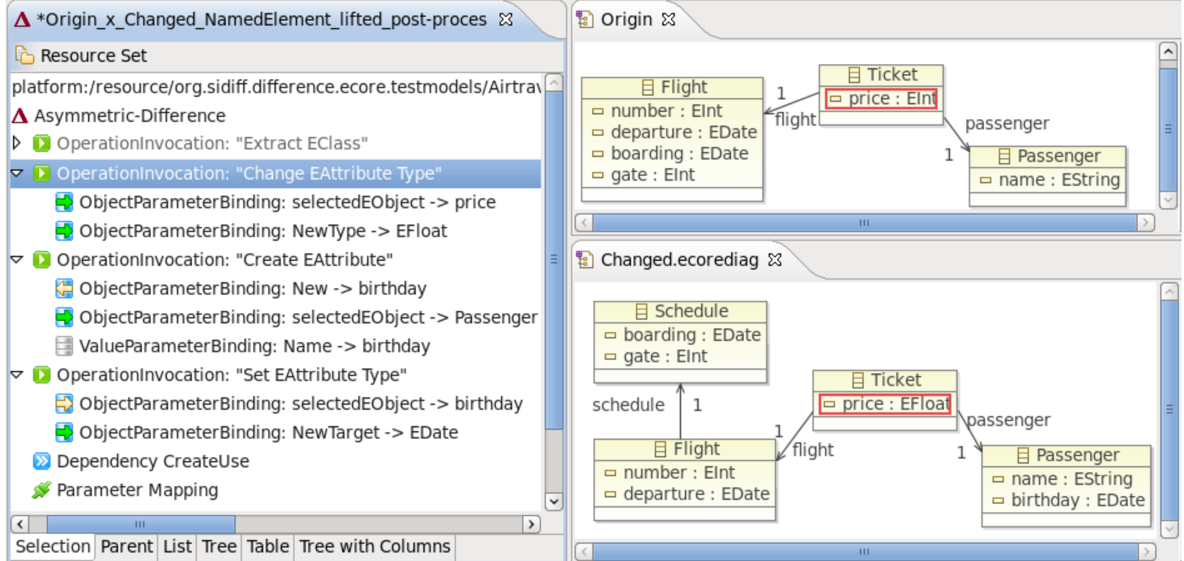


Figure 6.4: Guided editing of edit scripts

The control window on the left-hand side of the patch editor contains a list presenting the edit steps contained in the edit script. The order in which they are presented is consistent with the partial order which is specified by dependencies between the edit steps. Dependencies can be also examined in detail. Our example actually contains

only one sequential dependency; the attribute *birthday* must first be created before its datatype can be set to *EFloat*.

Moreover, operation arguments are shown for each edit step. Object parameter values are marked as input and output arguments. Output parameters which are mapped to input parameters of dependent edit steps are indicated separately, e.g. the attribute *birthday* which is created in one edit step and used in a subsequent one of our example. If an item of the control window is selected, the referenced model elements are focused and highlighted in the appropriate editor window; each operation argument can be selected individually. If a complete edit step is selected then all involved model arguments are highlighted.

The patch creator can remove individual edit steps from the edit script. Edit steps which depend on the removed one are removed implicitly; the user is made aware of these deletions. In our Example 2.3, we assume that original and target model are revisions of a system variant A which is supposed to manage the flight scheduling at the airport. The edit step extractClass("Schedule", Flight.gate, Flight.boarding) is removed from the edit script because it refactors parts of the original model, namely departure *gate* and the *boarding* time, which are highly specific to this special variant of the model. In this case, no dependent edit steps are removed implicitly.

6.1.3 Distribution of Edit Scripts

Typically, different hosts are used for the creation and editing of a patch on the one hand, and for the application of the patch on the other hand. Thus, in addition to the edit script itself, implementations of all CPEOs which are used by edit steps contained by the edit script must be available in the tool environment of the patch applier. Consequently, implementations of the required CPEOs are distributed together with the edit script. While we use Henshin rules for that purpose in our practical work, our approach to model patching does not depend on this choice; any other in-place model transformation technology could be used as well to implement edit operations. We only assume that each edit operation has an interface specifying all semantically relevant input and output parameters as well as an implementation and that a suitable transformation engine is available in the tool environment of the patch applier.

Before an edit script is finally distributed to a patch applier, references to elements of the original model can be "symbolized" [149], e.g. using path names or persistent identifiers. Thus, the edit script can be distributed without the original model. In some cases, however, the complete original model is required by the patch applier, e.g. to establish reliable correspondences between the original and the target model as explained in the following section.

6.2 Error and Conflict Detection

The application of an edit script proceeds basically as shown in Figure 6.1. A basic prerequisite is to resolve all references to model elements which occur as arguments of

edit steps (cf. Section 2.2.4).

Resolution of operation arguments. References to model elements in an edit script are initially, when an edit script is being created, references to elements of the original model (s. Section 6.1.1). Resolving these references at the patch applier's site means to find corresponding elements in the target model. In simple cases, in particular in case of scenario SC2, surrogates such as persistent identifiers can be used to reliably identify model elements.

If no reliable references are available, our approach is to compute a matching between the original model v_0 and the target model v_2 by using the matcher of a model comparison tool. The resulting matching defines for each element of v_0 its corresponding element in v_2. Of course, the original model v_0 must be available on the site where the edit script is applied. This can be achieved by either automatically producing a local backup copy whenever a model is checked out or checked in (e.g. in case of scenario SC5) or by distributing the original model together with the edit script (scenarios SC3 and SC4).

The following preparations are performed when the process of patching (updating) v_2 by edit script $\Delta_{v_0 \Rightarrow v_1}$ is started and before any interaction with the user:

1. A matching $M_{v_0,v_2} : v_0 \rightsquigarrow v_2$ between v_0 and v_2 is computed.

2. Some arguments of edit steps in $\Delta_{v_0 \Rightarrow v_1}$ are references to model elements in v_0. If such a model element e_0 in v_0 corresponds to e_2 in v_2, then the argument in the edit step is resolved to e_2. Otherwise the argument cannot be resolved and a missing argument error is flagged. The detection of wrongly chosen arguments, i.e. false positives being produced by a matcher, is discussed in Section 6.2.1.

3. If a workspace copy is to be updated, for each operation argument in v_2 a check is performed whether it shall be flagged as modified. Section 6.2.2 will describe how this check can be performed and, if necessary, be configured for a particular modeling language.

6.2.1 Detection of Wrongly Chosen Arguments

Virtually no matcher is free from matching errors [146]; the likelihood of errors depends on the characteristics of models and other factors. We have to distinguish two types of matching errors:

1. False negatives: A suitable model element exists but is not found. In this case, the reference must be resolved manually; the patch applier selects the resolving model element.

2. False positives: They lead, if undetected, to wrong operation arguments. The wrong correspondence must be deleted manually and the reference has to be resolved manually.

False negatives always force the patch applier to deal with the problem. False positives are more harmful because they are hard to detect: Without any further information, the patch applier can only detect them by manually checking all resolved references, which can be very tedious. Tool support can be provided by meta-data about the correspondences which have been established by the matcher. The SiDiff matching engine, for instance, delivers for each correspondence a value which estimates the reliability of this correspondence [259]. Section 6.3 will show how this information is displayed to a patch applier.

6.2.2 Warnings against Blind Overwriting of Local Changes

Section 2.4 has analyzed several reasons for the complexity of 3-way merge tools. Thus, a basic design decision of our approach to updating workspace copies is to avoid the calculation of a complete edit script $\Delta_{v0 \Rightarrow v_2}$ which leads to a complex analysis, representation and resolution of conflicts in classical operation-based merging. Instead, we install a rather simple warning mechanism against blindly overwriting local changes at the workspace site. Thus, the user interface of an interactive update tool, which will be presented in Section 6.3.2, is less complex than GUIs of interactive 3-way merge tools, and the user interaction concept does not force a user to mentally return to the common base version. The idea is to perform a check for each model element in v_2 being used as operation argument in $\Delta_{v0 \Rightarrow v_1}$ whether this element shall be flagged as modified against version v_0. In the remainder of this section, we present three different approaches to the detection of modified arguments.

Change-based detection of modified arguments. A straightforward way to perform a modification check is to compute a low-level difference δ_{v_0,v_2} between the base version v_0 and the revised workspace version v_2. An operation argument is considered to be modified if it is part of a change context, see (4.1.1) in Section 4.1, of δ_{v_0,v_2}.

Such a change-based detection of modified operation arguments is generic in the sense that it works for any model type, the only configuration parameter is the metamodel of this model type. On the contrary, the modification check is restricted to detect conflicting situations being caused by local changes only. In our Example 2.4 of scenario SC5, for instance, the check is sufficient for detecting the concurrent name change of operation *Ticket.getInfo* (conflict ①). However, it fails to reveal that performing the edit step createSubclass(SoccerMatch, Event) on v_2 potentially leads to an inconsistent definition of the attribute *artist* within the inheritance hierarchy of class *Event* (conflict ③). For the detection of this "semantic conflict", changes affecting local properties of class *Event* are irrelevant. Instead, we are rather interested in whether new attributes or associations have been added to class *Event* which will be implicitly inherited by all of its subclasses, in particular the new subclass *SoccerMatch* being created by createSubclass(Event, "SoccerMatch").

The example shows that sophisticated conditions under which an operation argument shall be flagged as modified must take the semantics (in the sense of meaning [121]) of the underlying model type and the way how instances of this type are edited into

account. To this end, we present two additional approaches which can be configured for a given model type and the set of available edit operations.

Signature-based detection of modified arguments. Signature-based detection of modified arguments is based on the principle of signature-based matching (cf. Section 2.3.1). For each model element in v_2 against v_0, a signature value is computed. A model element in v_2 is flagged as modified if the signature values of this element and the corresponding element in v_0 differ. In practice, a signature value is typically a hash value which incorporates many other values.

We adopt here the approach which is used to configure the signature-based matcher of the SiDiff matching pipeline [2]. A basic configuration defines for each relevant type, i.e. each node type declared by the meta-model, a list of data items to be incorporated in the signature. Data items can be local properties, properties at remote nodes which are specified using a path language, or signature values of remote nodes. The path language is similar to XPath [263] and defines axes *parent*, *children* and *neighbors* to locate an ASG node (or sets of nodes) relative to some other. We extend this basic configuration mechanism such that several signature variants can be specified for a particular node type, while each variant is valid in a specific context. A context is defined by a formal parameter exposed by an edit operation signature.

Technically, the signature-generation is performed by iteratively annotating the nodes of an ASG. Path expressions are evaluated by a generic path engine that exploits the reflection mechanism of the underlying modeling framework. An annotation is a key-value pair being attached to a node by the configurable model annotator. The annotation key `SIGNATURE` serves as pre-defined keyword; the value of this annotation will be finally incorporated in the hash value that represents the signature of a node.

Example 6.1 (Signature-based detection of modified arguments)
The pseudo code fragment shown in Listing 6.1 illustrates how the SiDiff annotation procedure and signature generation can be configured for UML class diagrams.

The imperative section (lines 1 and 2) specifies that an ASG shall be traversed two times in a top-down manner, following its containment structure. In the first traversal, an annotation at key `PATH` is to be attached to each node of the ASG. In the second traversal, each node is to be annotated with an annotation value which is later accessible by key `SIGNATURE`.

The subsequent declarative section defines how the annotation values are to be derived from specific element properties or other annotations:

For all nodes of type *Element*, which is a common supertype of all node types defined by the UML superstructure, the value to be annotated at key `PATH` is to be computed by concatenating the `PATH` annotation of the parent node and the name of the concrete type of a node (lines 6-8). This `PATH` value is the only value which is to be incorporated into the signature for nodes of type NamedElement (s. lines 9-11). The signature is valid in any context, i.e. for each formal object parameter of type *Element* or one of its subtypes. Lines 14-17 show that annotation specifications can be overridden for

```
1  traverse top-down "PATH";
2  traverse top-down "SIGNATURE";
3
4  // Basic annotations
5  NodeType Element
6    annotation:
7      key = PATH,
8      value = (self.parent.PATH) ∘ (self.type.name);
9    annotation:
10     key = SIGNATURE,
11     value = self.PATH;
12
13 // Override PATH annotation for NamedElements
14 NodeType NamedElement
15   annotation:
16     key = PATH,
17     value = (self.parent.PATH) ∘ (self.name);
18
19 // Signature being valid in a specific context
20 NodeType Class
21   annotation:
22     context = createSubclass(superClass,name)::superClass,
23     key = SIGNATURE,
24     value = (self.neighbors.PATH) ∘ (self.children.PATH);
```

Listing 6.1: Sample configuration of a signature-based detection of modified arguments in UML class diagrams

subtypes of *Element*; for all nodes of type *NamedElement*, the value of `PATH` is to be computed by concatenating the parent `PATH` and the value of the local attribute *name*. The signature configuration is inherited from type *Element*. Lines 20-24 illustrate how to specify a signature which is only valid in a certain context. This signature shall be used to perform the modification check when a node of type *Class* occurs as argument superClass of operation createSubclass(superClass, name). This specific signature value is computed by concatenating `PATH` annotations of all neighbors and all children of a particular *Class* node.

Comparison-based detection of modified arguments. Defining a specific signature for each formal parameter being exposed by edit operations is a powerful mechanism to configure the detection of modified operation arguments. However, signature specifications might still be too coarse-grained to express certain "conflict patterns", as we can only integrate values of *all* nodes being collected via a specific ASG axis in a uniform way. Moreover, we cannot exploit information about corresponding elements in v_2 and v_0, e.g. to express that classes c in v_0 and c' in v_2 are associated to "the same" set of classes, while these classes do not necessarily share the same signature values.

Thus, the basic idea is to exploit the comparison facilities of a similarity based matcher and to use comparison heuristics in order to configure a check for modified

operation arguments. Adopting the SiDiff approach to similarity-based matching, the configuration data defines, for each node type defined by a meta-model, a set of properties which are relevant for the similarity and, for each property, a *compare function* which computes the similarity of two values of this property [2]. A compare function takes two nodes of the same type as input and returns a value between 0 and 1, where 0 means "no similarity" and 1 means that the two nodes are identical w.r.t. to the considered property. Let A and B be two models being typed over the same type graph, then a compare function for a property p is defined as

$$compare_p : A_N \times B_N \rightarrow [0,1] \tag{6.2.1}$$

with $[0,1] = \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$.

While a similarity-based matcher generally computes the overall similarity of two elements (cf. Section 2.3.1), which is defined in SiDiff as the weighted arithmetic mean of the similarities of all relevant properties [138], we are only interested in the fact whether an element has been modified or not. To that end, we define another function that converts the result of a compare function into a boolean value (*true* = "modified", *false* = "unmodified") in a straightforward way:

$$convert : [1,0] \rightarrow \mathbb{B}, \text{ with } convert(x) = \begin{cases} false, & \text{if } x = 1 \\ true, & \text{if } x \neq 1. \end{cases} \tag{6.2.2}$$

Finally, an element e_2 in v_2 is considered to be modified against its corresponding element e_0 in v_0 if any of its properties being considered as relevant has been modified:

$$Mod_{e_0,e_2} = \bigwedge_{p \in P} convert(compare_p(e_0, e_2)) \tag{6.2.3}$$

where P is the set of relevant properties and $compare_p$ is the selected compare function for property p.

Example 6.2 (Comparison-based detection of modified arguments)
Table 6.1 shows a fine-grained configuration for a comparison-based check of UML classes against modification. Similar to the signature-based configuration, a context can be defined in which a certain check is to be performed. Here, the check shall only be performed if a class occurs as parameter superClass of the operation create-Sublass(superClass,name). Properties being relevant in this context are summarized by column 3. Consequently, a modified argument flag will be issued for conflict ③ of Example 2.4 because class *Event* has different attributes in v_0 and v_2. No warning will be generated in case of a local property change such as a simple change of a class name, turning a class into an abstract class, etc.

6.3 Interactive Application of Edit Scripts

Our analysis in Section 2.2 has shown that the application of a patch on a given target model can fail for various reasons, a user must be enabled to analyze and resolve

NodeType	Context	Criterion
Class	createSubclass(superClass,name)::superClass	Attributes match Operations match Associated classes match Generalization targets match

Table 6.1: Sample configuration of a comparison-based detection of modified arguments in UML class diagrams

the problems somehow. The only solution is often to modify the target model, e.g. by creating missing model elements or modifying it in a way that a precondition of an edit step is fulfilled. In this section, we present a graphical user interface (GUI) design which enables developers to apply consistency-preserving edit scripts to a target model in a controlled, interactive way. We begin with traditional patching scenarios in which original and target model are unrelated and do not have a common base version (s. Section 6.3.1). Extensions to the basic GUI which specifically address the updating of workspace copies are described in Section 6.3.2.

6.3.1 Patch Application without Common Base Version

When the operation arguments have been determined initially, an edit script can be interactively applied. The graphical user interface of our interactive patching tool is shown in Figure 6.5. It is partitioned into four subwindows:

- The top left subwindow shows the edit script which is to be applied. The edit steps are ordered in an arbitrary order which is consistent with the partial order of the script. Dependencies can be shown if desired, and successfully executed or ignored edit steps can be hidden.
- The top right subwindow shows the current state of the target model in a standard editor for this modeling language.
- The bottom left subwindow shows a status protocol of executed edit steps.
- The bottom right subwindow is a property editor which shows details of the currently selected edit step. Details of report entries of the status protocol shown in the bottom left subwindow can also be inspected in the properties view.

An edit script can be applied on the target model step by step. If desired, all edit steps which are principally applicable can be executed automatically. Moreover, the GUI enables developers to control the effects of each step, the intermediate state of the target model and arguments of edit steps can be modified at any time.

Figure 6.5 shows an intermediate state of applying the edit script of our running example of scenario SC4. Note that the initial edit script obtained from a comparison of the original and the changed version of our sample flight ticketing system has been

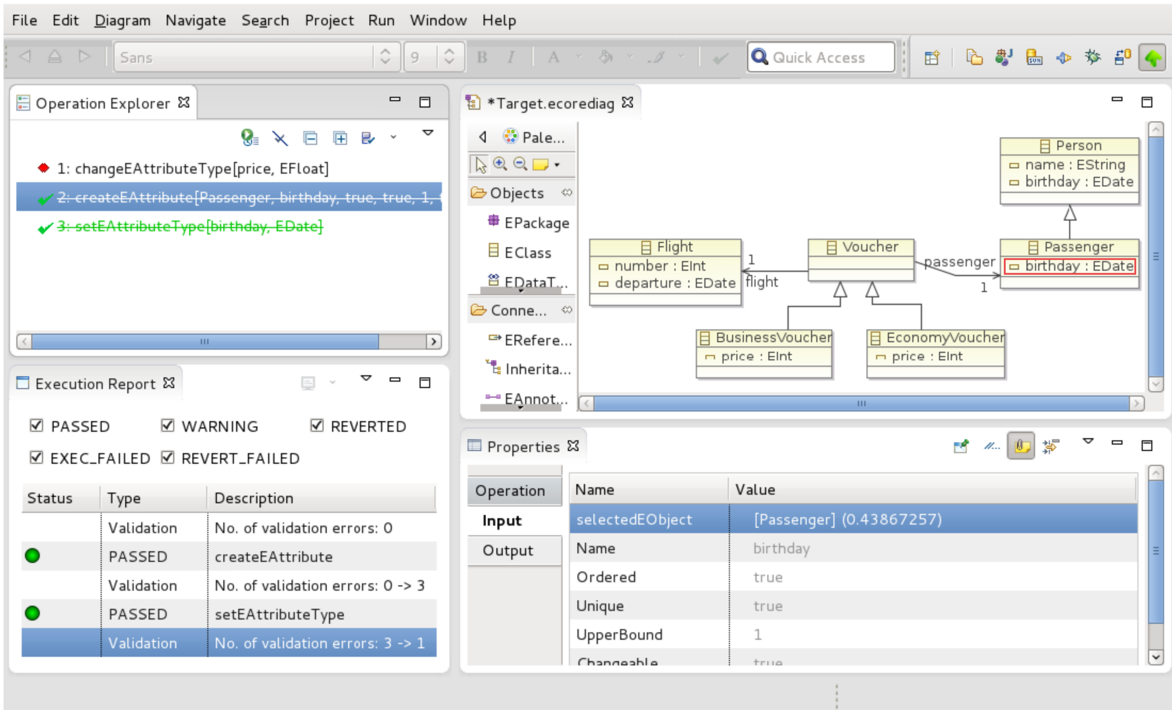


Figure 6.5: Graphical user interface of an interactive patch application tool

edited by the patch creator as described in Section 6.1.2, i.e. the edit step that extracts the class *Schedule* has been removed. Two edit steps, namely the edit steps labeled as 2 and 3 introducing the attribute *birthday* in class *Passenger* have been already executed.

Two kinds of warnings can be generated as a result of the execution of edit steps:

1. If the reliability of a correspondence used to resolve an operation argument is below a given threshold the user is warned.

   In our example, the reliability of the correspondence which has been established between the class *Passenger* of the original model and the class *Passenger* of the target model is estimated with a bad value of about 44% (s. bottom right subwindow).

2. CPEOs as assumed by our approach preserve the degree of consistency according to the effective meta-model, which guarantees that the target model remains displayable in a standard editor. Additionally, the modified target model is validated against the perfect meta-model after the execution of each edit step. Validation errors are a second kind of warning that can be generated as a result of the execution of an edit step.

   In our example, the name of the attribute *birthday* is not unique within the inheritance closure of class *Passenger*; the execution report finally reports one

validation error which can be inspected in the properties details view. The problem reported here can be solved by declaring the respective edit step to be skipped such that the attribute birthday will not be created at all. Analogously to the editing of edit scripts by a patch creator (s. Section 6.1.2), the exclusion of an edit step causes all dependent edit steps to be excluded from the execution of the edit script. All these edit steps are "marked" not to be executed by the patch engine. They are not completely deleted from the patch but can be re-activated on demand.

As shown in Figure 6.5, edit step 1 of our edit script initially can not be executed at all. A closer inspection of the arguments of this edit step using the property editor reveals that one operation argument could not be found during reference resolution, namely the attribute *price* whose datatype shall be changed to *EFloat*. This is due to an ambiguity which could not be resolved by the matcher; while the attribute *price* occurs once in the original model, there are two occurrences of the "identical" attribute *price* in the target model. Thus, no correspondence was established by the matcher. Missing operation arguments have to be selected by the user. If required, missing model elements can be created in the standard editor and afterwards be selected as operation arguments. In our example, the attribute *price* could be pulled up [107] to the common superclass *Ticket* and subsequently be selected as operation argument.

6.3.2 Updating Workspace Copies by Patching

The GUI of a tool which implements our approach to update workspace copies by patching is an extension of the basic patch application GUI introduced in the previous section. Analogously to the patch application without common base version, the GUI is partitioned into four subwindows and enables developers to execute an edit script step by step on the workspace model v_2, to control the effects of each step, and to modify the intermediate state of the target model at any time. The only extension w.r.t. to the basic patch application GUI is that another kind of warning is flagged which prevents workspace changes from being blindly overridden.

Figure 6.6 shows an intermediate state of performing a workspace update for our Example 2.4 in an interactive way. A screencast that demonstrates how to use the tool in this scenario can be found at the SiLift website[2]. In the intermediate state of Figure 6.6, two repository changes which are represented by edit steps labeled as 1 and 2 have been successfully applied to the workspace model. Note that edit step 1 has been applied first since edit step 2 depends on edit step 1. In Figure 6.6, this dependency is visually indicated as follows: The tree item representing edit step 2 has been expanded such that all edit steps to which edit step 2 has a direct dependency (which is only step 1) are shown.

As a result of edit step 1, a validation error against the perfect consistency is flagged as a warning. This indicates conflicting situation ② of Example 2.4. Thus, attribute

[2]http://pi.informatik.uni-siegen.de/Projekte/SiLift/ase2014.php

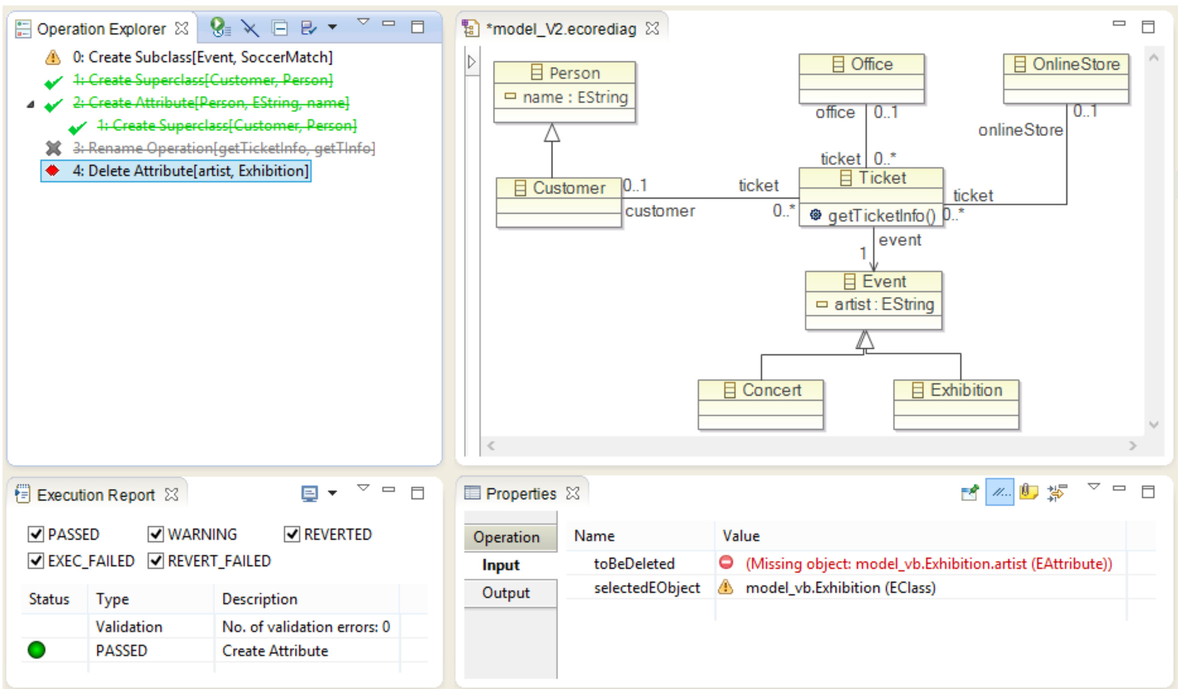


Figure 6.6: Graphical user interface of an interactive workspace update tool

Customer.name has been manually deleted such that attribute *name* is uniquely defined within the inheritance hierarchy of class *Person*.

A warning against blindly overwriting local changes has been initially attached to edit step 3. This warning represents conflicting situation ① of Example 2.4. The conflict has been resolved by choosing the workspace alternative of the concurrent name change, i.e. edit step 3 has been skipped, as indicated in Figure 6.6. Thus, the name of operation *Ticket.getInfo* is finally changed to *"getTicketInfo"*.

The remaining edit steps 0 and 4 represent consequences of the conflicting situation ③ of Example 2.4 for which a developer finally has to find a creative solution (cf. Section 2.1.5).

- Edit step 4 cannot be performed because attribute *Exhibition.artist* serving as argument of this edit step could not be resolved in the target model. The reason for this is that it has been pulled up to class *Event* in the workspace model.

- Edit step 0 leads to a warning against blindly overwriting local changes if the check procedure is configured according to the sample signature-based configuration of Example 6.1: The new subclass of *Event* called *SoccerMatch*, which is to be created by edit step 0, implicitly inherits the attribute *artist* which has been pulled up to class *Event*.

CHAPTER 7

Creation of a Set of Consistency-preserving Edit Rules

Our approach to model differencing is adapted to a given modeling language by providing edit rules for this language. This set of edit rules must be properly designed and meet certain correctness criteria to which we refer to as *soundness* and *completeness* of a set of edit rules. Section 7.1 clarifies these criteria and thus identifies requirements for a set of *mandatory edit rules*; these rules must be provided in order to guarantee that correct edit scripts are generated. While complete sets of mandatory rules can be easily defined based on primitive graph operations, engineering a mandatory set of consistency-preserving edit rules (CPERs) is much more challenging. Moreover, in contrast to primitive graph edit operations, CPERs are not generic, but have to be individually engineered for a given modeling language and the effective meta-model of a standard model editor for this language. A general process for creating a set of CPERs is shown in Figure 7.1:

Step 1: We assume that the effective level of consistency which is to be preserved by CPERs is expressed by an effective meta-model (cf. Section 3.2). The effective meta-model is typically constructed by creating a copy of a standard (or perfect) meta-model and dropping all consistency constraints except the relevant ones.

Step 2: Comprehensive languages such as the UML lead to large sets of CPERs, even if the required degree of consistency is reduced to the effective meta-model. The manual specification of such sets of edit rules is very tedious and prone to errors. To that end, we propose a semi-automated approach to derive a mandatory set of

CPERs for a given meta-model. The idea is to generate as many rules as possible, the effective meta-model serves as input of this generation process. Basic design decisions concerning our approach to generate CPERs are discussed in Section 7.2, while Section 7.3 presents a precise specification of a CPER generation algorithm. Our approach is constructive in the sense that it ensures the completeness (s. Section 7.4) and soundness (s. Section 7.5) of a generated rule set which thus serves as an initial version of a set of mandatory edit rules. Since our technique to operation detection starts from a given matching, the properties of the model matcher which is used in the first step of our differencing pipeline of Figure 1.2 must be, in general, consistent with the edit rules which are used by the operation recognition in the third step of the pipeline. Thus, Section 7.6 discusses the requirements which result from our generated CPERs.

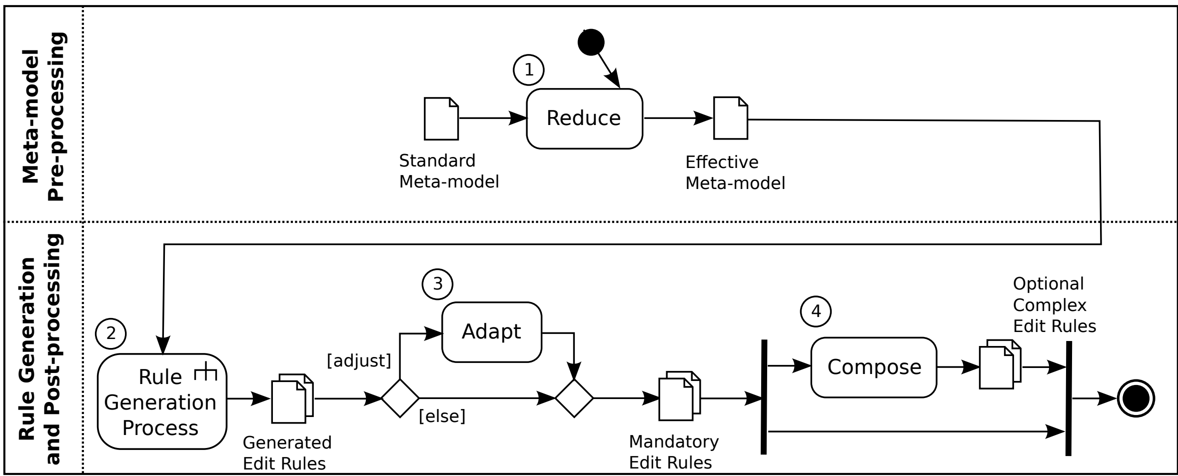


Figure 7.1: Process for creating a set of consistency-preserving edit rules

Step 3: The level of consistency which is guaranteed to be preserved by generated rules includes a restricted set of multiplicity invariants and all basic consistency-constraints except *no-containment-cycles*. Moreover, our approach is not yet capable of interpreting additional well-formedness rules attached to a given meta-model. In order to deal with these restrictions, the generated set of edit rules may have to be manually adapted in an additional post-processing step (s. Section 7.7). Typically, some of the generated edit rules have to be merged or complemented by additional application conditions.

Step 4: Obviously, the set of semantically rich complex edit rules which are *optional* but considered useful for a given modeling language has to be engineered manually. Complex edit rules are compositions of mandatory or other complex edit rules. We support two kinds of compositions; the sequential composition of edit rules, i.e. an application of the composite rule has the same effect as applying its

components in sequence, and the composition of edit rules to amalgamated rules (cf. Section 3.3.2). Complex edit rules are optional extensions of the mandatory rule set. They are consistency-preserving by construction.

7.1 Mandatory Edit Rules

Edit scripts generated with our approach are correct only if *all* change actions observed in a low-level difference are grouped to semantic change sets during operation recognition. To this end, a set of edit rules must be (a) *complete* in the sense that every (consistency-preserving) modification of a model can be expressed using edit rules available in this set, and (b) *sound* in the sense that every modification can be expressed without causing transient effects.

Definition 7.1 (Completeness of a set of edit rules)
Let $L(TG)$ be the set of models (or "language") induced by a type graph TG, then a set R of edit rules typed over TG is complete if and only if R has the following properties:

(1) **instance-generating**: Every consistent model $M \in L(TG)$ can be constructed starting from the empty model ϵ by exclusively using edit rules available in R.

(2) **instance-deleting**: Every consistent model $M \in L(TG)$ can be reduced to the empty model ϵ by exclusively using edit rules available in R.

In addition to the completeness criteria of Definition 7.1, our method of operation recognition requires any model modification to be expressible without transient model elements. Given a model M being modified by a sequence of rule applications, then for all sequential pairs of rule applications $t_1 = M1 \overset{r_1,m_1,n_1}{\Longrightarrow} M2$ and $t_2 = M2 \overset{r_2,m_2,n_2}{\Longrightarrow} M3$ within this sequence, one of the following conditions must hold (1) t_1 and t_2 do not produce transient effects at all; otherwise (2) t_1 and t_2 must be inverse to each other, or (3) the transient effect is avoidable in the sense that there is a rule application $t_3 = M1 \overset{r_3,m_3,n_3}{\Longrightarrow} M3$ yielding the same result as the sequence $t_1; t_2$. Consequently, the set R of edit rules must be properly designed.

Definition 7.2 (Soundness of a set of edit rules)
Let $L(TG)$ be the set of models induced by a type graph TG, then a set R of edit rules typed over TG is called *sound* if for all pairs (r_1, r_2) of rules $r_1, r_2 \in R$, one of the following statements is true:

(1) Applying r_2 after r_1 cannot lead to transient effects at all, i.e. for all models $M \in L(TG)$ and for all elements $e_c, e_d \in M$, an element e_c created by r_1 cannot be deleted by r_2, and an element e_d deleted by r_1 cannot be re-created by r_2.

(2) Rules r_1 and r_2 are inverse to each other, i.e. for each action $a \in r_1$ there has to be a corresponding inverse action $a^{-1} \in r_2$ such that a^{-1} reverses the effect of a:

For each *create node (edge) action* there has to be a *delete node (edge) action* of the same type and for each *attribute value change action* there has to be again a value change action of the same attribute.

(3) A potential transient effect is avoidable, i.e. there is a rule $r_3 \in R$ summarizing all effects of applying first r_1 and then r_2 in a suitable dependency relation. Furthermore, r_3 summarizes all pre-conditions of r_1 and r_2.

In a first approximation, a minimal set of edit rules which is sound and complete constitutes a set of *mandatory edit rules* which is required by our approach; every model modification can be expressed using edit rules available in this set and without causing transient effects.

However, an additional requirement for generating correct edit scripts is that the properties of the model matcher which is used in the first step of our differencing pipeline of Figure 1.2 must be consistent with the set of mandatory edit rules. The reason for this general requirement is that, given two models A and B which are to be compared with each other, the matcher determines the corresponding elements of A and B. In a difference derived from a model matching, corresponding elements will not be considered as deleted or created. Thus, given a matching $M_{A,B} : A \rightsquigarrow B$ and a mandatory edit rule set R, we have to find a transformation sequence

$$t_1 = A \overset{r_1,m_1,n_1}{\Longrightarrow} A_1, ..., t_q = A_{q-1} \overset{r_q,m_q,n_q}{\Longrightarrow} B \text{ with}$$

$r_i \in R$ and $m_i(Del_{r_i}) \cap dom(M_{A,B}) = \emptyset$ for all $i \in \{1, ..., q\}$, i.e. corresponding elements must be preserved instead of being deleted and later on re-created.

The properties which are expected from a model matcher can only be specified precisely, if precise specifications of our mandatory edit rules are readily available. Thus, we will discuss this requirement for our generated CPERs in more detail later, after having clarified the design and generation of CPERs.

7.2 Basic Design Decisions

In this section, we present basic design decisions concerning our approach to generate CPERs. We define four kinds of edit rules to which we refer as *creation*, *deletion*, *move* and *change* rules. We informally describe the design of each of these kinds of edit rules, details of our rule generation procedure will be presented in Section 7.3.

Creation and deletion rules. In order to minimize the number of potential transient effects, a basic design decision of our approach is that a mandatory rule set R must be instance-generating by exclusively using *creation* rules available in this set. A creation rule r is an edit rule which is implemented by creation actions only, i.e. we have $Del_r = \emptyset$. Creation rules can be further classified into two kinds of rules:

- **Node creation** rules basically create a new ASG node and connect it immediately to its container. All direct and indirect *mandatory children* are also created, i.e. we create a *minimal subtree*. Moreover, each node in this tree structure is immediately connected to its *mandatory neighbors*. Mandatory children and neighbors finally lead to a *minimal graph pattern* being created in a single step. This pattern is minimal in the sense that there is no smaller graph pattern which can be created in a consistency-preserving edit step.

- **Edge creation** rules create a new *non-containment* edge in the ASG.

For each creation rule r there must be an inverse *deletion* rule r^{-1} (with $Cre_{r^{-1}} = \emptyset$) such that R is instance-deleting. Consequently, deletion rules can be classified into two kinds of rules:

- **Node deletion** rules basically delete an ASG node together with its containment edge. All direct and indirect *mandatory children* are also deleted, i.e. we delete a *minimal subtree*. Moreover, each node in this tree structure is disconnected from its *mandatory neighbors*. Mandatory children and neighbors finally lead to a *minimal graph pattern* being deleted in a single step. This pattern is minimal in the sense that there is no smaller graph pattern which can be deleted in a consistency-preserving edit step.

- **Edge deletion** rules delete a *non-containment* from the ASG.

Note that the creation rules described above differ significantly from their primitive counterparts: Consistency-preserving node creations (deletions) usually comprise a set of primitive ASG operations in order to create (delete) minimal graph patterns: Nodes are only created (deleted) together with an incoming containment edge, all direct and indirect mandatory children are also created (deleted), and each node in such a minimal subtree is immediately connected to (disconnected from) its mandatory neighbors. Moreover, attribute values of created nodes are also set within a node creation rule since we conceptually treat them like mandatory neighbors. Consistency-preserving edge creations (deletions) are only defined for non-containment edge types; containment edges can only be created (deleted) together with the contained node[1].

Move and change rules. Mainly for reasons of convenience, the set of generated CPERs should also contain the following kinds of edit rules:

- **Move** rules that shift an ASG node to another container, i.e. rules that summarize the deletion and creation of a containment edge of a particular type referencing a particular child node.

- **Change** rules "exchanging" the target node of a non-containment edge, i.e. rules that summarize the deletion and creation of a non-containment edge (or an edge to a data node) of a particular type originating from a particular source node.

[1]Move rules provide another option for the deletion and creation of containment edges; they will be introduced later in this section.

Note that the effect of move and change rules can also be achieved by using creation and deletion rules, however very inconveniently. Relocations of ASG nodes which are performed by move rules can be achieved by deleting the node which is to be moved and by re-creating this node as a child node of the new parent. Exchanging the target node of a non-containment edge can be achieved by a sequence of edge deletions and creations. However, descriptions of model modifications based on such sequences of deletions and creations are very inconvenient if reported to difference tool users. This inconvenience is further aggravated by the fact that consistency-preserving creation (deletion) rules often lead to the creation (deletion) of larger fragments of an ASG.

7.3 Generation of Consistency-preserving Edit Rules

How instances of a meta-class can be edited depends on their context in the ASG and the mandatory or optional parts of this context as defined by the respective meta-model. To that end, our CPERs are determined by the occurrence of specific meta-model patterns. The idea is to associate to a particular meta-model pattern a CPER that modifies an instance of this pattern under certain conditions.

We support a restricted set of multiplicity invariants which will be clarified in Section 7.3.1. Subsequently, we describe how to generate a set $\mathcal{R}$ with

$$\mathcal{R} = \mathcal{R}_{Cre} \cup \mathcal{R}_{Del} \cup \mathcal{R}_{Chn} \cup \mathcal{R}_{Mov}$$

of consistency-preserving edit rules. The generation of creation and deletion rules leading to rule sets $\mathcal{R}_{Cre}$ and $\mathcal{R}_{Del}$ is addressed in Section 7.3.2, while the generation of change and move rules leading to rule sets $\mathcal{R}_{Chn}$ and $\mathcal{R}_{Mov}$ is covered by Section 7.3.3. In addition to restricted multiplicity invariants, generated CPERs preserve basic consistency constraints by construction (s. Section 7.3.4).

7.3.1 Preparations and Prerequisites

In this section, we clarify the conditions which must be fulfilled before the CPER generation (step ② in Figure 7.2) starts. This set of conditions finally leads to our notion of a *type graph with restricted multiplicities*. If any of these conditions is violated, certain multiplicity constraints must be relaxed (step ① in Figure 7.2). Consequently, the set of generated edit rules must be post-processed manually in step ③ of the overall process for creating a set of consistency-preserving edit rules shown in Figure 7.1.

Multiplicity patterns. In general, we support a restricted set of multiplicity patterns, i.e. combinations of multiplicities at edge types being declared as opposite to each other. An overview is shown in Figure 7.3. For containment edge types, we support multiplicity patterns MP_2 ([0..1] to $[k..l]$) and MP_3 ([1..1] to $[k..l]$). Pattern MP_1 ([0..$*$] to $[k..l]$) is not possible due to the *at-most-one-container* constraint. For

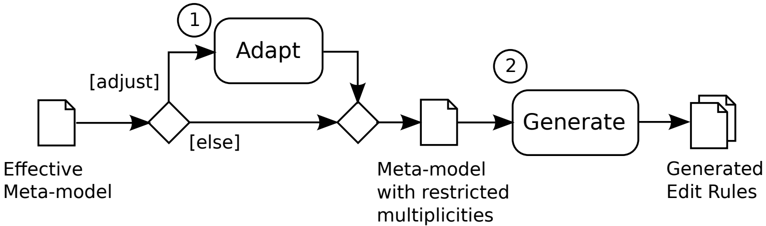


Figure 7.2: Edit rule generation process

non-containment edge types, we support multiplicity patterns MP_1 and MP_2. Pattern MP_3 is not supported since we would have to create (delete) a pair of nodes which mutually are mandatory neighbors to each other in a single step. As we will see in Section 7.3.2, this case is not supported by our node creation (deletion) rules. Definition 7.3 summarizes the conditions for a type graph in which all multiplicity patterns are properly defined.

Multiplicity pattern	**Containment edge types**	**Non-containment edge types**
MP_1: [0..*] to [$k..l$]	-	A * — k..l B A → k..l B
MP_2: [0..1] to [$k..l$]	A 0..1 ◆— k..l B A ◆→ k..l B	A 0..1 — k..l B
MP_3: [1..1] to [$k..l$]	A 1 ◆— k..l B	(not supported)

Figure 7.3: Supported combinations of multiplicity invariants (multiplicity patterns)

Definition 7.3 (Properly defined multiplicity patterns)
Given a type graph $TG = (T, I, A, C, OE, mult)$, multiplicity patterns in TG are properly defined if we have one of the following combinations of multiplicity invariants for all pairs of opposite edge types $(et_i, et_j) \in OE$:

- If $et_i \in C$ and $et_j \notin C$, we have

  MP_2: $mult(et_i) = [0..1]$ and $mult(et_j) = [k..l]$, or

  MP_3: $mult(et_i) = [1..1]$ and $mult(et_j) = [k..l]$.

- If $et_i \notin C$ and $et_j \notin C$, we have

  MP_1: $mult(et_i) = [0..*]$ and $mult(et_j) = [k..l]$ (or vice versa), or

MP_2: $mult(et_i) = [0..1]$ and $mult(et_j) = [k..l]$ (or vice versa).

Concerning multiplicities at the opposite end of edge types for which no opposite edge type is explicitly specified by the meta-model, we choose "default" interpretations which are commonly used in modeling frameworks such as EMF [88, 228]: If a containment edge type has no opposite edge type, then the multiplicity invariant at the opposite end is conceptually considered as [0..1]. If a non-containment edge type has no opposite edge type, then the multiplicity invariant at the opposite end is conceptually considered as $[0..*]$. Both interpretations are also illustrated in Figure 7.3.

Multiplicities at containment edge types. Consider the sample meta-model pattern of Figure 7.4. The containment edge type b has a multiplicity with property *required*, i.e. $k > 0$. Moreover, the target node type B of b has two subtypes, namely C and D. Assume that we have $k = 2$. Then, in a consistent instance graph, a node of type A must have two mandatory children, which can be achieved in three different ways; *i)* both child nodes are of type C, *ii)* both child nodes are of type D, or *iii)* one child node is of type C and the other one of type D. The number of possible valid instance structures grows combinatorially with the lower bound k and the number of (indirect) subtypes of B.

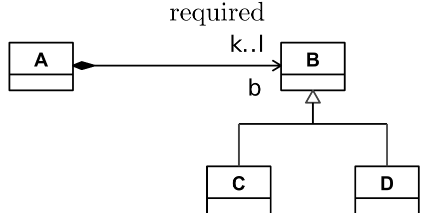


Figure 7.4: Example of an unsupported multiplicity at containment edge types

Since our node creation rules are designed to create all (indirect) mandatory children of a node which is to be created (cf. Section 7.3.2), a dedicated variant of a node creation rule would be required for each of these possible instance structures. The reason for this is that all nodes which are to be created by a Henshin rule must be concretely typed (s. Section 3.3.2). Thus, we currently do not support multiplicities with property *required* for containment edge types of which the target node type has subtypes.

Note that multiplicities having property *required* are not a problem for non-containment edge types of which the target node type has subtypes. They do not lead to a large number of variants of edit rules since edit rules which create instances of non-containment edge types always assume that the target node of the non-containment edge which is to be created already exists. In the respective edit rules, these target nodes can be typed over the general node type, even if this node type is abstract.

Definition 7.4 (Properly defined multiplicities at containment edge types)
Let $TG = (T, I, A, C, OE, mult)$ be a type graph according to Definition 3.5. Multi-

plicities at containment edge types in TG are properly defined, if all containment edge types which have a target node type that has subtypes specify a multiplicity [0..∗]:

$$\text{For all } et_c \in C : sub(et_c.tgt) \neq \emptyset \implies mult(et_c) = [0..*]$$

Minimal subtree definitions. As already mentioned in Section 7.2, all direct and indirect *mandatory children* of a node are also created (deleted) by our node creation (deletion) rules. In other words, we create (delete) a tree structure which can be considered as *minimal subtree* w.r.t. the containment hierarchy of the model in which this tree structure is created (deleted). Definition 7.5 introduces our notion of a *minimal subtree definition* in a meta-model represented as type graph.

Definition 7.5 (Minimal subtree definition)
Let $TG = (T, I, A, C, OE, mult)$ be a type graph according to Definition 3.5. A subgraph $MSD \subseteq T$ is called a *minimal subtree definition* if there is a distinct node type $r \in MSD_N$ such that the following three conditions hold:

(1) Node type r is (transitively) connected to all other node types in MSD (given by the set $MC = MSD_N \setminus \{r\}$) via containment edge types having multiplicity property *required*.

(2) There is no other node type $c \notin MC$ to which r is (transitively) connected via containment edge types having multiplicity property *required*.

(3) There is no other node type $p \in (T_N \setminus MSD_N)$ which is connected to r via a containment edge type having multiplicity property *required*.

Let MSD_{TG} be the set of all minimal subtree definitions induced by TG, then the function $msd : T_N \to \mathcal{P}(MSD_{TG})$ identifies for each node type $nt \in T_N$ the set of minimal subtree definitions of which nt is a part of.

Remark. Note that the set MC of Definition 7.5 may be the empty set, i.e. we consider a single node type which adheres to conditions (1), (2) and (3) as the smallest minimal subtree definition.

Minimal subtrees have to be properly defined. Thus, some combinations of multiplicities in certain meta-model patterns are not supported by our edit rule generation procedure. In the following, we describe these unsupported meta-model patterns and briefly outline the reasons why they are not supported:

- We do not allow cyclic containment structures in which each containment edge type has a multiplicity property *required*. This restriction includes containment loop edge types having a multiplicity property *required*. In other words, minimal subtree definitions must form a directed acyclic graph (DAG).

  Cyclic containment structures in meta-models as described above may lead to type graphs which are not finitely satisfiable [234]. They are problematic for our

CPER generation because the procedure that supplements a basic node creation rule such that mandatory children are also created (s. Section 7.3.2) then runs into an endless loop.

- Moreover, we do not allow non-containment edge types which connect node types being part of the same minimal subtree definition and which have a multiplicity property *required*. Note that this restriction includes non-containment loop edge types having a multiplicity property *required*.

  Meta-model patterns as described above lead to valid instance graphs in which a node may have a mandatory neighbor which is part of the same minimal subtree. Such instance structures can only be created in a consistency-preserving way if the subtree is created *and* its nodes are connected in a single edit step. This is not possible with our node creation rules since we assume that all mandatory neighbors of a node which is to be created already exist (s. Section 7.3.2)

Definition 7.6 summarizes the conditions for a type graph in which all minimal subtree definitions are properly defined.

Definition 7.6 (Properly defined minimal subtree definitions)
Let $TG = (T, I, A, C, OE, mult)$ be a type graph according to Definition 3.5. MSD_{TG} refers to the set of all minimal subtree definitions induced by TG, and $msd : T_N \rightarrow \mathcal{P}(MSD_{TG})$ identifies for each node type $nt \in T_N$ the set of minimal subtree definitions of which nt is a part of (s. Definition 7.5). Let further $msd_I : T_N \rightarrow \mathcal{P}(MSD_{TG})$ be a function which is an extended variant of msd in that it identifies for each node type $nt \in T_N$ the set of minimal subtree definitions of which nt or any of its subtypes is a part of, i.e.

$$msd_I(nt) = msd(nt) \cup \left(\bigcup_{nt' \in allsub(nt)} msd(nt') \right)$$

Minimal subtree definitions are properly defined if TG adheres to the following two conditions:

(1) For each cycle of containment edge types $(et_1, et_2, ..., et_n)$, i.e. $et_i \in C$ and $1 \leq i \leq n$, there must be at least one edge type $et_i \in \{et_1, et_2, ..., et_n\}$ with $et_i.lb = 0$.

(2) For each non-containment edge type $et \in T_E$ $(et \notin C)$ with $et.lb > 0$, we have $msd_I(et.src) \cap msd_I(et.tgt) = \emptyset$.

Relationships between atomic subtree definitions. Furthermore, we do not allow certain *cyclic relationships* between minimal subtree definitions. Examples serving as minimal representatives of these unsupported meta-model patterns are illustrated by cases *(C.i)* and *(C.ii)* in Figure 7.5. *MSD1* and *MSD2* are meant to represent minimal subtree definitions which, in general, may include several node types. Connections which are visually represented as dashed non-containment edge types are meant to be

a path $(et_1, et_2, ..., et_n)$ of an arbitrary number of non-containment edge types in which each edge type has a multiplicity property *required*. Connections which are visually represented as containment edge types are meant to be a path which consists of a single containment edge type.

- Meta-model pattern *(C.i)* may lead to valid instance graph structures in which we have a cyclic relationship between mandatory neighbors. Such instance structures cannot be created by our CPERs because mandatory neighbors are intended to be created independently of each other.
- Meta-model pattern *(C.ii)* may lead to valid instance graph structures in which a node serves as parent *and* mandatory neighbor of its child node. These instance structures cannot be created with our CPERs because they are intended to be matched injectively.

Definition 7.7 summarizes the conditions for a type graph in which all cyclic relationships between minimal subtree definitions are properly defined.

Definition 7.7 (Properly defined cyclic relationships between minimal subtree definitions)
Let $TG = (T, I, A, C, OE, mult)$ be a type graph according to Definition 3.5. Let further MSD_{TG} refer to the set of all minimal subtree definitions in TG. Cyclic relationships between minimal subtree definitions are properly defined if TG adheres to the following two conditions:

(1) For each pair (MSD_i, MSD_j) of minimal subtree definitions $(MSD_i, MSD_j \in MSD_{TG}, i \neq j)$ and for each pair (ncp_{ij}, ncp_{ji}) where

  - ncp_{ij} is a path $(et_{ij,1}, ..., et_{ij,n})$ of non-containment edge types with $n \geq 1$ and $et_{ij,1}.src_I \cap MSD_i \neq \emptyset$ and $et_{ij,n}.tgt_I \cap MSD_j \neq \emptyset$,
  - ncp_{ji} is a path $(et_{ji,1}, ..., et_{ji,m})$ of non-containment edge types with $m \geq 1$ and $et_{ji,1}.src_I \cap MSD_j \neq \emptyset$ and $et_{ji,m}.tgt_I \cap MSD_i \neq \emptyset$,

  there is at least one edge type $et \in (\{et_{ij,1}, ..., et_{ij,n}\} \cup \{et_{ji,1}, ..., et_{ji,m}\})$ with $et.lb = 0$.

(2) For each pair (MSD_i, MSD_j) of minimal subtree definitions $(MSD_i, MSD_j \in MSD_{TG}, i \neq j)$ and for each pair (ncp, c) where

  - ncp is a path $(et_1, ..., et_n)$ of non-containment edge types with $n \geq 1$ and $et_1.src_I \cap MSD_i \neq \emptyset$ and $et_n.tgt_I \cap MSD_j \neq \emptyset$,
  - c is a containment edge type with $c.src_I \cap MSD_j \neq \emptyset$ and $c.tgt_I \cap MSD_i \neq \emptyset$,

  there is at least one edge type $et \in \{et_1, ..., et_n\}$ with $et.lb = 0$.

Finally, we do not allow certain *parallel relationships* between minimal subtree definitions. Examples serving as minimal representatives of these unsupported meta-model

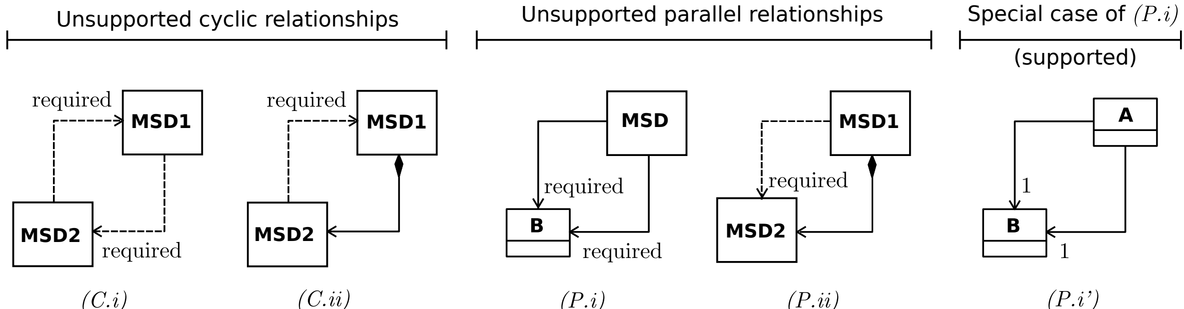


Figure 7.5: Unsupported relationships between minimal subtree definitions

patterns are illustrated by cases *(P.i)* and *(P.ii)* in Figure 7.5. Note that for *(P.i)* we do not support the general case but a restricted variant called *(P.i')* of this meta-model pattern. Again, *MSD*, *MSD1* and *MSD2* represent a minimal subtree definitions which, in general, may include several node types. In contrast, A and B refer to ordinary node types. Connections which are visually represented as dashed non-containment edge types are meant to be a path $(et_1, et_2, ..., et_n)$ of an arbitrary number of non-containment edge types in which each edge type has a multiplicity property *required*. Connections which are visually represented as ordinary non-containment edge types are meant to be a path which consists of a single non-containment edge type. Connections which are visually represented as containment edge types are meant to be a path which consists of a single containment edge type.

- Parallel relationships as illustrated by case *(P.i)* of Figure 7.5 are currently not supported since they would lead to a huge number of CPERs. Assume that we have two "parallel" edge types having lower bounds lb_1, lb_2 with $0 < lb_1 \leq lb_2$. In an instance graph, due to the *no-parallel-edges* constraint, we need at least lb_2 nodes being connected as mandatory neighbors such that lower bounds lb_1 and lb_2 are fulfilled. We may also have $lb_1 + lb_2$ distinct nodes serving as mandatory neighbors. In addition, we may have any number k with $lb_2 < k < lb_1 + lb_2$ of distinct nodes serving as mandatory neighbors. Thus, we already have $lb_1 + 1$ variants of different valid instance structures for two "parallel" edge types. Obviously, the number of possible combinations is much higher if we have more than two "parallel" edge types.

  Since our node creation rules are intended to be matched injectively, a dedicated variant of a node creation rule would be required to create each of these possible instance structures[2] (cf. Section 7.3.2). Thus, we only support a restricted variant of this meta-model pattern, namely two required parallel edge types having lower bounds $lb_1 = lb_2 = 1$ (s. case *(P.i')* in Figure 7.5). In order to support the special case *(P.i')*, only two variants of a node creation rule are required: One to fulfill the lower bounds with a single mandatory neighbor and one to fulfill the lower bounds with two distinct mandatory neighbors. Experience shows that

[2]The same argument applies to node deletion rules.

this special case is the only one that occurs frequently in effective meta-models, e.g. in terms of node types which are represented as visual edges in the external representation. The definition of the two parallel edge types *source* and *target* from *Transition* to *Vertex* in the meta-model for simple UML state machines of Figure 3.4 is an example of this.

- Parallel relationships as illustrated by case *(P.ii)* of Figure 7.5 may lead to valid instance structures in which a node being part of a minimal subtree instance of *MSD2* is a (indirect) child node *and* mandatory neighbor of another node or of two nodes being part of the same minimal subtree instance of *MSD1*.

  Such instance structures cannot be created with our CPERs: On the one hand, a parent node must be created first before the child node can be created in a subsequent edit step. On the other hand, the child node serving as mandatory neighbor must be created before it can be connected. This leads to a cyclic dependency of the respective rule applications.

Definition 7.8 summarizes the conditions for a type graph in which all parallel relationships between minimal subtree definitions are properly defined.

Definition 7.8 (Properly defined parallel relationships between minimal subtree definitions)
Let $TG = (T, I, A, C, OE, mult)$ be a type graph according to Definition 3.5. Let further MSD_{TG} refer to the set of all minimal subtree definitions in TG. Parallel relationships between minimal subtree definitions are properly defined if TG adheres to the following two conditions:

(1) For each pair (MSD, nt) with $MSD \in MSD_{TG}$, $nt \in T_N$ and for each pair (et_i, et_j) where

  - $et_i \in T_E$ is a non-containment edge type ($et_i \notin C$) with $et_i.src_I \cap MSD \neq \emptyset$ and $nt \in et_i.tgt_I$,
  - $et_j \in T_E$ is a non-containment edge type ($et_j \notin C$) with $et_j.src_I \cap MSD \neq \emptyset$ and $nt \in et_j.tgt_I$,

  the multiplicity combination $(et_i.lb > 0 \wedge et_j.lb > 0)$ implies that we have

  - $et_i.lb = et_j.lb = 1$, and
  - $et_i.src = et_j.src$, and
  - there is no non-containment edge type $et_z \in T_E$ ($et_z \notin C, et_z \neq et_i, et_z \neq et_j$) with $(et_z.src_I \cap MSD \neq \emptyset) \wedge (nt \in et_z.tgt_I) \wedge (et_z.lb > 0)$.

(2) For each pair (MSD_i, MSD_j) of minimal subtree definitions ($MSD_i, MSD_j \in MSD_{TG}$, $i \neq j$) and for each pair (ncp, c) where

  - ncp is a path $(et_1, ..., et_n)$ of non-containment edge types with $n \geq 1$ and $et_1.src_I \cap MSD_i \neq \emptyset$ and $et_n.tgt_I \cap MSD_j \neq \emptyset$,

– $c \in C$ is a containment edge type with $c.src_I \cap MSD_i \neq \emptyset$ and $c.tgt_I \cap MSD_j \neq \emptyset$,

there is at least one edge type $et \in \{et_1, ..., et_n\}$ with $et.lb = 0$.

Type graph with restricted multiplicities. We formally treat an effective meta-model that adheres to the above conditions as a special kind of type graph to which we refer as *type graph with restricted multiplicities*. Definition 7.9 summarizes the above conditions.

Definition 7.9 (Type graph with restricted multiplicities)
A type graph with restricted multiplicities $TG_{rmult} = (T, I, A, C, OE, mult)$ is a type graph according to Definition 3.5 which adheres to the following additional conditions:

(1) Multiplicity patterns are properly defined according to Definition 7.3.

(2) Multiplicities at containment edge types are properly defined according to Definition 7.4.

(3) Minimal subtrees are defined properly according to Definition 7.6.

(4) Cyclic relationships between minimal subtree definitions are defined properly according to Definition 7.7.

(5) Parallel relationships between minimal subtree definitions are defined properly according to Definition 7.8.

Example 7.1 (Meta-model of UML state machines as type graph with restricted multiplicities)
Reconsider the meta-model of simple UML state machines of Figure 3.4 without well-formedness rules formulated in OCL. Note that conditions (1), (2), (3), (4) and (5) of Definition 7.9 are fulfilled, i.e. the meta-model can be formally represented as a type graph with restricted multiplicities.

7.3.2 Generation of Creation and Deletion Rules

In the following, we describe how creation and deletion rules are to be derived for certain meta-model patterns. We begin with the generation of basic node creation/deletion rules. Subsequently, we show how these basic node creation/deletion rules are to be supplemented such that *i)* mandatory children are created (deleted) and *ii)* all created (deleted) nodes are connected to (disconnected from) their mandatory neighbors in a single step. Finally, we consider the generation of edge creation/deletion rules. According to our design principles of Section 7.2, a general policy is that for each node creation rule an inverse node deletion rule, and for each edge creation rule an inverse edge deletion rule is to be generated.

Basic root node creation/deletion rules. For any non-abstract node type B that has no incoming containment edge types and no supertype with incoming containment edge types (s. meta-model pattern P_0 in Figure 7.6), a basic root node creation rule of the form `create_B` is derived; it simply creates a single root node of type B. Moreover, a deletion rule of the form `delete_B` being inverse to `create_B` is to be derived for each occurrence of meta-model pattern P_0.

Meta-model pattern	Basic root node creation rule	Basic root node deletion rule
P_0 B $\nexists et \in in_I(B) : et \in C_{TG}$	create_B(out: r) «create» r:B	delete_B(in: r) «delete» r:B

Figure 7.6: Generation of basic root node creation and deletion rules

Basic non-root node creation/deletion rules. In our notion of a graph with containment, each node which is not a root node of a model has a unique container from which it is referenced by an incoming containment edge. Rules of the form `create_B*_b` (`delete_B*_b`) are the most basic rules for creating (deleting) a non-root node of a specific type. We derive these rules for meta-model pattern P_1 of Figure 7.7 in which B refers to a non-abstract node type defined by the given meta-model. Note that the notation $B*$ means that we derive these rules for node type B and any of its subtypes given by $allsub(B)$. If b has an associated opposite edge type a, this requires a consistent handling of opposite edges, i.e. edges of types b and a are created and deleted in pairs. If b does not define an opposite edge type, we basically derive the same rules, except that there is no opposite edge which is to be created (deleted). For brevity, we omit this special variant in Figure 7.7.

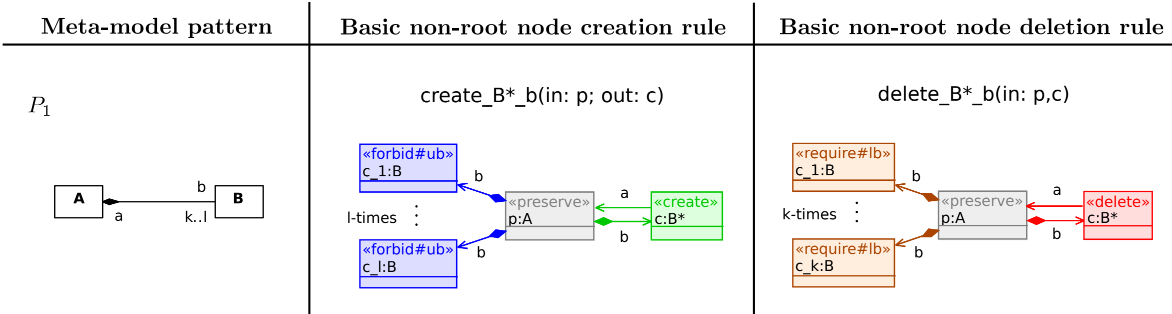


Figure 7.7: Generation of basic non-root node creation and deletion rules

If containment edge type b specifies a multiplicity with property *bounded*, i.e. $l \neq *$, a non-root node creation rule is only applicable to a model M at match m if the selected parent node $m(p)$ has at most $l-1$ outgoing edges of type b. To that end, `create_B*_b` is supplemented by NAC `ub` which forbids the existence of l outgoing edges of type b for parent node `p`. We refer to this NAC as *upper bound check*.

Analogously, if containment edge type b specifies a multiplicity with property *required*, i.e. $k > 1$, rule delete_B*_b is only applicable to a model M at match m if the selected parent node $m(p)$ has at least $k+1$ outgoing edges of type b. Thus, delete_B*_b is supplemented by PAC lb to which we refer to as *lower bound check*.

Supplementing the creation of mandatory children. After basic node creation rules have been generated for all occurrences of meta-model patterns P_0 and P_1, these basic rules are supplemented such that mandatory children of created nodes are created, too. We refer to these rules as *mc-supplemented node creation rules*.

Algorithm 6 Supplementing the creation of mandatory children

```
 1: function SUPPLEMENTMCCREATIONS(Rule r)
 2:     NodeSet N_Cre = {n ∈ R_N | n ∈ Cre_r};        ▷ all nodes to be created by r : L ⇝ R
 3:     NodeSet N'_Cre = N_Cre;                       ▷ "new" nodes to be created by r : L ⇝ R
 4:     repeat
 5:         for all Node n ∈ N'_Cre do
 6:             SUPPLEMENTMCCREATION(r, n);           ▷ s. Figure 7.8
 7:             NodeSet N''_Cre = {n ∈ R_N | n ∈ Cre_r};   ▷ temporary variable N''_Cre
 8:             N'_Cre = N''_Cre \ N_Cre;             ▷ update N'_Cre
 9:             N_Cre = N''_Cre ;                     ▷ update N_Cre
10:         end for
11:     until N'_Cre ≠ ∅
12:     return ;
13: end function
```

Function SUPPLEMENTMCCREATIONS listed in Algorithm 6 illustrates how to supplement a node creation rule referred to as r. First, variable N_{Cre} is initialized to refer to the set of nodes which are to be created by r (line 2). The same set of rule nodes, which (at this point of time) contains only a single node, is assigned to variable N'_{Cre} (line 3). As a node can recursively have (indirect) mandatory children and our intention is to create all of them by a single rule application, rule r will be iteratively refined (lines 4-11): In each iteration, we generate additional change actions for the creation of mandatory children. Thus, each iteration potentially leads to new nodes which are to be created by r, these nodes are maintained by the set N'_{Cre}.

The supplementation is delegated to subroutine SUPPLEMENTMCCREATION(Rule r, Node n) whose implementation is straightforward and illustrated in Figure 7.8. The type of node n is referred to as B. A supplementation is to be performed for each outgoing containment edge type c of B which has a multiplicity property *required* and which references a concrete node type C (s. meta-model pattern P_2). Then, rule r is refined such that all mandatory children mc_1, ..., mc_k of n are to be created, too. Additionally, created nodes mc_1, ..., mc_k are immediately connected to their parent n via the respective containment edges of type c. Opposite edges are created if necessary. In Figure 7.8, we omit the special variant where containment edge type c does not define an opposite edge type b and thus no consistent handling of opposite edges is required.

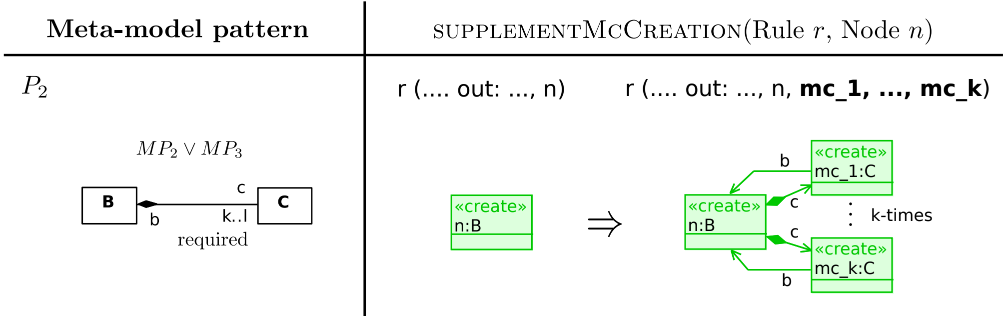


Figure 7.8: Supplementing the creation of mandatory children

Supplementing the deletion of mandatory children. The same supplementary procedure has to be applied to all basic node deletion rules since nodes are to be deleted together with their mandatory children. The respective function SUPPLEMENTMCDELETIONS(Rule r) and subroutine SUPPLEMENTMCDELETION(Rule r, Node n) can be implemented according to the same principle. We refer to these rules as *mc-supplemented node deletion rules*.

Lemma 7.1 (Termination of mandatory children supplementation)
Let $TG_{rmult} = (T, I, A, C, OE, mult)$ be a type graph with restricted multiplicities according to Definition 7.9. Let further $\mathcal{R}_{Cre}$ and $\mathcal{R}_{Del}$ be sets of node creation and node deletion rules generated for all occurrences of meta-model patterns P_0 and P_1 in TG_{rmult}. Then, our mc-supplementary procedures terminate when being applied to rules in $\mathcal{R}_{Cre}$ and $\mathcal{R}_{Del}$, respectively:

(1) Procedure SUPPLEMENTMCCREATIONS(Rule r) terminates for each creation rule $r_{Cre} \in \mathcal{R}_{Cre}$ provided as input argument.

(2) Procedure SUPPLEMENTMCDELETIONS(Rule r) terminates for each deletion rule $r_{Del} \in \mathcal{R}_{Del}$ provided as input argument.

Proof.

(1) The procedure SUPPLEMENTMCCREATION(Rule r) sketched in Algorithm 6 terminates if the recurrent supplementation of mandatory children which are to be created by rule $r \in \mathcal{R}_{Cre}$ (lines 4-11) terminates. An endless loop is only possible if the meta-model contains a cycle of containment edge types having multiplicity property *required*. This contradicts condition (1) of Definition 7.6.

(2) Follows directly from the proof of (1) and the design principle that there is an inverse deletion rule $r^{-}1 \in \mathcal{R}_{Del}$ for each creation rule $r \in \mathcal{R}_{Cre}$.

□

Supplementing the connection of mandatory neighbors. As node creation rules are intended to preserve multiplicity invariants defined by the effective meta-model, each created node must be immediately connected to its *mandatory neighbors*.

Algorithm 7 Supplementing the connection of mandatory neighbors

```
1:  function SUPPLEMENTMNCONNECTIONS(Rule r, TG_rmult)
2:      NodeSet N_Cre = {n ∈ R_N | n ∈ Cre_r};          ▷ all nodes to be created by r : L ⇝ R
3:      for all Node n ∈ N_Cre do
4:          for all EdgeType et ∈ allout(n.type) do
5:              if et ∉ TG_rmult.C and et.lb > 0 then     ▷ required non-containment edge type
6:                  SUPPLEMENTMNCONNECTION(r, n, et);                          ▷ s. Figure 7.9
7:              end if
8:          end for
9:      end for
10:     return ;
11: end function
```

Function SUPPLEMENTMNCONNECTIONS(Rule r, TG_{rmult}) listed in Algorithm 7 illustrates how to supplement the connection of mandatory neighbors. We assume here that mc-supplementation has been already performed, i.e. the function takes an mc-supplemented rule $r \in \mathcal{R}_{Cre}$ as input. The corresponding type graph with restricted multiplicities $TG_{rmult} = (T, I, A, C, OE, mult)$ is provided as additional input. Initially, variable N_{Cre} is initialized to refer to the set of nodes which are to be created by r (line 2). For each node n which is to be created by r (line 3) and for each outgoing edge type et of the node type of n (including inherited outgoing edge types, s. line 4), we check whether mandatory neighbors have to be connected. This is the case if et is a non-containment edge type and has a multiplicity property *required* (line 5).

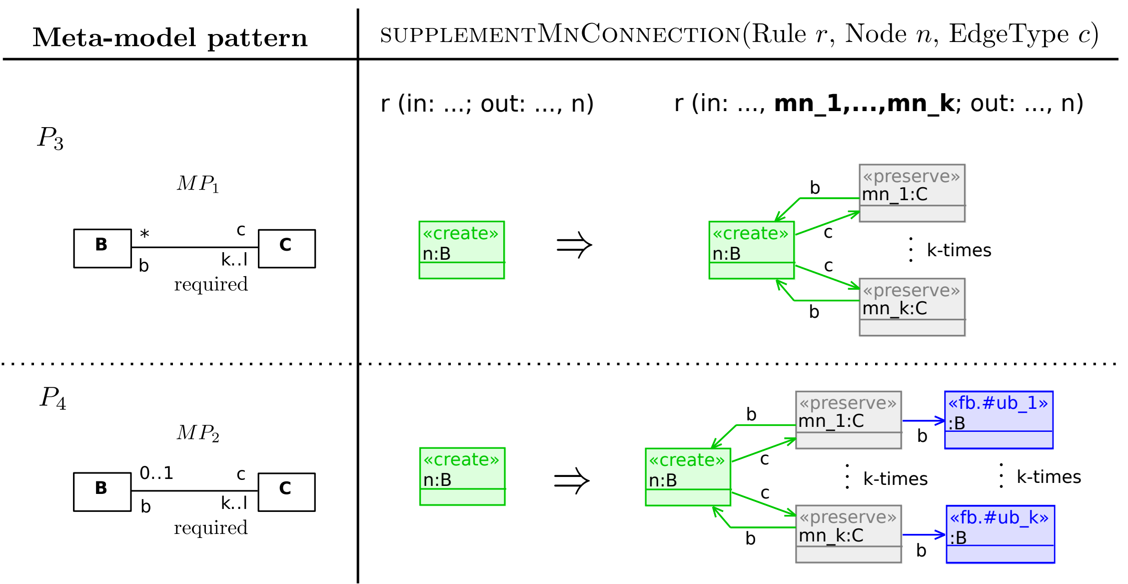


Figure 7.9: Supplementing the connection of mandatory neighbors

The supplementation is delegated to subroutine SUPPLEMENTMNCONNECTION(Rule r, Node n, EdgeType c) whose pattern-based implementation is illustrated in Figure 7.9. A supplementation is to be performed for meta-model patterns P_3 and P_4. Note that mandatory neighbors mn_1, ..., mn_k have to be provided as additional input parameters of the creation rule. Again, opposite edges are created in pairs. If edge type c does not define an opposite edge type, we derive the same rule as in the case of P_3, except the consistent handling of opposites (not shown in Figure 7.9). If an opposite edge of type b is to be created and if b specifies a multiplicity [0..1] (s. pattern P_4), then for each mandatory neighbor mn_i (with $i \in 1, ..., k$) a NAC ub_i prevents that mn_i is already connected via an edge of type b.

As a final post-processing step, we check whether additional variants of a rule which has been supplemented by function SUPPLEMENTMNCONNECTIONS have to be created. This is the case if TG_{rmult} defines "parallel" edge types according to the supported case *(P.i')* of Figure 7.5: Initially, this case results in an mc/mn-supplemented creation rule which contains a rule pattern as illustrated on the left side of Figure 7.10: A node of type B, which is to be created, is to be connected to mandatory neighbors of type C via two different types of non-containment edges *c1* and *c2*. The additional rule variant which is to be created for this rule pattern is illustrated on the right side of Figure 7.10. Note that only those parts are shown which differ from the initial variant on the left.

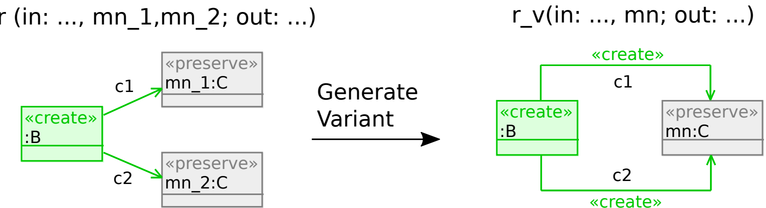


Figure 7.10: Generation of rule variants in the case of "parallel" relationships to mandatory neighbors

Supplementing the disconnection of mandatory neighbors. Analogously to node creations, a node which is to be deleted by a deletion rule must be disconnected from its *mandatory neighbors* by the same rule. The respective supplementary function SUPPLEMENTMNDISCONNECTIONS(Rule r), which is to be applied to all node deletion rules in $\mathcal{R}_{Del}$, can be implemented according to the same principle as our procedure for supplementing the creation of edges to mandatory neighbors.

Lemma 7.2 (Creation of minimal graph patterns)
Let $L(TG_{rmult})$ be the set of valid models induced by a type graph with restricted multiplicities TG_{rmult}. For every consistent model $M_1 \in L(TG_{rmult})$ and for every application $M_1 \overset{r_{Cre}}{\Longrightarrow} M_2$ of a mc/mn-supplemented creation rule $r_{Cre} \in \mathcal{R}_{Cre}$ transforming M_1 into a consistent model $M_2 \in L(TG_{rmult})$, the following conditions hold for the graph fragment F which is created by $M_1 \overset{r_{Cre}}{\Longrightarrow} M_2$:

(1) The created graph fragment F is *minimal* in the sense that there is no true subfragment $F' \subset F$ which can be created starting from M_1 such that the resulting model M_2' is a consistent model.

(2) If there is a fragment F'' with $F'' \cap F \neq \emptyset$ which can be created starting from M_1 such that the resulting model M_2'' is a consistent model, then there is also a rule r''_{Cre} to perform the transformation $M_1 \stackrel{r''_{Cre}}{\Longrightarrow} M_2''$. In other words, *all variants* of a minimal pattern can be created.

Proof.

(1) There are the following possibilities to reduce a fragment F being created by $M_1 \stackrel{r_{Cre}}{\Longrightarrow} M_2$ to a subfragment $F' \subset F$:

1. We can omit the creation of one of the containment edges in F. According to our mc-supplementary procedure, we know that each containment edge which is created by the application of r_{Cre} is used for connecting a mandatory child which is also created. If one of these connections is missing, the *no-lower-bound-violation* condition is violated for at least one lower bound defined by a containment edge type in TG_{rmult}.
2. We can omit the creation of one of the non-containment edges in F. According to our mn-supplementary procedure, we know that each non-containment edge which is created by the application of r_{Cre} is used for connecting a mandatory neighbor. If one of these connections is missing, the *no-lower-bound-violation* condition is violated for at least one lower bound defined by a non-containment edge type in TG_{rmult}.
3. We can omit the creation of one of the nodes in F. Since our rule graphs are always connected, we also have to omit the creation of the corresponding containment edge in order to prevent dangling containment edges in the model, which in turn leads to case 1.

(2) For the proof of (2), we have to investigate what we can say about the relation between F and F''. We consider the creation of minimal subtrees first, then we investigate possible variants of how mandatory neighbors can be connected.

1. Concerning the creation of a minimal subtree, we know that each (indirect) mandatory child of the root node of this subtree must be represented by a *distinct* node. This follows directly from the *at-most-one-container* constraint. From this fact and from (1), we can conclude that there must be a bijective mapping $m_T : T \rightarrow T''$ between minimal subtrees $T \subseteq F$ and $T'' \subseteq F''$, respectively. Provided that the mapping m_T is *type-preserving*, T'' can be created by the same rule r_{Cre}. Another variant for creating a minimal subtree such that m_T is not type-preserving is only possible if TG_{rmult} contains *required* containment edge types referencing a node type which has

subtypes. This contradicts our assumption that multiplicities at containment edge types are properly defined according to Definition 7.4.

2. From 1., we know that created minimal subtrees $T \subseteq F$ and $T'' \subseteq F''$ are unique up to isomorphism. The possibilities of how nodes in T and T'' can be connected to mandatory neighbors can be classified as follows:
   - Each mandatory neighbor is represented by a distinct node in M_1. In this case, there is a bijective mapping $m_B : B_T \rightarrow B_{T''}$ between the boundary graphs $B_T \supseteq F$ and $B_{T''} \supseteq F''$, respectively. Obviously, the mapping m_B must be *type-compatible*. Since we use the most general types for connecting mandatory neighbors in our mn-supplemented rules, connecting all mandatory neighbors of T'' can be achieved by the same rule r_{Cre}.
   - How can we deviate from the above pattern that each mandatory neighbor is represented by a distinct node? Connecting one node via multiple non-containment edges of the same type is not possible since this leads to a violation of the *no-parallel-edges* constraint. Thus, a deviation is only possible if TG_{rmult} defines at least two *required* non-containment edge types which are "parallel" in the sense that they originate from the same minimal subtree definition and have the same target node type. In general, this contradicts our assumption that all parallel relationships between minimal subtree definitions are properly defined according to Definition 7.8. Variants resulting from the special case that we have two parallel edge types where each of these edge types defines a lower bound of 1 are generated.

□

Properties (1) and (2) of Lemma 7.2 also hold for our mc/mn-supplemented deletion rules. The respective lemma (and its proof) is omitted here, it follows directly from Lemma 7.2 and the design principle that there is an inverse deletion rule $r^{-1} \in \mathcal{R}_{Del}$ for each creation rule $r \in \mathcal{R}_{Cre}$.

Edge creation/deletion rules. If a non-containment edge type b does not have a fixed multiplicity, then rules for creating and deleting an edge of this type are derived (s. meta-model patterns P_5 and P_6 in Figure 7.11). Such a rule of the form create_b (delete_b) takes two parameters as input: In case of create_b, nodes s and t identify the source and target node between which an edge of type b is to be created. In case of delete_b, source node s and target node t uniquely identify an edge of type b between s and t which is to be deleted. If necessary, opposite edges of type a are created (deleted) consistently (in Figure 7.11, we again omit the special variant of P_5 in which edge type b does not have an opposite edge type and thus no consistent handling of opposite edges is required).

NAC parallel (s. edge creation rules derived for meta-model patterns P_5 and P_6), to which we refer as *parallel edge check*, prevents modifications of an ASG which lead to a

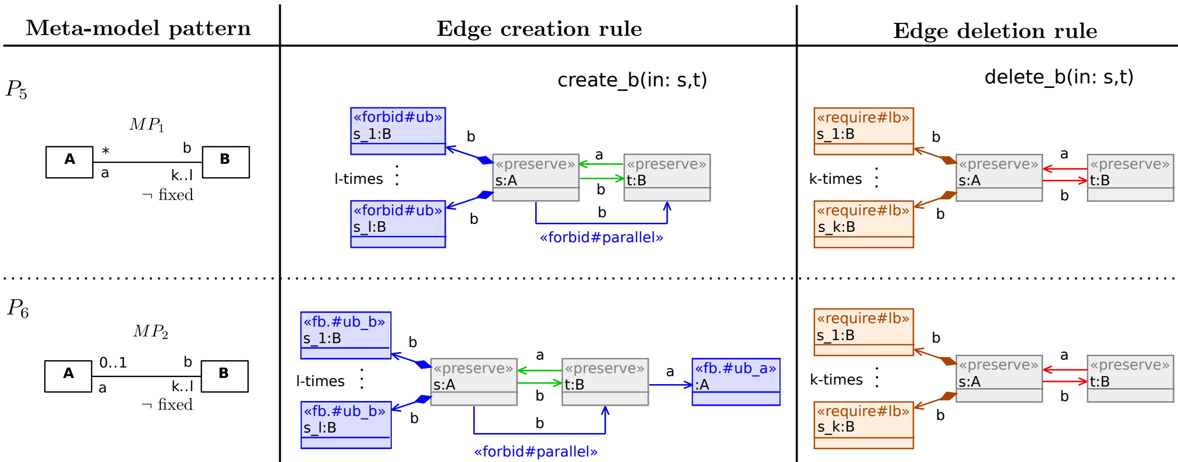


Figure 7.11: Generation of edge creation and deletion rules

violation of the *no-parallel-edges* constraint, i.e. it prevents an edge creation rule from being applied to a model M at match m if $m(s)$ is already linked with $m(t)$ via an edge of type b. Due to the consistent handling of opposite edges, parallel edge checks are only required in one direction. Note that parallel edge checks can be omitted in case of edge types b that do not have a multiplicity property *many* because parallel edges will be implicitly prevented by the upper bound check (s. below).

Edge creation rules derived for meta-model patterns P_5 and P_6 are supplemented by an *upper bound check* to ensure the *no-upper-bound-violation* condition specified by upper bound l of the multiplicity invariant for edge type b (s. NACs ub and ub_b in Figure 7.11). In case of pattern P_6, we have an additional NAC called ub_a which ensures that node t is not yet connected via an edge of type a which has an upper bound of 1. While upper bound checks can be obviously omitted for inverse edge deletion rules, a *lower bound check* has to be inserted to ensure the *no-lower-bound-violation* condition for lower bound k of edge type b.

If the edge types b in the meta-model patterns P_5 and P_6 are loop edges, i.e. we have $A = B$, then we derive a second variant for the respective edge creation (deletion) rules in which rule edges of type a (and opposite edges of type b) are loop edges, too (not shown in Figure 7.11).

Example 7.2 (Creation and deletion rules for simple state machines)
Figure 7.12 illustrates a subset of creation and deletion rules generated for our state machine meta-model of Figure 3.4. In sum, we get the following rules:

Firstly, we get the root node creation rule create_StateMachine and its inverse deletion rule delete_StateMachine. Both rules are supplemented such that a mandatory child of type *Region* is created (deleted).

Moreover, we have basic non-root node creation rules, i.e. create_FinalState_subvertex, create_State_subvertex, create_Region_region and create_Region_subregion. For each of these

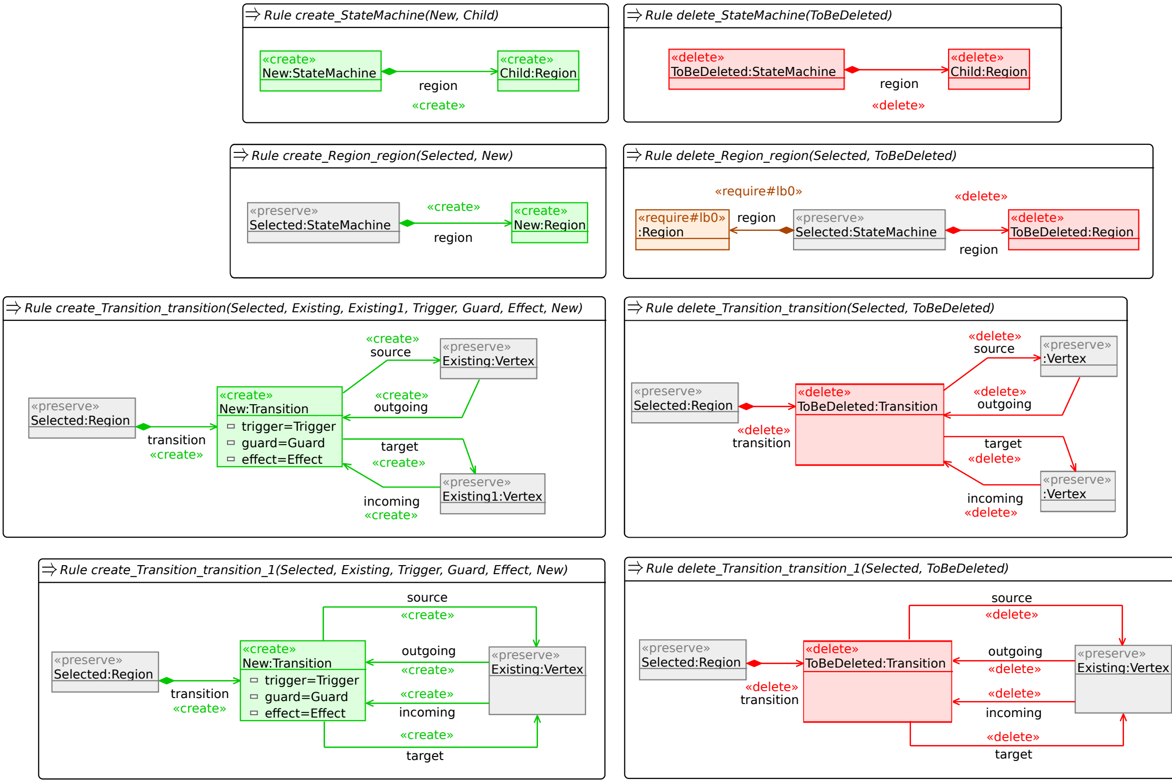


Figure 7.12: Subset of generated creation and deletion rules for UML state machines

basic creation rules, we have a corresponding deletion rule, i.e. delete_FinalState_subvertex, delete_State_subvertex, delete_Region_region and delete_Region_subregion. Note that *region* is the only containment edge type of our sample meta-model that specifies a multiplicity (namely [1..*]) which is relevant w.r.t. to basic consistency constraint *no-lower-bound-violation*. Consequently, a lower bound check is generated for delete_Region_region.

Rule create_Transition_transition requires the source and target states which are to be connected by the transition as input arguments and immediately connects these mandatory neighbors to the created transition. Due to the parallel edge types *source* and *target* which both have a multiplicity [1..1] (s. Figure 3.4), we get a second variant of this rule: create_Transition_transition_1. This variant creates edges of types *source* and *target* referencing the same *Vertex* node, i.e. it creates a "loop transition" in the visual representation of a state machine. Analogously, we get two deletion rules delete_Transition_transition and delete_Transition_transition_1 being inverse to the corresponding creation rules.

Note that the examples in Figure 7.12 illustrate that attribute declarations are conceptually handled as special edge types with a *fixed* multiplicity of [1..1]. Thus, attribute values are treated as "mandatory neighbors", a concrete value has to be assigned to each attribute of a node created by a node creation rule.

7.3.3 Generation of Move and Change Rules

Move and change rules re-structure the relations between existing nodes of an ASG. We derive these rules for meta-model patterns P_7, P_8, and P_9 as illustrated in Figure 7.13. Move and change rules do not require an inverse generation as they are inverse to themselves. Note that the generation of edit rules changing attribute values of ASG nodes is straightforward and will therefore not be discussed here in detail. Conceptually, attribute declarations can be regarded as special edge types with multiplicity [1..1] (cf. Section 3.1). Thus, attribute declarations basically lead to the same kind of edit rules as non-containment edge types with a *fixed* multiplicity (s. *change rules* derived for meta-model pattern P_8).

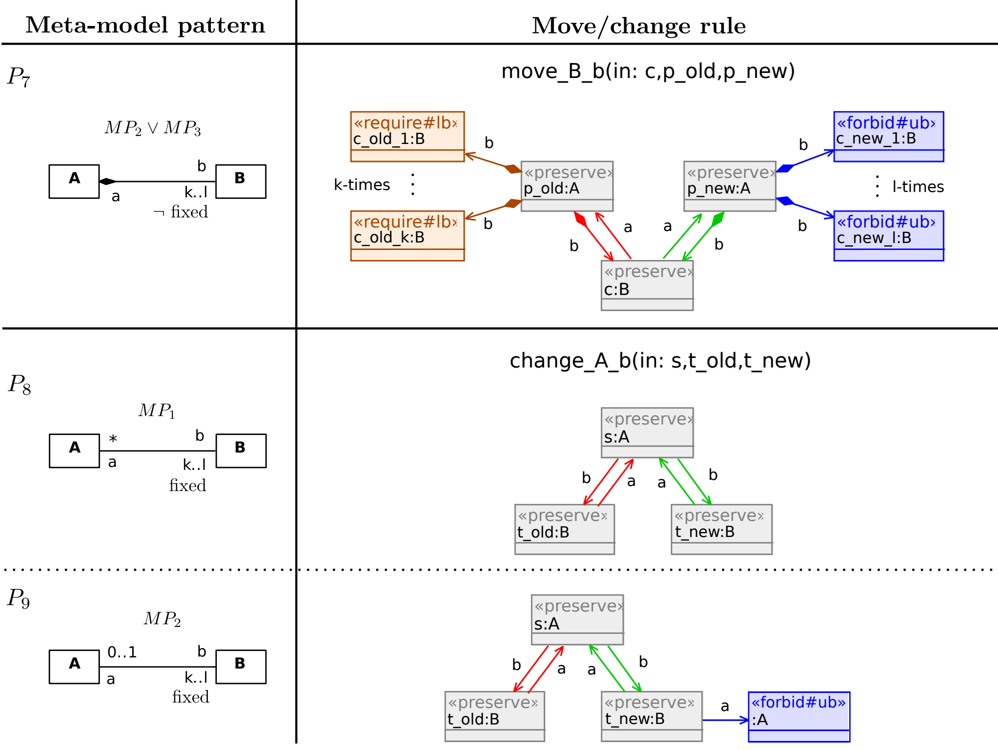


Figure 7.13: Generation of move and change rules

Move node. If a containment edge type b does not have a fixed multiplicity, then a *move rule* of the form move_B_b for moving an instance of the respective target node type B is generated (s. meta-model pattern P_7 in Figure 7.13). Such a move rule changes the container of a selected ASG node of type B (or any subtype of B) and takes three input arguments; node c which is to be moved and nodes p_old and p_new representing its old and new container. Lower and upper bound checks are inserted to ensure *no-lower-bound-violation* and *no-upper-bound-violation* conditions. If containment edge type b has an opposite edge type a, then the respective edges of these types are created and deleted in pairs.

Change edge. For each non-containment edge type b which does not have a fixed multiplicity, a *change rule* of the form change_A_b is derived (s. patterns P_8 and P_9). Such a change rule takes three input arguments; node s of type A (or any subtype of A) which is the source of an edge of type b, and nodes t_old and t_new representing its old and new target node. Note that no lower and upper bound checks are needed for edge type b since the number of edges of type b outgoing from node a is not changed. If edge type b has an opposite edge type a, then the respective edges of these types are created and deleted in pairs. If the opposite edge type a specifies an upper bound of 1 (s. pattern P_9), then an upper bound check is generated to ensure the *no-upper-bound-violation* constraint for the new target node t_new.

Example 7.3 (Move and change rules for simple state machines)
Figure 7.14 illustrates a subset of move and change rules generated for our in state machine meta-model of Figure 3.4.

We get move rules move_Region_region, move_Region_subregion, move_Transition_transition and move_Vertex_subvertex. A lower bound check for rule move_Region_subregion prevents a violation of multiplicity invariant [1..∗] specified by containment edge type *region*, while all other move rules are constructed analogously to move_Vertex_subvertex.

Furthermore, we get change rules change_Transition_source and change_Transition_target (s. Figure 7.14) for non-containment edge types *source* and *target* which both specify a fixed multiplicity of [1..1].

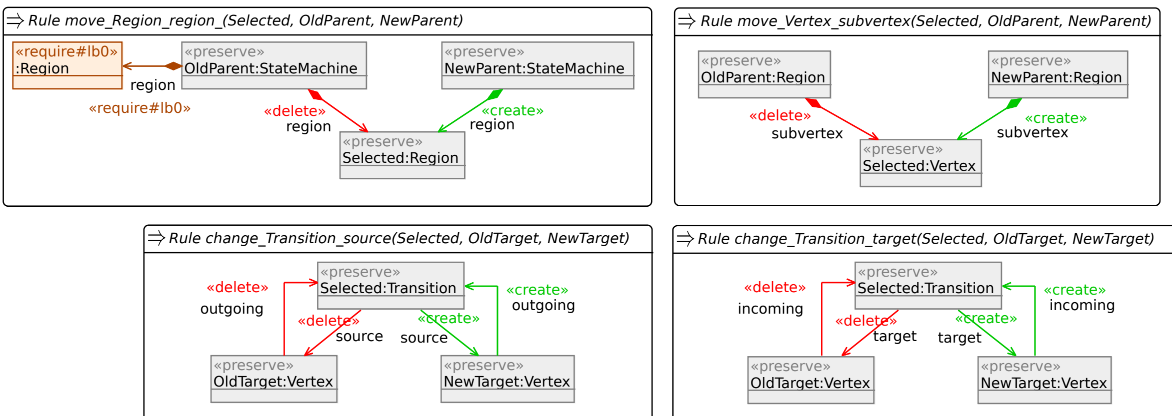


Figure 7.14: Subset of generated move and change rules for UML state machines

7.3.4 Consistency-preservation of Generated Rules

Now that we have precisely described our procedure for generating CPERs, we can finally conclude to which extent we can guarantee the preservation of the consistency of a model to which these rules are applied. Note that ASGs resulting from rule-based transformations of typed graphs are guaranteed to be correctly typed in general [94],

thus we will only consider the preservation of further *basic consistency constraints* and *multiplicity invariants* (s. Section 3.2).

Creation and deletion rules. Given a type graph with restricted multiplicities TG_{rmult}, then creation and deletion rules derived from TG_{rmult} preserve all basic consistency constraints and all multiplicity invariants w.r.t. TG_{rmult}:

- *at-most-one-container* and *no-containment-cycles*: The containment-related constraints *at-most-one-container* and *no-containment-cycles* cannot be violated by edge creation rules because they are only generated for non-containment edges. Moreover, these constraints cannot be violated by node creation rules because we create and delete containment edges only together with the contained node.

- *no-parallel-edges*: In case of containment edges, the *no-parallel-edges* constraint is implied by the *at-most-one-container* constraint. In case of non-containment edges, parallel edge checks of edge creation rules prevent modifications of an ASG leading to a violation of the *no-parallel-edges* constraint.

- *all-opposite-edges*: The *all-opposite-edges* constraint cannot be violated because we create and delete opposite edges in pairs.

- *no-lower-bound-violation*: Lower bound violations caused by basic node deletion rules and edge deletion rules are prevented by lower bound checks. According to Lemma 7.2, we know that mc/mn-supplemented rules create and delete minimal patterns, i.e. they cannot lead to lower bound violations, too.

- *no-upper-bound-violation*: Upper bound violations can only be caused by non-root node creation rules and edge creation rules. We prevent such violations by upper bound checks.

Move and change rules. Given a type graph with restricted multiplicities TG_{rmult}, then move and change rules derived from TG_{rmult} preserve all multiplicity invariants w.r.t. TG_{rmult} as well as basic consistency constraints *at-most-one-container*, *no-parallel-edges*, and *all-opposite-edges*:

- *at-most-one-container*: Constraint *at-most-one-container* cannot be violated by move rules because they only change the container of a given node.

- *no-parallel-edges*: In case of containment edges, the *no-parallel-edges* constraint is implied by the at-most-one-container constraint. In case of non-containment edges, parallel edge checks of change rules prevent modifications of an ASG which lead to a violation of the *no-parallel-edges* constraint.

- *all-opposite-edges*: The *all-opposite-edges* constraint cannot be violated because we create and delete opposite edges in pairs.

- *no-lower-bound-violation*: Change rules cannot violate lower bounds since the number of edges of a particular type outgoing from a particular node is not affected. Lower bound violations caused by move rules are prevented by lower bound checks.

- *no-upper-bound-violation*: Change rules cannot violate upper bounds since the number of edges of a particular type outgoing from a particular node is not affected. Upper bound violations caused by move rules are prevented by upper bound checks.

7.4 Completeness of the Generated Edit Rule Set

With respect to completeness, we have to show that the set of generated edit rules fulfills conditions (1) and (2) of Definition 7.1, i.e. edit rules have to be instance-generating and instance-deleting.

Instance generation. For condition (1) of Definition 7.1, we have to show that the set $\mathcal{R}_{Cre} \subseteq \mathcal{R}$ of creation rules is instance-generating. To that end, we will first show how a model being represented as TG_{rmult}-typed graph can be conceptually split into smaller "partitions" which we will later utilize for proving the instance-generating property of rule set $\mathcal{R}_{Cre}$.

Definition 7.10 (Conceptual splitting of a TG_{rmult}-typed graph)
Let M be a consistent model which is represented as ASG being typed over a type graph with restricted multiplicities TG_{rmult}. M can be conceptually split into a 4-tuple $P = (SR, NR, MP, NC)$ where SR, NR, MP and NC refer to the following sets of graph fragments:

$SR \subseteq \mathcal{P}(M_N)$ is the set of all root node fragments. A root node fragment $sr \in SR$ consists of a root node $r \in M_N$ which does neither have mandatory children nor mandatory neighbors. Please note that r, although being a single node, is treated as fragment $sr = \{r\}$ here for the sake of homogeneity w.r.t. fragment sets NR, MP and NC.

$NR \subseteq \mathcal{P}(M_N \cup M_E)$ is the set of all non-root node fragments. A non-root node fragment $nr \in NR$ consists of a non-root node n together with its incoming containment edge $e \in in(n)$. If e has an opposite edge then it is included in nr, too. Moreover, n is not contained in a minimal pattern, i.e. it does neither have mandatory children nor mandatory neighbors.

$MP \subseteq \mathcal{P}(M_N \cup M_E)$ is the set of all minimal graph patterns, each pattern $mp \in MP$ cannot be reduced to a smaller pattern without violating lower bounds defined by multiplicity invariants of TG_{rmult}.

$NC \subseteq \mathcal{P}(M_E)$ is the set of all non-containment edge fragments, each fragment $nc \in NC$ consists of a non-containment edge e which is not part of a minimal pattern $mp \in MP$. If e has an opposite edge then it is included in nc, too.

Example 7.4 (Conceptual splitting of a sample ASG representation)
Figure 7.15 shows parts of the sample state machine of Figure 2.2 in its abstract syntax. The excerpt contains states *Idle* and *Active* as well as transitions *lift* and *hangup* between *Idle* and *Active*. Substates of composite state *Active* and transitions between them are omitted. We have three minimal patterns, labeled mp_{1-3} in Figure 7.15. and three non-root node fragments referred to as nr_{1-3}. Sets SR and NC are empty

The example illustrates that minimal patterns differ from simple non-root node fragments (or single root nodes) in the sense that they cover at least one node which has mandatory children or neighbors. Minimal pattern mp_1 covers the root node of type *StateMachine* together with its mandatory child of type *Region* and the respective containment edge. Minimal pattern mp_2 covers the node of type *Transition* representing transition *lift*. It has two mandatory neighbors, the non-containment edges connecting these neighbors are also covered by mp_2. Since the *Transition* node is not a root node, the containment edge connecting the node with its parent node of type *Region* is also covered by mp_2. The structure of the minimal pattern mp_3 is similar to mp_2.

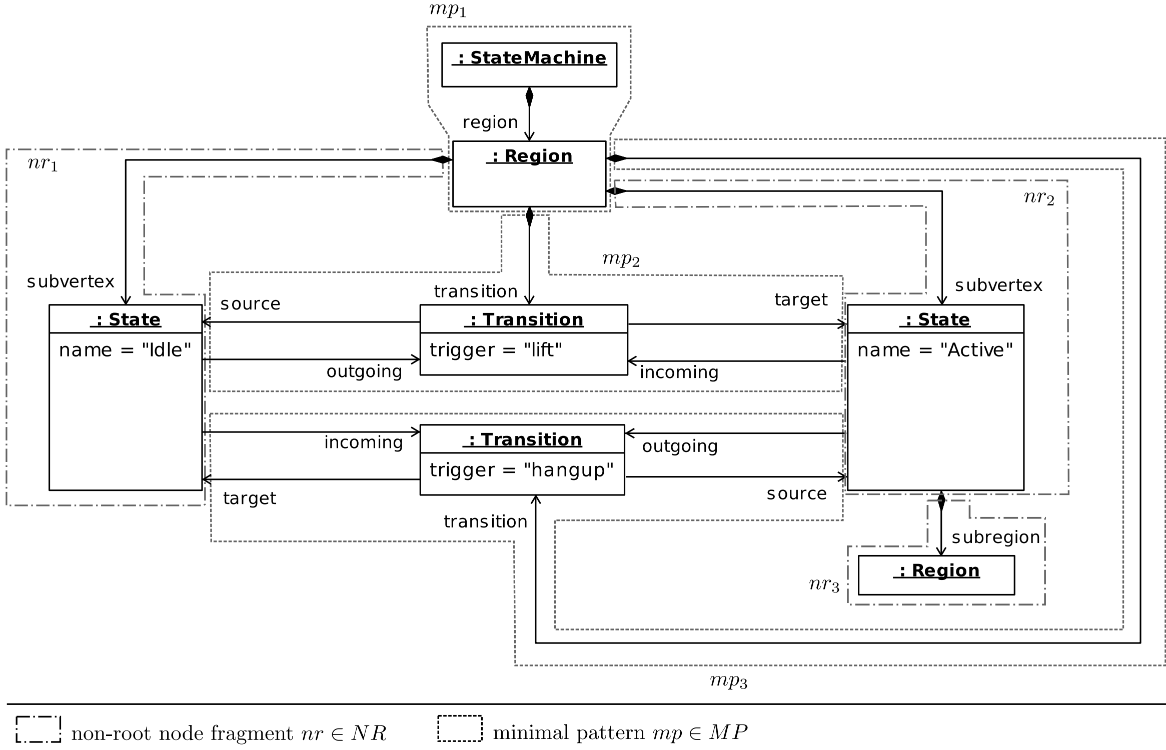


Figure 7.15: Conceptual splitting of a sample ASG representation

Lemma 7.3 (Partitioning property of a conceptual splitting)
Let M be a consistent model which is represented as ASG being typed over a type graph with restricted multiplicities TG_{rmult}. A splitting $P = (SR, NR, MP, NC)$ can be constructed such that the set $\mathcal{F}_P = \{F_1, F_2, ..., F_n\} = SR \cup NR \cup MP \cup NC$ of all fragments of P adheres to the following two conditions:

(1) $\bigcup_{i=1}^{n} F_i = M$, i.e. the fragments induced by splitting P cover the complete model M, and

(2) $F_i \cap F_j = \emptyset$ for all $1 \leq i < j \leq n$, i.e. the fragments induced by splitting P are pairwise disjoint.

Proof.

(1) Assume that there is a node $n \in M_N$ which is not covered by one of our fragments in $\mathcal{F}_P$. We know that n cannot be a "single" root node (a root node which has neither mandatory children nor mandatory neighbors), otherwise it would be contained by a fragment $sr \in SR$. We also know that n cannot be part of a minimal pattern, otherwise it would be contained by a pattern $mp \in MP$. Finally, if n is a "single" non-root node and it is not contained by a fragment $nr \in NR$, then n has no container node at all which contradicts our assumption that M represents a consistent model.

    Analogously, assume that there is an edge $e \in M_E$ which is not covered by one of our fragments in $\mathcal{F}_P$. We know that e cannot be part of a minimal pattern, otherwise it would be contained by a pattern $mp \in MP$. Further on, e cannot be a non-containment edge, otherwise it would be contained by a fragment $nc \in NC$. We also know that e cannot be part of the tree-structure of M, otherwise e would be contained by a fragment $nr \in NR$. Finally, e can only be a containment edge being part of a containment-cycle in M, which again contradicts our assumption that M represents a consistent model.

(2) While the condition that fragments $F_i, F_j \in \mathcal{F}_P$ are disjoint is obvious for almost all possible pairs of fragments, we have to clarify this condition for minimal patterns, i.e. we have to show that $mp_i \cap mp_j = \emptyset$ for all $mp_i, mp_j \in MP$.

    Concerning the nodes which are covered by a minimal pattern, we know that these nodes are forming a tree structure, i.e. we have a subtree root (which does not necessarily have to be a root node of model M) together with direct and indirect mandatory children. Mandatory neighbors of these nodes are not included in a minimal pattern, but only the non-containment edges indicating the mandatory neighbors. Thus, two patterns mp_i and mp_j do not overlap in their node sets.

    Consequently, two patterns mp_i and mp_j do not overlap in their sets of containment-edges. Otherwise, the *at-most-one-container* constraint would be violated which contradicts our assumption that M is a consistent model.

Finally, two patterns mp_i and mp_j do not overlap in their non-containment edge sets indicating mandatory neighbors because each mandatory neighbor is indicated by a distinct non-containment edge or a pair of non-containment edges being opposite to each other: The situation that two minimal patterns "share" the same non-containment edges indicating mandatory neighbors would only be possible if we have a meta-model that specifies *i)* edge types et_1 and et_2 being opposite to each other and *ii)* both et_1 and et_2 have a multiplicity with property *required*, which contradicts condition (1) of Definition 7.7.

□

Theorem 7.1 (Instance-generation with CPERs)
Let TG_{rmult} be a type graph with restricted multiplicities. Let further $\mathcal{R}_{Cre}$ be the set of creation rules derived from TG_{rmult}. Then, every consistent model M being properly typed over TG_{rmult} can be constructed starting from the empty model ϵ by exclusively using rules available in $\mathcal{R}_{Cre}$.

Proof. Given a consistent model M being typed over TG_{rmult}, we have to construct a sequence of rule applications which, starting from the empty model ϵ, creates model M. We know that model M can be conceptually split into a 4-tuple $P = (SR, NR, MP, NC)$ according to Definition 7.10 such that the splitting fulfills the "partitioning" property of Lemma 7.3. A derivation sequence can be constructed as follows:

1. Initially, we create all "single" root nodes identified by fragments $sr \in SR$. To that end, we apply basic root node creation rules in any order. It is easy to see that all fragments $sr \in SR$ can be created by our basic root node creation rules and that these rules are sequentially independent of each other.

2. Next, we create non-root nodes identified by fragments $nr \in NR$ and all minimal graph patterns $mp \in MP$. It is easy to see that non-root node fragments can be created by our basic non-root node creation rules. From Lemma 7.2 we can conclude that mc/mn-supplemented node creation rules lead to the creation of minimal graph patterns and that all variants of minimal graph patterns that may occur in a consistent TG_{rmult}-typed graph can be created.

   These rules must be applied in a suitable order as they may depend on each other. A rule application creating fragment $F_i \in (NR \cup MP)$ depends on a rule application creating fragment $F_j \in (NR \cup MP)$, if a boundary node of F_i (i.e. a parent node or mandatory neighbor which is needed for creating F_i) is part of the fragment F_j. In this case, F_j must be created first. A suitable order exists if there are no instance structures in a consistent TG_{rmult}-typed model leading to cyclic dependencies between rule applications creating the respective fragments. There are four cases which may lead to cyclic dependencies:

   i) "parent/parent": There is a node $p_i \in mp_i$ which serves as parent for a node $n_j \in mp_j$ and there is also a node $p_j \in mp_j$ which serves as mandatory neighbor for a node $n_i \in mp_i$.

ii) "neighbor/neighbor": There is a node $n_i \in mp_i$ which has a mandatory neighbor $mn_j \in mp_j$ and there is also a node $n_j \in mp_j$ which has a mandatory neighbor $mn_i \in mp_i$.

iii) "parent/neighbor": There is a node $p_i \in mp_i$ which serves as parent for a node $n_j \in mp_j$ and there is also a node $mn_j \in mp_j$ which serves as mandatory neighbor for a node $n_i \in mp_i$.

iv) "neighbor/parent": Analogously to *iii)*

Case *i)* represents cyclic containment which is invalid in a consistent model. Instance structures representing case *ii)* are only possible if type graph TG_{rmult} defines a cycle of required non-containment edge types between minimal subtree definitions. This contradicts condition (1) of Definition 7.7. Instance structures representing cases *iii)* and *iv)* are only possible if type graph TG_{rmult} defines a cyclic relationship between two minimal subtree definitions MSD_i and MSD_j such that MSD_i is connected to MSD_j via a path of required non-containment edge types and MSD_j is connected to MSD_i via a containment edge type. This contradicts condition (2) of Definition 7.7. Thus, we can conclude that cyclic dependencies between the respective creation rules cannot occur and that the rules can be applied in any order which is consistent to the partial order induced by the boundary nodes of the fragments which are to be created.

3. Finally, we create all non-containment edges being covered by fragments $nc \in NC$ by applying edge creation rules in any order. It is easy to see that all non-containment edges can be created by our edge creation rules and that these rules are sequentially independent of each other.

□

Example 7.5 (Generation of a sample instance model with CPERs)
Consider again the ASG representation of our sample UML state machine of Figure 7.15. Fragments nr_1, nr_2 and nr_3 can be created by non-root node creation rules create_State_subvertex (nr_1 and nr_2) and create_Region_subregion (nr_3). Minimal patterns mp_1, mp_2 and mp_3 can be created by the application of mc/mn-supplemented creation rules create_StateMachine (mp_1) and create_Transition_transition (mp_2 and mp_3). Obviously, nr_1 has to be created first. Subsequently, we can create fragments nr_2 and nr_3 in any order. The creation of patterns mp_2 and mp_3 requires fragments nr_1 and nr_2, while the creation of nr_3 requires only the fragment nr_2 to be already created.

Remark. As already mentioned in Section 3.3, a drawback of using meta-models in order to define the abstract syntax of a modeling language is that meta-models are declarative and do not directly specify "editing behavior". This problem generally re-appears when one wants to provide a specification of how to generate the (usually infinite) set of all valid instances that conform to a given meta-model, e.g. for the purpose of generating test models for model processing tools [200].

In this regard, the proof of Theorem 7.1 provides another result (which is of minor importance in the context of this thesis): It is constructive in the sense that it implicitly specifies a procedure of how to generate all valid instances that conform to a given meta-model which can be formalized as a type graph with restricted multiplicities. The generation process can be organized into three "layers", parts of this process are similar to the principle of instance-generating graph grammars (s. related work [96, 234] in Chapter 10):

1. Starting from the empty model, we first create an arbitrary number of single root nodes, which results in a discrete graph. The application of root node creation rules has to be stopped by user interaction or some termination criterion.

2. Subsequently, we create non-root nodes and minimal graph patterns. To that end, basic node creation rules and mc/mn-supplemented node creation rules can be applied as often as possible. Usually, the rule application process has to be stopped by user interaction or some termination criterion.

3. Finally, edge creation rules can be applied to create optional relationships, i.e. additional non-containment edges may be inserted as long as upper bounds are not reached. If the meta-model specifies multiplicities with property $\neg$*bounded* (i.e. upper bound = *), which is usually the case, the rule application process again has to be stopped by user interaction or some suitable termination criterion.

Note that the generated instance models are consistent since they are constructed using consistency-preserving edit rules only. Thus, instance sets generated according to the above procedure and instance sets induced by a type graph with restricted multiplicities are equivalent.

Instance deletion. Since deletion rules are constructed as inverse rules to creation rules, it is obvious that every consistent model can be stepwise deleted using rules available in $\mathcal{R}_{Del} \in \mathcal{R}$. In other words, the rule set $\mathcal{R}_{Del}$ is *instance-deleting* and thus fulfills condition (2) of Definition 7.1.

Theorem 7.2 (Instance-deletion with CPERs)
Let TG_{rmult} be a type graph with restricted multiplicities. Let further $\mathcal{R}_{Del}$ be the set of deletion rules derived from TG_{rmult}. Then, every consistent model M being properly typed over TG_{rmult} can be reduced to the empty model ϵ by exclusively using edit rules available in $\mathcal{R}_{Del}$.

Proof. Follows directly from Theorem 7.1 and the design principle that there is an inverse deletion rule $r^{-}1 \in \mathcal{R}_{Del}$ for each creation rule $r \in \mathcal{R}_{Cre}$. □

7.5 Analysis of Potential Transient Effects

So far, we have only discussed the completeness of the set of mandatory edit rules for a given modeling language. However, our method of operation recognition additionally requires any model modification to be expressible without causing transient effects in order to guarantee a complete lifting of low-level differences (s. Section 4.4). Thus, a thorough analysis of potential transient effects and how they are avoidable is given in the remainder of this section. According to Definition 7.2, we have to show for all pairs of generated edit rules that they (1) do not cause transient effects at all, or (2) are inverse to each other, or (3) can be summarized by an alternative rule.

The idea is to utilize the fact that generated edit rules are very schematic and to study potential transient effects on a per-kind basis. Table 7.1 summarizes possible combinations of how a pair of rules of different kinds can be applied in an editing sequence. A table entry has to be read as follows; r_1 and r_2 are representatives of our four different kinds of mandatory edit rules (s. Section 7.2) and we assume that r_2 is applied after r_1 in an editing sequence. Node creation and edge creation rules are summarized as creation rules, node deletion and edge deletion rules are summarized as deletion rules. We analyze each possible combination w.r.t. potential transient effects.

$r_1; r_2$	**Creation (Cre)**	**Deletion (Del)**	**Change (Chn)**	**Move (Mov)**
Cre	(1):$\bot$	(2) $\vee$ (3):*Cre'* $\vee$ (3):*Del'*	(3):*Cre'*	(3):*Cre'*
Del	(2) $\vee$ (3):*Cre'* $\vee$ (3):*Del'*	(1):$\bot$	(1):$\bot$	(1):$\bot$
Chn	(1):$\bot$	(3):*Del'*	(2) $\vee$ (3):*Chn'*	(1):$\emptyset$
Mov	(1):$\bot$	(3):*Del'*	(1):$\emptyset$	(2) $\vee$ (3):*Mov'*

Table 7.1: Analysis of potential transient effects in sequential applications of generated edit rules

Entries in Table 7.1 marked as (1) represent pairs of edit rule kinds that do not cause transient effects for all representatives of the respective kinds of edit rules. This is possible for two reasons, which are indicated by symbols $\emptyset$ and $\bot$ in Table 7.1:

$\emptyset$: The respective kinds of edit rules operate on different types of ASG elements. This is the case for change rules, which operate on non-containment edges, and move rules, which operate on containment edges.

$\bot$: These kinds of rules cannot modify the same elements in one editing sequence because the application of r_1 prevents r_2 from being successfully applied after r_1. In particular, an element cannot be changed or moved after it has been deleted. Moreover, an element cannot be created or deleted twice.

Transient effects are uncritical w.r.t. our approach to operation recognition if the application of r_2 takes back all effects of the application of r_1, i.e. the overall *effect is like no rule application*. This is the case for edit rules being inverse to each other, i.e. creation and deletion as well as change and move rules if applied with suitable arguments. These entries are marked as (2) in Table 7.1.

In the remaining cases marked as (3), transient effects are avoidable in the sense that there is an *alternative rule application summarizing all effects* of applying first r_1 and then r_2. Please note that generated edit rules specify only elementary consistency-checks which prevent basic consistency constraints no-lower-bound-violation and no-upper-bound-violation from being violated (s. Section 7.3). As we assume that an edit script $\Delta_{A \Rightarrow B}$ transforms a model from one consistent state A into another consistent state B and all intermediate states also represent a consistent model, application conditions of these kinds cannot be fulfilled only transiently. To that end, application conditions of generated edit rules and the condition that r_3 summarizes all pre-conditions of r_1 and r_2 (s. Definition 7.2) are sensibly omitted in the following considerations:

deletion': If applications of change and move rules are followed by a deletion of the modified element, the overall effect can be caused by only applying the deletion rule, possibly with slightly modified arguments, i.e. the deletion is performed in a different parent context.

creation': In case of creating an element being followed by the application of a change or move rule, the overall effect can be caused by only applying the creation operation, again with possibly slightly modified arguments. Value parameters must be adjusted according to the attribute value change caused by the application of a change operation. Nodes which are passed as arguments to object parameters must be selected properly according to relocations of model elements.

change'/move': If change and move rules are applied to the same element several times without being inverse to each other, the effect can be summarized by just applying the last change or move rule of such an editing sequence.

As each entry of Table 7.1 has at least one of the above properties (1), (2) or (3), we can conclude that a set of generated edit rules is sound according to Definition 7.2.

7.6 Requirements Induced by Consistency-preserving Edit Rules

As already mentioned in Section 7.1, a set of mandatory edit rules must be consistent with the properties of the matcher which is used in the first step of our differencing pipeline. To that end, we have to precisely specify the properties we expect from a given matching. These properties depend on the design of the edit rules. In this section, we discuss this requirement for our generated CPERs.

A trivial solution is to require that, for every pair (A, B) of consistent models A and B which are to be compared, the matcher produces a matching $M_{A,B} : A \rightsquigarrow B$ such that the intersection $A \cap B$ over $M_{A,B}$ is guaranteed to form again a consistent model. In this case, creation and deletion rules are sufficient to express the modification from A to B. However, this requirement would be very restrictive, as it prevents several kinds of edit operations to be detected by our approach to operation recognition.

Consider Figure 7.16 which shows parts of the introductory example of Figure 2.2 in abstract and concrete syntax:

- Firstly, assume states *Idle* to be corresponding, while states *DialTone* and *Active* are not in a correspondence relationship. If $A \cap B$ shall be a consistent model, then transitions *lift* must not correspond to each other. In this case, changes from A to B can be explained using creation and deletion rules only, i.e. transition *lift* including its various connections to *Idle* and *DialTone* is first deleted, and later on re-created and connected to *Idle* and *Active*.

- Alternatively, the difference can be explained by a single edit step which simply exchanges the target of transition *lift* from *DialTone* to *Active* by deleting and creating edges *target* and *incoming*. In this case, transitions *lift* must be in a correspondence, as shown in Figure 7.16. Note that this matching leads to an inconsistent model $A \cap B$; transition *lift* is missing a mandatory neighbor which is connected via an edge of type *target*.

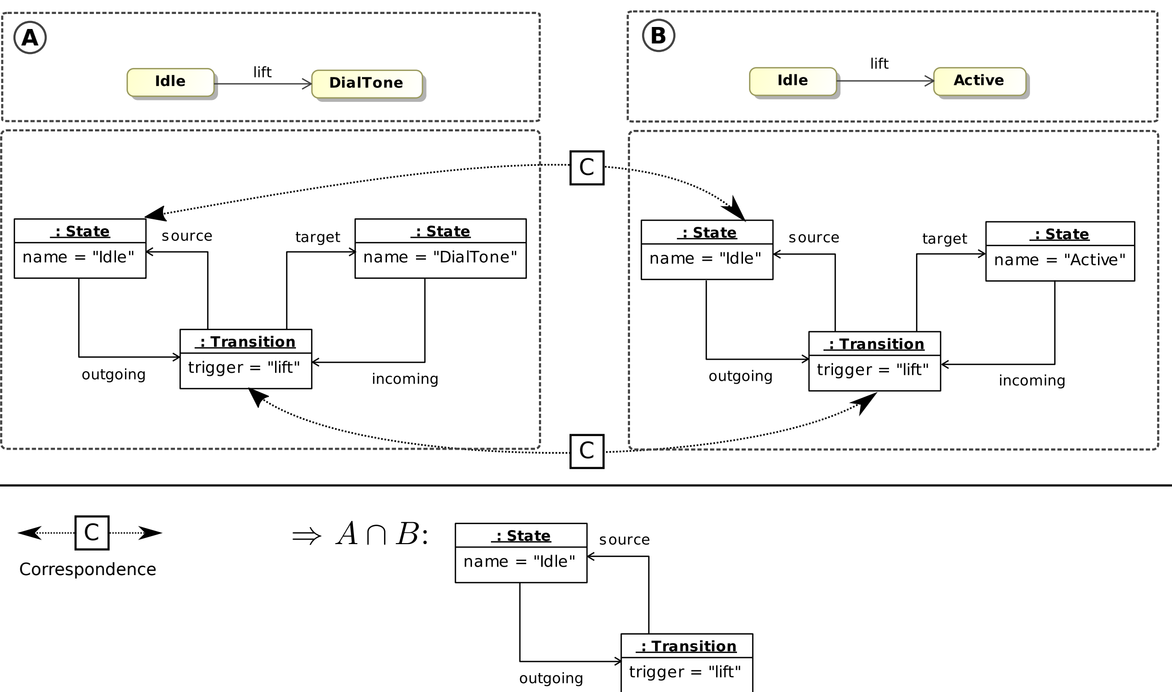


Figure 7.16: Example: Intersection $A \cap B$ over a matching $M_{A,B} : A \rightsquigarrow B$

The example of Figure 7.16 shows that it is useful to permit that mandatory neighbors of ASG nodes are not corresponding.

The same observation applies to other kinds of mandatory ASG elements, notably parent nodes which are required for all non-root nodes of an ASG[3]. If we allow correspondences between non-root nodes having different parents, i.e. the parents are not

[3]The observation also applies to attribute values which are conceptually treated like mandatory neighbors.

corresponding, then these nodes obviously have no parent nodes in $A \cap B$. However, it is generally useful to permit correspondences between nodes having different parents such that move operations can be detected.

As a conclusion of the above discussion, we can finally specify the properties which we require from a model matcher:

For every pair (A, B) of consistent models A and B which are to be compared, the matcher produces a matching $M_{A,B} : A \rightsquigarrow B$ such that the intersection $A \cap B$ over $M_{A,B}$ is a model which fulfills all basic consistency constraints. Concerning full consistency w.r.t. the effective meta-model, the following invariants may be relaxed:

1. The no-lower-bound-violation condition induced by non-containment edge types having property *required* may be violated in $A \cap B$, i.e. the mandatory neighbors of two corresponding nodes $n_A \in A$ and $n_B \in B$ may be not in a correspondence relationship.

2. A non-root node may have no parent node in $A \cap B$, i.e. the parent nodes of two corresponding nodes $n_A \in A$ and $n_B \in B$ may be not in a correspondence relationship.

Please note that, as a consequence of these relaxed consistency constraints for $A \cap B$, the change and move rules which are produced by our CPER generation algorithm must be included in the set of mandatory edit rules such that every possible modification can be expressed using CPERs. In a model difference derived from a matching, relaxation 1 leads to an exchange of a mandatory neighbor which cannot be expressed using creation and deletion CPERs. To that end, we have introduced change edit rules summarizing the effect in a single rule. Relaxation 2 leads to an exchange of a parent node which, again, cannot be expressed using creation and deletion CPERs. To that end, we have introduced move edit rules performing relocations of ASG nodes in a single edit step.

7.7 Adapting the Generated Rule Set

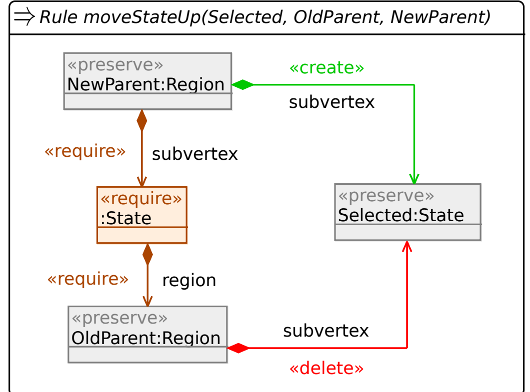
There are a few cases in which the generated edit rule set has to be manually adapted. The first one has been already explained in Section 7.3.1, namely if certain multiplicity constraints have to be relaxed in order to fulfill all conditions of a type graph with restricted multiplicities (s. Definition 7.9). In this section, we will discuss two further cases and present possible options of how to deal with them. Section 7.7.1 addresses the basic consistency constraint *no-containment-cycles* which cannot be guaranteed to be preserved by our move rules (s. Section 7.3.4). Options of how to support additional well-formedness rules, which are not yet interpreted by our edit rule generation procedure, will be discussed in Section 7.7.2.

7.7.1 Handling of Cycle-capable Containment Edge Types

Cycle-capable containment edge types would be containment cycles (or containment loops) in a flattened type graph [51]. In general, such cyclic containment structures in

meta-models are problematic since the application of move rules can lead to containment cycles in an ASG. Thus, we cannot guarantee the preservation of basic consistency constraint *no-containment-cycles* for our generated move rules (s. Section 7.3.4). In our state machine meta-model of Figure 3.4, the edge types *subvertex* and *subregion* are cycle-capable. Thus "arbitrary" movements of nodes of type *State* (resp. *FinalState*) may lead to containment cycles in an ASG.

A strategy to prevent containment cycles caused by move rules is presented in [51]: For a node which is contained by its parent via a cycle-capable containment edge, a rule may change its parent only if the old parent and the new one are (transitively) related via containment edges. This condition can be implemented by generating rules that move a node only along the containment hierarchy, i.e. rules that move a node "up" or "down". An example is shown in Figure 7.17: The move rule moveStateUp differs from an ordinary move rule as it specifies an additional application condition such that the new parent *Region* of the *State* to be moved must contain its old parent *Region* via containment edges of type *subvertex* and *region*. The rule moveStateDown is constructed in a similar way.



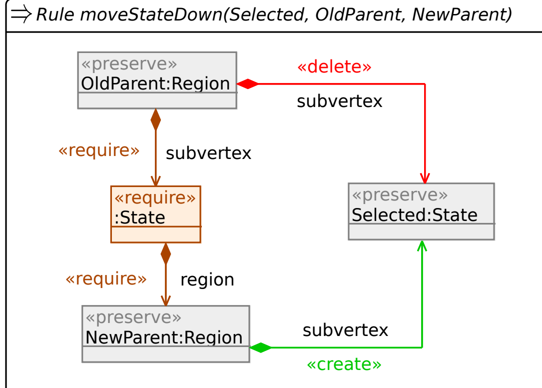


Figure 7.17: Rules moveStateUp and moveStateDown generated in addition to the basic move rule

However, applying "moveUp" (or "moveDown") several times in a sequence of edit steps may lead to transient effects. Thus, in order to not violate the soundness criterion of a mandatory set of edit rules, move rules are not replaced, but only complemented by moveUp and moveDown rules. In addition, we extend our basic differencing pipeline of Figure 1.2 such that the edit script generation is followed by an additional post-processing step. Whenever a move operation is detected, we check whether this "arbitrary" move can be replaced by a sequence of moveUp/moveDown operations shifting a node along the containment hierarchy. For any of the containment edges in this path which is created by another rule application, a dependency relationship to the respective edit step is added.

7.7.2 Supporting Arbitrary Well-formedness Rules

The edit rule generation algorithm presented in Section 7.3 is not capable of interpreting arbitrary well-formedness rules attached to a meta-model element. There are two options how to manually deal with these well-formedness rules; *i)* adapting the meta-model serving as input for or the edit rule generation (s. step ① in Figure 7.2) or *ii)* adapting the generated edit rules (s. step ③ in Figure 7.1).

Adapting the effective meta-model. For the purpose of edit rule generation, the effective meta-model may be slightly adapted before the rule generation starts. The basic idea is to rewrite certain well-formedness rules as multiplicity constraints which can be interpreted by our our edit rule generator.

For example, in order to specify well-formedness constraint (a) listed in Table 2.1 ("a final state cannot have any outgoing transitions") as a multiplicity invariant, our state machine meta-model of Figure 3.4 can be slightly modified such that the edge type *outgoing* is refined for final states, i.e. its upper bound value is set to 0. Of course, the adapted version of an effective meta-model must be fully compatible to the effective meta-model in the sense that each instance of the effective meta-model is also an instance of the effective meta-model. Moreover, the rule generation algorithm has to be slightly extended such that refined multiplicities for subtypes are properly interpreted.

Manual adaptations of generated rules. In some exceptional cases, even an adapted version of the effective meta-model contains a small set of well-formedness rules which cannot be expressed or be rewritten as multiplicity constraints. Typically, some of the generated edit rules have to be merged or complemented by additional application conditions. A quantitative assessment of the manual effort which is needed to adapt the generated edit rules in various case studies is given in Section 9.2.

An example of this is the well-formedness rule (q) listed in Table 2.1: "All members of a namespace (which includes subvertices of a region in UML state machines) must be distinguishable by their names". Hence, a state may only be created in a region if there is not yet another state with the same name. This precondition can be implemented in Henshin by a negative application condition as shown in Figure 7.18. Note that the variable Name, as a placeholder for an attribute value, appears twice in this rule.

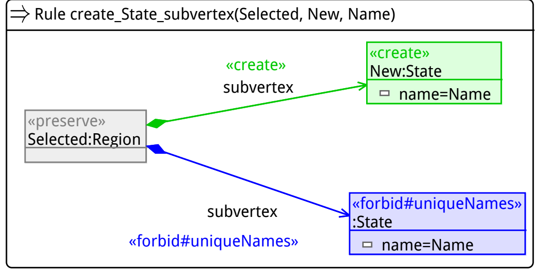


Figure 7.18: Manually adapted edit rule create_State_subvertex

CHAPTER 8

A Generic Model Versioning Framework

Difference tools for models must be adapted to each modeling language and usage scenario which is to be supported. One relevant approach are individually developed tools and algorithms which are highly optimized for a given modeling language (s. Section 10.2). This leads to a high effort. Adaptable frameworks, which can be configured with limited effort for a new language, are another, less costly approach.

In this section, we present SiLift, a generic framework for building highly customized differencing and patching tools for models which are based on the concepts developed in this thesis. The framework design is mainly driven by our domain analysis presented in Section 2.2 where variability in the problem space is modeled by the feature diagram of Figure 2.8. According to basic principles of Software Product Line Engineering (SPLE), we choose a compositional approach in order to represent the required variability in the technical solution space; most of our user-visible features are mapped to distinct implementation artifacts which can be flexibly composed to form a particular tool configuration. Thereby, solution space variability in SiLift is primarily represented on the architectural level, i.e. we propose a set of software components as compositional implementation artifacts, some of these components can be adapted by additional configuration data.

Differencing and patching tools for models are only two examples of a comprehensive MDE tool environment. Additional important tools are model editors, refactoring tools, validators, search tools, clone detection tools etc., which we assume to be available in any event. They provide several tool components, notably matchers, annotators for model elements and model transformation systems, which can be re-used to implement differencing and patching tools for models. Thus, another technically motivated goal

in the design of the SiLift framework is to integrate existing components from other MDE tools as far as possible.

The remainder of this section is structured as follows: Core components and corresponding feature mappings for the configuration of differencing tools are presented in Section 8.1, the construction of highly customized patching tools is addressed in Section 8.2. Section 8.3 finally introduces our prototypical implementation which is based on the widely used Eclipse Modeling Project (EMP) [89]. The implementation is publicly available as a set of Eclipse plug-ins from the SiLift update site[1].

8.1 Configuration of Differencing Tools

An overview of the core components for constructing difference calculation and difference presentation tools based on the SiLift framework is shown is Figure 8.1. Note that components which are re-usable from an existing MDE tool suite are colored in light gray. Corresponding feature mappings are shown in Table 8.1. Note features *Error and Conflict Detection* and *Error and Conflict Handling* are interpreted as abstract features; they are not mapped to a concrete implementation artifact but listed in Table 8.1 for the sake of readability.

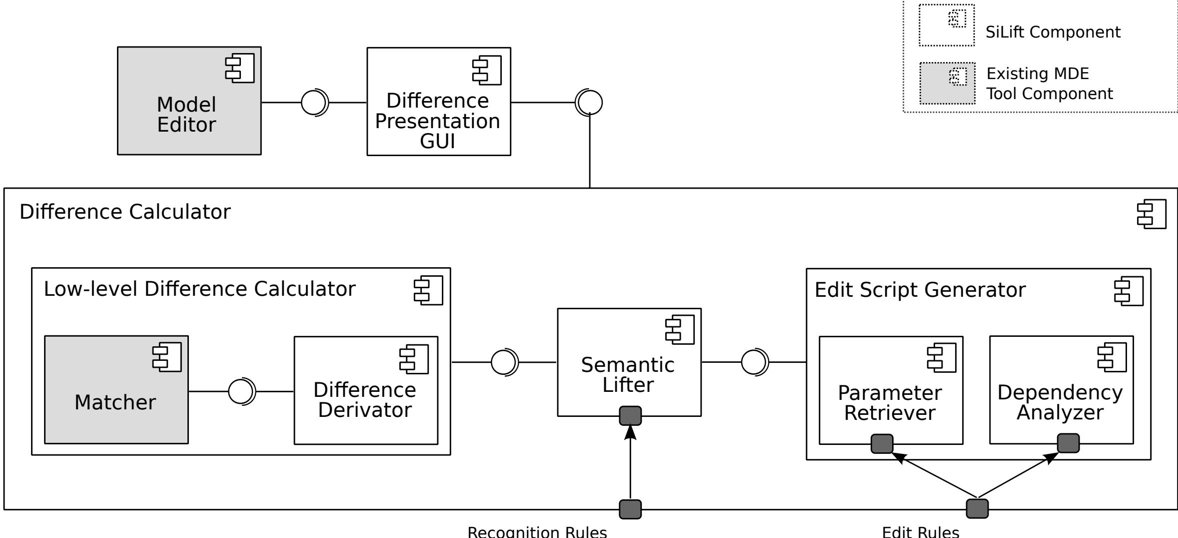


Figure 8.1: Core components for constructing a customized differencing tool

Difference calculation. A *difference calculator* is typically composed of several subcomponents that calculate a difference in a step-wise manner. Each sub-component implements a particular step of our conceptual differencing pipeline of Figure 1.2. Note that a *matcher* which is used by the *low-level difference calculator* is assumed to be

[1] http://pi.informatik.uni-siegen.de/Projekte/SiLift/updatesite

User-visible Feature	Implementation Artifact												
	Diff. Presentation GUI	Difference Calculator	Recognition Rules	Edit Rules	Model Editor	Batch-oriented Patch App.	Patch Application GUI	Patch Engine	Validator	CPEOs	Transformation Engine	Argument Manager	Modification Detector
Standard Editor					✓								
User-level Edit Operations			✓	✓						✓			
Difference Calculation		✓											
Difference Presentation	✓				✓								
Editing of Patches	✓				✓								
Change Propagation								✓					
Error and Conflict Detection													
Missing Arguments												✓	
Unfulfilled Application Conditions											✓		
Wrongly Chosen Arguments												✓	
Blind Overwriting of Changes													✓
Error and Conflict Handling													
Interactive					✓		✓						
Non-interactive						✓							
Consistency-preservation									✓	✓			

Table 8.1: Mapping user-visible features to implementation artifacts

available for the given modeling language. In principle, any matcher can be used, as long as the delivered results adhere to the basic conditions as described in Section 4.1, and as long as the properties of the matcher are consistent with the design of the mandatory edit rules serving as input for the operation recognition (cf. Chapter 7). An overview of the matching engines which are currently available within the SiLift tool suite is given in Section 8.3.2. Please also note that *parameter retriever* and *dependency analyzer* are only required if executable edit scripts are to be generated.

Difference presentation and editing of patches. Semantically lifted differences and edit scripts are intended to be presented according to the concept of an "interactive list of edit steps" (s. Section 2.2). The implementation of a difference presentation GUI, which basically lists the edit steps being part of a difference, is straightforward. The effect of an edit step is explained on the basis of the concrete syntax, original and changed model are displayed in their standard editor. Furthermore, a difference presentation GUI can be slightly extended to become an editing tool for the controlled modification of patches, an example has been presented in Figure 6.4.

In principle, such a GUI component can be integrated with any model editor. We

only require that the editor exposes an API such that external representations of model elements, i.e. diagram elements or certain text blocks, can be highlighted. Reference implementations based on standard Eclipse technologies for building visual and textual model editors are briefly outlined in Section 8.3.2.

8.2 Configuration of Patching Tools

In this section, we show how to configure patching tools for models based on the SiLift framework. We give an overview of the core components and mappings of user-visible features in Section 8.2.1. Implementation variants of certain components are discussed in Section 8.2.2.

8.2.1 Core Components and Feature Mappings

An overview of the core components of a patching tool and their integration into an existing MDE tool environment is shown in Figure 8.2. Analogously to Figure 8.1, components which are re-usable from an existing MDE tool environment are colored in light gray. The basic functionality and high-level interfaces of each component are described in the remainder of this section.

Batch-oriented vs. interactive patch application. Patches can be applied in an interactive and non-interactive way; while a *batch-oriented patch application* is useful for our versioning scenario SC2, more advanced scenarios SC3-5 are likely to be better supported by an *interactive patching tool*. Note that the latter variant uses a patch application GUI according to our concept presented in Section 6.3. Similar to a difference presentation GUI, a patch application GUI is to be integrated with a standard model editor for a given modeling language (cf. figures 6.5 and 6.6).

Patch engine. The *patch engine* is the central component of the core architecture of the patching tool. The target model and the edit script to be executed as patch must be provided as input parameters. Concrete patching tools can be built upon three basic interfaces provided by the patch engine:

1. The *execution interface* offers clients the possibility to execute edit steps contained by the edit script on the given target model. Clients can execute all applicable edit steps at once or step by step.

2. The execution (or failure) of each edit step is logged in a *patch report* which is provided to clients.

3. Finally, clients can be informed about *validation errors* being caused by the effect of an executed edit step via a callback interface.

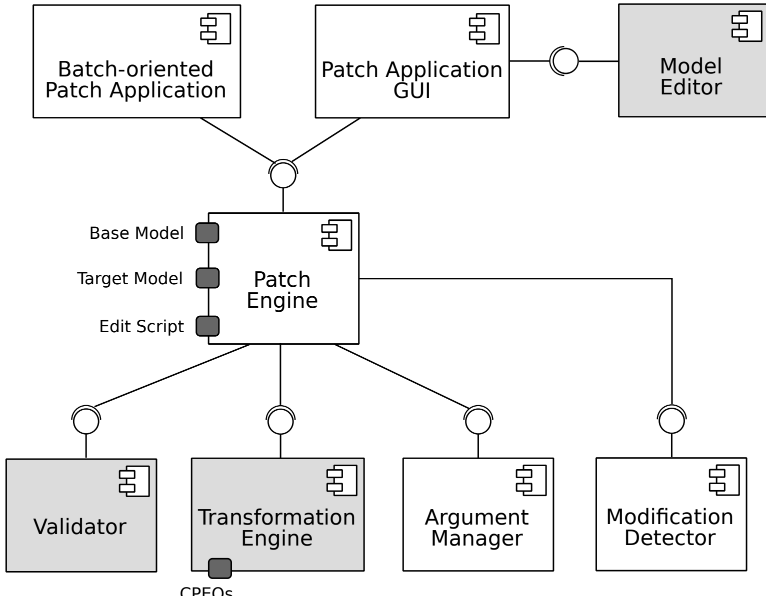


Figure 8.2: Core components for constructing a customized patching tool

Argument manager. The *argument manager* is responsible for the initial resolution of references to operation arguments and for calculating the reliability of an argument resolution. Variants for implementing a modification detector are discussed in Section 8.2.2. W.r.t. user-visible features defined by the variability model in Figure 2.8, the argument manager identifies missing operation arguments and, in case of unreliable references to model elements, detects potentially wrongly chosen arguments (s. Table 8.1). Variants for the implementation these services of an argument manager will be presented in Section 8.2.2.

Furthermore, the argument manager provides an interface to complement or modify the initial resolution of arguments. This notification interface is only required for the construction of interactive patching tools. Here, the argument manager must be continuously informed about manual modifications of the target model.

Transformation engine. Precondition checks and the execution of edit operations are finally delegated to a model *transformation engine*. Thus, the transformation engine is utilized for the detection of unfulfilled application conditions (s. Table 8.1). As shown in Figure 8.2, implementations of all CPEOs used by an edit script must be available for the transformation engine. Depending on the model transformation technology, edit operations are either

- executed by the transformation engine directly, if imperative implementations of CPEOs are provided, or

- by calling an appropriate interpreter, if a declarative approach to implementing CPEOs is chosen.

Encapsulating the concrete transformation technology has the advantage that our patching tool architecture is not affected by possible technological restrictions.

Model validator. CPEOs preserve the degree of consistency according to the effective meta-model, i.e. they guarantee that the target model remains displayable in a standard editor. Additionally, the modified target model is validated against the perfect meta-model after the execution of each edit step (s. Table 8.1). Validation errors are a kind of warning that can be generated as a result of the execution of an edit step. If desired, the respective edit step can be reverted.

The interface of a *model validator* basically defines two functions that have to be implemented: Firstly, a validation of the current state of the target model checks all well-formedness rules defined by the (perfect) meta-model of the given modeling language and returns a set of all validation errors. Secondly, a simple difference operator on sets of validation errors must be provided. Thus, the patch engine is able to compare sets of validation errors before and after each edit step, and to properly assign the reason for violating one or several well-formedness rules to the last edit step.

Modification detector. The *modification detector* is only required for the construction of a workspace update tool or, more generally, when there is a common base version being the direct ancestor of the original model and the target model of a patching scenario. It specifies the conditions when model elements shall be flagged as modified. Each modified operation argument finally leads to a warning against blindly overwriting local changes (s. Table 8.1). Variants for the implementation of a modification detector are discussed in Section 8.2.2.

8.2.2 Implementation Variants

While we have shown how to map user-visible features onto a set of re-usable components in Section 8.2.1, we discuss variability in the solution space in the remainder of this section. We present different implementation variants for two of our core components; the *argument manager* and the *modification detector*.

Argument manager. The implementation of an argument manager strongly depends on the method how references to model elements are managed in an MDE environment, notably whether we have reliable references or not. Consequently, two possible implementation variants of an argument manager are shown in Figure 8.3:

- If reliable references such as persistent identifiers or unique path names are available, references to elements used as arguments in an edit script can be symbolized at the patch creator's site (s. Section 6.1.3). These symbolic references are to be resolved at the patch applier's site.

- If no reliable references are available, our approach is to compute a matching between the original model and the target model by using a model matcher (s.

Section 6.2). Ideally, the matcher also provides information about the reliability of correspondences.

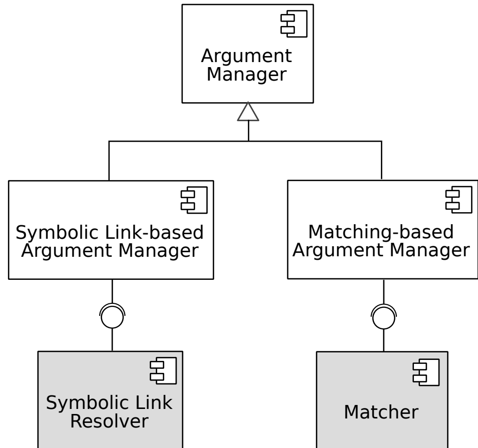


Figure 8.3: Implementation variants of an argument manager

Modification detector. Our three approaches to detect modified operation arguments in a workspace update scenario (s. Section 6.2.2) lead to implementation variants of a modification detector as shown in Figure 8.4.

- In our simplest case, a *change-based modification detector* uses a low-level difference calculator to calculate a low-level difference between the common base version and the workspace version.
- A *signature-based modification detector* re-uses a signature generator, e.g. as offered by a signature-based matcher, and a model annotator to attach calculated signatures to model elements in the base model and the workspace model. If the signature generator is adaptable, then the signature configuration is provided as input parameter
- Finally, a *comparison-based modification detector* uses the comparator of a similarity-based matcher to check workspace model elements used as arguments for modification. A set of compare functions and a comparison configuration that specifies which compare functions to use in which context are provided as additional input parameters.

8.3 Reference Implementation based on the Eclipse Modeling Project

Meta-models and ASGs can be implemented in various technical frameworks. In our reference implementation, we assume models to be implemented using the Eclipse Mod-

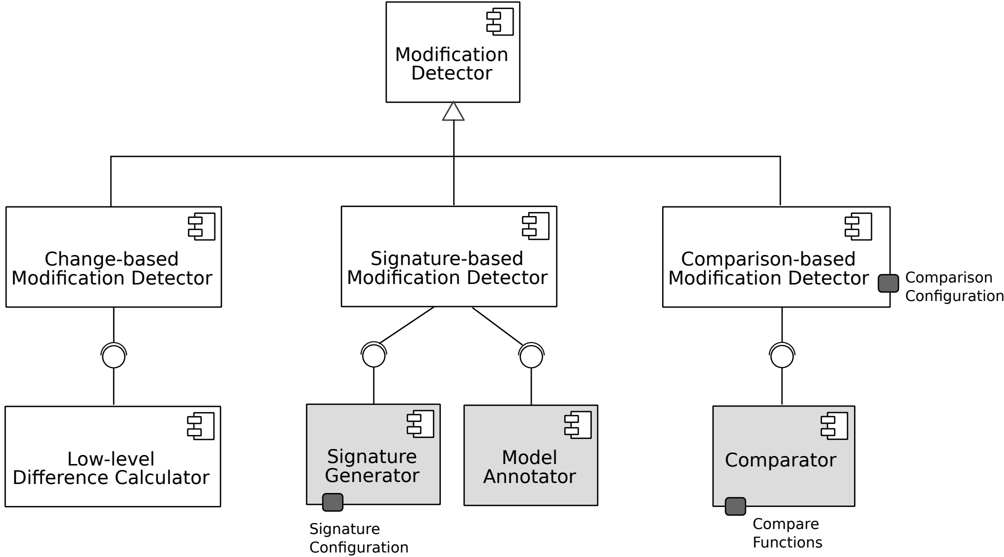


Figure 8.4: Implementation variants of a modification detector

eling Framework (EMF) [88]. EMF is one widely used technology which supports constructing runtime representations of models in the JAVA language. Section 8.3.1 briefly outlines how our graph model is mapped to implementation constructs provided by EMF. Thereupon, the implementation of the basic algorithms presented in this thesis is straightforward and not discussed in any further detail here. Of course, all concepts can be transferred to other technologies and frameworks such as MOFLON [29] or KM3 [132].

EMF itself is part of the Eclipse Modeling Project (EMP) which provides a variety of core MDE technologies being based on EMF, e.g. generators for building visual and textual editors, reference implementations of related OMG standards such as the Object Constraint Language, etc. Besides numerous projects which are officially hosted as part of the EMP, several important technologies and frameworks such as Henshin and SiDiff are implemented on the basis of EMF. Section 8.3.2 describes the integration of the SiLift core components into a comprehensive MDE tool environment being based on the EMP technology stack. Meta-tool support for generating and managing edit and recognition rules is addressed in Section 8.3.3.

8.3.1 EMF-based Implementation of Meta-models and ASGs

EMF is the de-facto reference implementation of the EMOF standard defined by the OMG [189]. Consequently, EMF provides an object-oriented approach to model representation, i.e. nodes and edges of an ASG are represented by runtime objects (called *EObjects* in EMF) and references between them. The EMF runtime framework provides a reflective API for generic read/write access to EObjects. For typing purposes, EMF provides *Ecore*, which is basically a structured data modeling language which can be used for application and language engineering purposes. In the latter case, Ecore

models serve as design-level meta-models which can be finally translated to Java type definitions using the built-in code generation facility of EMF.

Relation between type graphs and Ecore-based meta-models. Basically, node types are modeled as *EClasses* in Ecore, and edge types are modeled as *EReferences* between them. Attribute declarations are modeled as *EAttributes* which are an integral part of the definition of an EClass. Additional (meta-)modeling constructs, namely *eSuperTypes* of EClasses, *abstract* EClasses, *containment* EReferences, pairs of EReferences being declared as *eOpposite* to each other, and *lowerBounds/upperBounds* attached to EReferences can be used to further restrict the allowed instance structures.

Figure 8.5 illustrates how Ecore (meta-)modeling constructs are related to our notion of a type graph introduced in Section 3.1. Note that Figure 8.5 shows only a simplified subset of the Ecore meta-model based on [228]; we focus here on the core meta-modeling constructs and leave out *EPackages*, *EAnnotations*, *EOperations* etc.

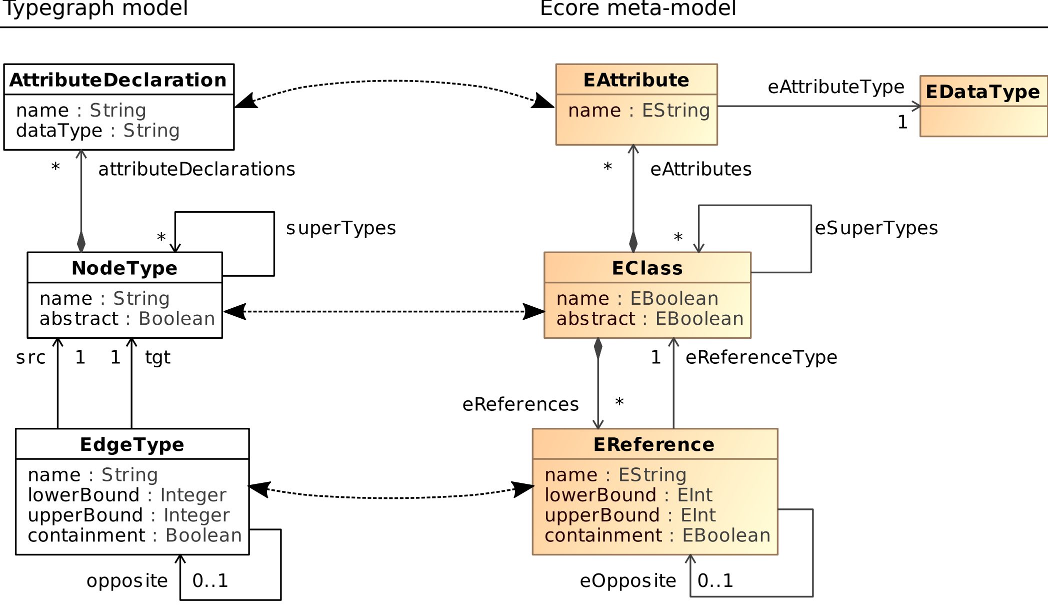


Figure 8.5: Relation between our type graph model and the Ecore meta-model

Model transformation and pattern matching with Henshin. Many concepts developed in this thesis are based on the Henshin transformation language which is based on graph transformation concepts. Thus, it was natural for us to use the Henshin transformation framework as one of the core technologies in SiLift. The Henshin runtime environment is implemented on top of EMF and uses Ecore-based meta-models for typing purposes. All of the Henshin runtime services are available via API. We use the rule interpreter to execute edit operations in terms of a patching tool. The change set recognition engine of our semantic lifter is implemented using the match finder of the Henshin runtime.

Please note that our consistency-preserving edit rules (and the derived recognition rules) represent implementations of consistent EMF model transformations which behave like algebraic graph transformations (cf. [51]). Thus, all concepts developed in this thesis can be applied to EMF models and the EMF-based implementation of the Henshin interpreter.

8.3.2 Tool Integration

In this section, we outline how to integrate the SiLift core components with other important tools and tool components being based on the EMP technology stack.

Available matching engines. The following standard model matchers are provided as built-in components at the SiLift update site:

- A reliable UUID-based matcher which is based on XMI-IDs can be used if models are serialized in the XMI format and if persistently managed XMI-IDs are available.
- A simple but efficient signature-based matcher which establishes correspondences based on equal values of name attributes. Names are available for many conceptual model element types in typical modeling languages.
- Furthermore, we provide an adapter to the matching interface of EMF Compare (EMFC) [63, 86]. This way, correspondences can be established by using the generic matching engine of EMFC or any of the customized matching engines being available as additional extensions.
- A matching adapter for the SiDiff matching engine is available upon request. The SiDiff matching engine is not included in the product configuration provided at the SiLift web site due to license restrictions.

Detection of modified arguments using the SiDiff signature generator. In order to specify the conditions when operation arguments shall be flagged as modified in a workspace update tool, we implemented a signature-based approach to modification detection. The implementation re-uses the annotation service and signature generator provided by the SiDiff model comparison framework. These SiDiff components are publicly available. The SiDiff annotation service implements a visitor pattern in order to attach arbitrary values to EObjects. A highly configurable signature generator is one of the available visitors computing the values which are to be annotated. This way, the modification detector can be configured according to our concept of signature-based detection of modified arguments presented in Section 6.2.2.

Integration with GMF-based and Xtext-based editors. GUI components of the SiLift framework, e.g. the difference presentation GUI, are loosely coupled with native editors via the *Eclipse Selection Service*. All SiLift subwindows implement the *selection provider* interface and thus report which conceptual model elements are currently selected (s. Figure 8.6). The Eclipse selection service notifies registered *selection listeners* about selection changes induced by a selection provider. We have implemented two selection listeners supporting the highlighting of external representations of model elements for two widely used technologies for developing EMF model editors:

- **GMF**: The Graphical Modeling Framework (GMF), which is part of the Graphical Modeling Project (GMP) [90], is one widely used technology for developing visual editors for EMF models. GMF provides generative components and a runtime infrastructure for building visual editors. Diagram rendering in GMF is based on the Graphical Editing Framework (GEF) [91].

  Our selection listener implementation decorates shapes and connections serving as diagram elements in a GMF-based editor. Examples of decorated diagram elements are shown in Figure 6.4.

- **Xtext**: Xtext [92] is a popular framework for the definition of textual modeling languages. Basically, Xtext adopts a grammar-based approach to language specification (s. Section 2.2.1). An Xtext grammar can be converted to an Ecore-based meta-model. Therefore, the Xtext grammar specification language provides several features (in addition to standard notations based on the Backus-Naur Form (BNF) or any of its extensions) which can be used to control the meta-model generation process.

  Our selection listener implementation is based on the Eclipse marker framework which can be used to annotate certain lines of texts (or text blocks) in all kinds of textual Eclipse editors. In order to get the position of a conceptual model element within the textual representation of a model, we utilize the Xtext contribution to the EMF adapter mechanism: For each EObject which originates from a Xtext resource, we get an adapter for this EObject providing access to the corresponding node of the parse tree of the Xtext resource. The nodes in a parse tree provide the required position information.

This solution is generic in the sense that it works for all GMF-based and Xtext-based editors. If required, the element highlighting facility can be extended to support editors which are based on other technologies as well, as long as an external interface which can be utilized for element highlighting is available.

8.3.3 Meta-tool Support

In this section, we briefly outline the most important meta-tools and tool components being integrated into the SiLift development environment.

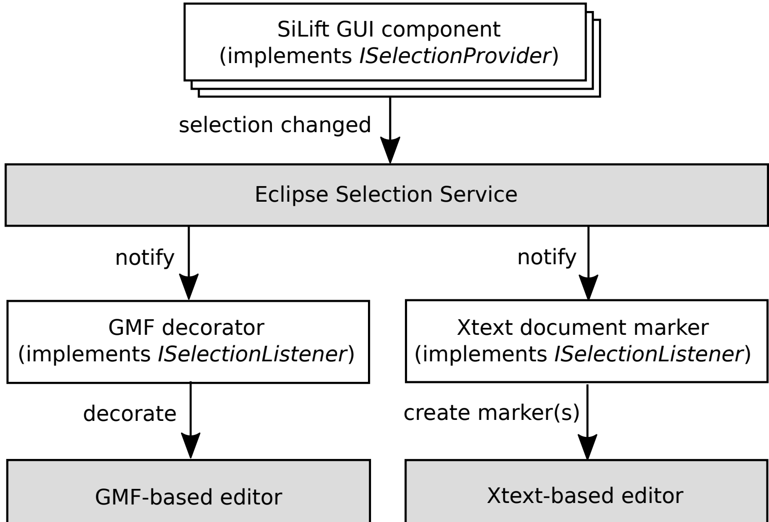


Figure 8.6: Integration of SiLift GUI components with GMF-based and Xtext-based editors

Generation of CPERs with the SiDiff Edit Rules Generator. In Chapter 7, we introduced a semi-automated approach to create a set of mandatory edit rules for a given modeling language. This process, namely the task of generating an initial set of CPERs (step 2 in Figure 7.1), is supported by the SiDiff Edit Rule Generator (SERGe) [10]. SERGe is implemented on the basis of EMF and Henshin and derives sets of edit rules from a given Ecore-based meta-model. If configured properly, SERGe derives all kinds of CPERs, namely creation, deletion, move and change rules, according to our edit rule generation procedure presented in Section 7.3.

In addition, SERGe provides ad-hoc support to generate edit rules for profiled UML models [5]. Although profile support is yet limited to simple stereotype structures, e.g. a UML base element may have at most one associated stereotype, standard profile definitions such as SysML or MARTE are covered to a large extent.

Edit and recognition rule management. The manual adaptation of generated edit rules and the construction of optional complex rules is supported by the Henshin development environment. In particular, the Henshin IDE provides a sophisticated editor which is based on the visual syntax of the Henshin transformation language. A technical convention is that each Henshin rule representing and edit operation implementation must be encapsulated by a transformation unit which represents the corresponding edit operation interface. The reason behind this convention is that a transformation rule treats internal variables as well as input and output parameters in a uniform way. On the contrary, a unit may define only the externally visible formal parameters; input parameters are mapped from the unit to their corresponding rule parameter, while output parameters are mapped in the opposite direction. We provide integrated validation and quickfixing techniques to support the development of edit operations adhering to

these conventions.

Edit rules are statically analyzed for potential dependencies and conflicts using the CPA extension of Henshin [53] which is based on the graph transformation environment AGG [232]. Dedicated export and import functions translate Ecore-based meta-models and Henshin rules to AGG and CPA results back to EMF.

If desired, additional configuration parameters can be attached to edit rules. These annotations are to be interpreted by our procedure translating edit rules to their corresponding recognition rules (s. Section 4.2.2). We support the following configuration parameters:

- **Pre/Post**: Each application condition can be labeled “pre” or “post” to determine whether the condition is to be interpreted as pre- or postcondition by our edit to recognition rule translation (cf. Section 4.2.2). By default, we assume all application conditions to be interpreted as post-conditions, i.e. they are to be checked on the revised model version during operation detection.

- **Priority**: A priority value can be assigned to each edit rule. Thus, ambiguities can be resolved if several semantic change sets contain exactly the same low-level changes and the ambiguity can not be resolved by the postprocessing algorithm described Section 4.3. By default, we assume a priority value of 0.

To enable a seamless translation of edit rules to recognition rules, we integrated our transformation procedure into the build framework provided by the Eclipse IDE. This way, recognition rules are treated as binaries, a tool chain configurator will never see the compiled recognition rules. To support an incremental build process, each edit rule must be defined in a separate Henshin file being available as workspace resource. Whenever an edit rule file is being modified, our recognition rule builder will be triggered to recompile the corresponding recognition rule.

CHAPTER 9

Evaluation

In this chapter, we evaluate the approach presented in this thesis from two different perspectives: *i)* From the point of view of an *MDE tool chain configurator*, the evaluation should show the *suitability* of the proposed framework; *ii)* from the point of view of a *model versioning tool user*, we are interested in certain *quality aspects* of individual tools and tool functions being implemented on top of the framework. Evaluation goals for both perspectives are presented in Section 9.1. Concerning *i)*, the suitability of the generic framework has been evaluated in several case studies (s. Section 9.2). Concerning *ii)*, experimental results assessing quality aspects for a set of representative tool configurations and data sets are presented in Section 9.3.

9.1 Evaluation Goals

In this section, we present the evaluation goals related to the perspectives (tool chain configurator vs. model versioning tool user) from which we evaluated our approach.

Suitability of the proposed framework. Concerning the suitability of the proposed framework (cf. contribution 5 in Section 1.4), we identified two main evaluation goals:

G_1 *(Flexibility):* The evaluation should show that the framework is flexible in the sense that it enables the construction of a family of tools that cover a broad range of different use cases and application contexts. Flexibility includes *(a)* that the

framework components are highly re-usable and *(b)* that the proposed methodology mapping user-visible features to implementation components is feasible.

G_2 *(Adaptability):* Engineering individual tools and tool functions using the SiLift framework should be possible with moderate time and effort. This particularly refers to the configuration of the generic tool components for a given modeling language.

Quality aspects of individual tool functions. From the point of view of a model versioning tool user, we are mainly interested in the evaluation of certain quality aspects of individual tools and tool functions being implemented on top of the SiLift framework. The framing of which quality aspects have to be evaluated is mainly driven by the claims of thesis contributions 1 - 4 (cf. Section 1.4).
For the semantic lifting of low-level model differences, we identified the following quality aspect:

Q_1 *(Understandability of model differences):* Semantically lifting a low-level model difference to the level of user-level edit operations increases the *understandability* of the difference.

Furthermore, we identified the following quality aspects for generated edit scripts which are intended to be used as patches:

Q_2 *(Correctness of edit scripts):* An edit script is *correct*, i.e. the application of an edit script $\Delta_{A \Rightarrow B}$ to model version A finally results in model version B.

Q_3 *(Consistency-preservation of edit scripts):* Given a consistent target model to which an edit script is applied, then each edit step is consistency-preserving, i.e. each intermediate model version as well as final model are consistent. The minimal level of consistency which is to be achieved is defined by the effective meta-model of a given editing environment.

Q_4 *(Abstraction level of edit scripts):* An edit script raises the *level of abstraction*, i.e. the algorithm finds as many complex, semantically rich operations as possible.

Finally, contribution 4 provides a completely new approach to update workspace copies which differs methodologically and technically from traditional 3-way merging. While the reduced configuration effort and the much simpler GUI of an interactive update tool are obvious, we identified the following quality aspect w.r.t. to our approach to conflict detection:

Q_5 *(Accuracy of conflict detection):* The accuracy of the conflict detection facility of our approach should be fairly equal to traditional 3-way merging.

9.2 Case Studies and Example Applications

Based on the SiLift framework, a set of individual tools and tool functions with different characteristics and targeting different usage scenarios have been developed in several case studies (s. sections 9.2.1-9.2.6). An overview of the respective tool configurations (in terms of the selected user-visible features) is presented in Table 9.1. Language-specific implementation variants of technical variation points as well as configuration artifacts that have been developed to adapt generic tool components are summarized in Table 9.2. Conclusions concerning the achievement of our main goals G_1 and G_2 are drawn in Section 9.2.7. A critical discussion w.r.t. the limitations of the approach and threats to validity is included.

		User-visible features												
Case Study	**Configuration**	Visual	Textual	Elementary Operations	Complex Operations	Difference Presentation	Editing of Patches	Missing Arguments	Unfulfilled Appl. Cond.	Wrongly Chosen Arguments	Blind Overwriting of Changes	Interactive	Non-Interactive	Consistency-preservation
Study I	$Ecore_1$	✓		✓	✓	✓	✓							
	$Ecore_2$	✓		✓	✓			✓	✓	✓		✓	✓	✓
	$Ecore_3$	✓		✓	✓			✓	✓	✓	✓	✓	✓	✓
Study II	$SysML_1$	✓		✓		✓	✓							
	$SysML_2$	✓		✓				✓	✓				✓	✓
Study III	FM	(✓)		✓	✓	✓								
Study IV	$SWML_1$	✓		✓	✓	✓								
	$SWML_2$		✓	✓	✓	✓								
Study V	$JavaAST$	(✓)		✓	✓									
Study VI	SA	(✓)		✓	✓									
	FT	(✓)		✓	✓									

Table 9.1: Feature configurations of individual tools and tool functions which have been developed in different case studies

9.2.1 Study I: Comparison and Versioning of Ecore Models

Ecore is the most commonly used model type in the context of the Eclipse Modeling Project. It is used for two main purposes: In the context of language engineering, it used as a technique to define design-level meta-models (s. Section 8.3.1). In the context of application engineering, it is used for the purpose of object-oriented modeling. The visual diagram editor offers a concrete syntax which is similar to design-level class diagrams known from the UML.

	Configuration Data & Implementation Variants						
Language	Matcher	Mandatory Edit Rules[1]	Optional Edit Rules	Validator	Transformation Engine	Argument Manager	Modification Detector
Ecore	{UUID, EMFC}	134/14	58	EMFV	Henshin	{Symbl., Match.-based}	Sign.
SysML	UUID	788/56	-	EMFV	Henshin	Match.-based	Sign.
FM	Sign.	8/11	30	-	-	-	-
SWML	Sign.	29/-	5	-	-	-	-
JavaAST	Sim.	174/2	12	-	-	-	-
SA	Sign.	42/-	-	-	-	-	-
FT	Sign.	70/-	-	-	-	-	-

[1] #generated/#manually specified

Table 9.2: Language-specific adaptations: Configuration of generic tool components and implementation of technical variation points

Tool configurations. Because of its central role and the wide range of usage scenarios within the EMP, we decided to develop a set of standard versioning tools adapted to EMF Ecore. They are provided at the SiLift update site. All of the model versioning scenarios presented in Section 2.1 are supported by the following individual tool configurations:

$Ecore_1$*:* Concerning scenario SC1, we provide a differencing function to calculate semantically lifted differences which, if desired, can be converted to executable edit scripts to be used as patches. Differences can be visually inspected in our standard difference viewer. Moreover, edit scripts which are intended to be used as patches can be modified in a controlled manner.

$Ecore_2$*:* Traditional and advanced patching scenarios SC2-4 are supported by a patching tool which can be operated in interactive and non-interactive mode. The operation mode can be chosen at runtime (cf. Table 9.1). All types of patching-related errors being defined by our variability model of Figure 2.8 can be handled.

$Ecore_3$*:* Updating workspace copies of Ecore models is supported by an extension of the interactive patching tool. In addition to tool configuration $Ecore_2$, the problem of blindly overwriting local workspace changes is also handled.

Configuration data and implementation variants. Configuration data to adapt generic tool components are shared among tool configurations $Ecore_{1-3}$. Moreover, the same implementation variants are used for technical variation points (s. Table 9.2):

Matcher: All standard matchers described in Section 8.3.2 can be used for Ecore models. The most suitable one can be selected at runtime, depending on the nature of input models.

Edit rules: The edit operation detection engine has been configured with two sets of edit rules, implementing the mandatory and optional edit rules, respectively. In sum, we identified 148 mandatory edit rules for Ecore models, 134 of them could be fully generated by our edit rule generator. 14 mandatory edit rules had to be manually engineered, most of them are only slightly adapted versions of the edit rules which were initially created by the edit rule generator. Additionally, 58 complex edit operations have been implemented manually. Most of them are taken from the catalogue of object-oriented refactorings presented in [107]. Another source of inspiration was the Ecore configuration of the model refactoring tool EMFRefactor [32]. Finally, we implemented several evolutionary edit operations, all of them are motivated by own experiences in object-oriented modeling with Ecore.

Validator and transformation engine: Implementation variants for several core patching components could be re-used from existing MDE tools and technologies. Consistency-checks against the perfect level of consistency are performed using the EMF validation framework (EMFV). The Henshin runtime environment and interpreter is used for the execution of edit operations.

Argument manager: We provide two implementations of an argument manager, the concrete variant can be dynamically selected at runtime. The first implementation is based on the principle of symbolic references (cf. Section 8.2.2). The second one is matching-based and uses the SiDiff matching engine; it re-uses the configuration data developed in [259] to estimate the reliability of a reference resolution.

Modification detector: For the interactive update tool ($Ecore_3$), we choose a signature-based approach to specify the conditions when model elements shall be flagged as modified. In our own experiments conducted so far, a rather simple configuration has been used for all types of model elements. The signature incorporates (a) the values of *all attributes* attached to the respective node in the ASG, (b) the values of *all attributes attached to all directly referenced ASG nodes* and (c), if available, the element's *qualified name*. This relatively simple and compact specification has turned out to be very powerful (s. Section 9.3.3).

Example applications. Tool configuration $Ecore_1$ has been used in several tool demonstrations and tutorials, see e.g. [3, 12]. The interactive patching tool ($Ecore_2$) has been used in [4] in order to illustrate our approach to model patching. The workspace update tool ($Ecore_3$) has been used in a formal tool demonstration [8], a screencast of this demonstration can be found at the SiLift website[1] In addition, These configurations

[1] `http://pi.informatik.uni-siegen.de/Projekte/SiLift/ase2014.php`

of our semantic lifting engine and the edit script generation service are used in the experiments presented in sections 9.3.1 and 9.3.2.

9.2.2 Study II: Model Variant Management in Industrial Plant Automation

Nowadays industrial plants are software-intensive systems which are typically operated and evolved for several decades. First research results show that the MDE paradigm can be successfully applied to industrial plant automation in order to deal with the complexity of evolution [19, 255]. Promising approaches are based, e.g., on the Systems Modeling Language (SysML) [191]. SysML is defined as an extension of a subset of the UML using the UML profile mechanism.

As already explained in Section 2.1.4, a particular challenge in the model-based engineering of automation systems are model modifications during operation and maintenance at a specific customer's variant [253]. Such changes are (1) often not sufficiently documented with respect to the original version and (2) may include changes of general interest that need to be propagated to other variants or a centrally managed module kit [103]. In [9], we propose a method which enables developers to conveniently propagate changes between variants of a model.

Tool configurations. Technologically, the approach presented in [9] is based on the principle of document patching. Thus, two kinds of individual tools have been configured based on the SiLift framework:

$SysML_1$*:* The differencing function has been configured to calculate executable edit scripts which are intended to be used as patches. Patches can be visually inspected and adapted in our standard patch editor.

$SysML_2$*:* The propagation of changes to another variant or a central module kit is supported by an interactive patching tool.

Configuration data and implementation variants. We have successfully adapted all generic components to SysML models that have been developed with the modeling tool Papyrus [93, 208]. We support a subset of the available element types, namely those parts of the SysML which are visually represented in terms of block definition diagrams (BDDs), requirement diagrams (RDs) and state charts (re-used from the UML):

Matcher: Our UUID-based matcher which is based on XMI-IDs is used for the purpose of model matching.

Edit rules: The set of mandatory edit rules comprises 844 edit rules, 788 of them could be generated with our edit rule generator. Most of the 56 manually engineered edit rules deal with the handling of associations since the SysML re-uses the rather complex specification of associations from the UML Superstructure [190].

For example, rules for the creation (and deletion) of different kinds of binary associations (navigable in both directions, one direction or not navigable at all) between different kinds of stereotyped classes (e.g. blocks, requirements, etc.) had to be implemented manually.

We left the identification of further complex edit operations for future work. One option is to adopt well-known object-oriented refactorings which we have already successfully used in configurations for other modeling languages, e.g. for Ecore (s. Study I) and JavaAST (s. Study V). A second option is to identify meaningful evolutionary operations which are specific to the domain of automation engineering. The primary source for identifying useful evolutionary edit operations are experienced users who need to be interviewed.

Core patching components: Analogously to our adaptation to Ecore in Study I, the EMF validation framework is used to perform consistency-checks against the perfect level of consistency, edit operations are executed by interpretation using the Henshin transformation engine. The argument manager is matching-based; it adapts the UUID-based matcher.

Example applications. We evaluated the applicability of our approach using one of the main case studies of the DFG Priority Programme SPP1593 [116]; the Pick and Place Unit (PPU) [254]. Experimental results of studying the evolution of a SysML model of the PPU are presented in Section 9.3.1.

9.2.3 Study III: Documenting and Reasoning about Feature Model Changes

In this case study, a tool for differencing feature models has been developed [20]. A specific contribution of [20] are complex edit operations whose semantic impact on the set of valid feature configurations can be classified semantically as refactoring, generalization or specialization of a feature model.

Motivation and general description. When a product-line evolves in order to react to changing requirements, the feature model which formally specifies the valid configuration space has to be modified, too. Thus, product-line engineers are often faced with the problems that *i)* feature models are changed ad-hoc without proper documentation, and *ii)* the semantic impact of syntactic changes is unclear. In [20], we propose a comprehensive approach which is based on model differencing techniques presented in this thesis in order to tackle both challenges. For *i)*, our approach compares the old and new version of the diagram representation of a feature model and specifies the changes using meaningful user-level edit operations on feature diagrams. Hence, feature model changes are automatically detected and documented formally. For *ii)*, we propose an approach for reasoning about the semantic impact of diagram changes. The idea is to statically classify the available user-level edit operations w.r.t. their effect on

the set of valid feature combinations (s. edit categories proposed by Thüm et al. [239]): A *refactoring* leaves this set unchanged, i.e., it transforms a feature diagram into a semantically equivalent diagram. A *generalization* (*specialization*) enlarges (shrinks) the set of valid feature combinations. If none of these categories applies, an edit operation is classified as *arbitrary edit*. Thus, high-level changes being observed between two versions of a feature diagram can be classified w.r.t. the above categories, too. Knowing the semantic impact of feature diagram changes is very useful for further development tasks, e.g., in the field of regression testing.

Tool configuration and adaptation to feature models. To support the above methodology, a structural differencing tool for feature models has been implemented on top of the SiLift framework (s. configuration FM in Table 9.1). The differencing results are visualized within a slightly modified variant of our standard difference display GUI; additional information about the semantic impact is attached to feature diagram changes. Yet, the feature diagram versions being compared with each other are displayed in the rather simple tree-based EMF editor. A full integration into a widely used feature modeling environment such as FeatureIDE [240] has been left for future work.

Matcher: Currently, we use a rather simple feature diagram matching strategy which is similar to [21]. Features are matched based on their unique names. Constraints and groups are matched by calculating signatures that characterize these elements: For cross-tree constraints, a signature value is computed by concatenating the names of the related features. Groups are enumerated within each parent feature serving as group context.

Edit rules: The difference calculation procedure uses edit rules being implementations of edit operations from the catalogue developed in [20]. In sum, we have 19 mandatory edit rules. In order to classify these rules according to the categories proposed by Thüm et al. [239], a considerable large number of 11 edit rules had to be specified manually. Most often, the generated rules had to be complemented by additional application conditions.

Complex edit operations are primarily derived from observations in the evolution of an existing product line. We also gathered typical editing scenarios through expert interviews. In sum, we identified 30 complex edit rules comprising 11 refactorings, 8 generalization operations, 8 specialization operations and 3 arbitrary edits.

The whole catalogue of edit operations, implementations of these edit operations in Henshin as well as the tool itself are available at the SiLift website[2].

[2] `http://pi.informatik.uni-siegen.de/Projekte/SiLift/asejournal2014.php`

9.2.4 Study IV: Understanding Complex Changes in Domain-Specific Models

As already mentioned in the introductory part of this thesis, the use of domain-specific modeling languages (DSMLs) is an important trend in MDE. In contrast to general-purpose modeling languages such as the UML, DSMLs typically have a small scope and formalize the key concepts of a particular domain of interest. Obviously, tool support for the version and variant management of domain-specific models is strongly required, too.

Example application. In [11, 13] we show how to customize our approach to support high-level model differencing for an example DSML called *Simple Web Modeling Language (SWML)* [54]. SWML aims at defining platform-independent models for a specific kind of web applications. Simple web models can be specified using either a visual or a textual notation. In the literature, both notations are used in slightly different variations [54]. In our tutorial, we use the Ecore-based meta-model presented in [33]. Concerning the specification of a textual SWML syntax, an Xtext grammar which can be translated to the same SWML meta-model is also presented in [33].

Tool configurations and adaptation to SWML Differencing of SWML models is supported by tool configurations $SWML_1$ (for visual SWML models used in [13]) and $SWML_2$ (for textual SWML models used in [11]) of Table 9.1. In both cases, differences can be displayed in our standard difference viewer. Since there is a single meta-model that defines the conceptual structure of visual and textual SWML models, both tool configurations are adapted using the same configuration artifacts:

Matcher: To determine corresponding elements in SWML models, we implemented a simple signature-based matching strategy which is similar to the one for feature models. Almost all types of elements have a name attribute whose value can be utilized for the purpose of model matching, i.e. names are used as unique signature-values. Nameless elements can be reliably matched based on the correspondences of context elements.

Edit Rules: As shown in Table 9.2, the SWML meta-model leads to 29 mandatory edit rules, all of them could be generated by SERGe. In addition, 5 refactoring operations have been specified to improve the quality of SWML models if certain bad smells are detected [33].

9.2.5 Study V: Statistical Analysis of Changes in Evolving Software Models

Analyzing the history of a software system is a widely used method in the context of mining software repositories. In this case study, we used our differencing functionality as a basis for the statistical analysis of model evolution in terms of the applied edit operations.

Motivation and general description. Adequate test models for evaluating all kinds of MDE tools are scarcely available in many application domains. In this regard, test model generators have been proposed as a vehicle to artificially create the required test models. Basically, a model generator such as the SiDiff Model Generator (SMG) generates test models by modifying a base model (which may be the empty model) using a pre-defined set of edit operations. In many cases, the applied changes should resemble the evolution that one observes in real-world models. This information can be obtained by statistically analyzing model histories with methods as proposed in [14, 16, 17]. A basic prerequisite for the application of these statistical methods is to properly capture the evolution of a model in terms of the applied edit operations.

Example application to JavaAST models. In the work presented in [14, 16, 17], we automatically reverse-engineered nine open source Java software systems of the "Helix-Software Evolution Data Set" [251] and derived design-level "class diagrams" (referred to as JavaAST in Tables 9.1 and 9.2) from the Java source code. The complete JavaAST meta-model is available at the accompanying web site[3]. For each pair of revisions of the history of each of these systems, the reverse-engineered models were compared using the following configuration [15]:

Matcher: Since the reverse-engineered models do not have persistent identifiers, we used the similarity-based matching facility of the SiDiff framework. The configuration of the matching engine is similar to an older configuration for UML class diagrams [138].

Edit Rules: In sum, we have a total number of 188 edit rules for JavaAST models. 176 of these edit rules are mandatory edit rules, 174 of them could be generated. Additionally, we chose the well-known catalog of object-oriented refactoring operations [107] and selected 12 refactorings being applicable to design-level class diagrams.

9.2.6 Study VI: Analyzing the Co-Evolution of Interrelated Models

The multi-view paradigm often leads to the problem to consistently co-evolve a set of semantically interrelated models. In this regard, our approach presented in [18] proposes to observe the history of semantically interrelated models in order to learn about correlating changes in the co-evolving models. The procedure is supported by a co-evolution analysis framework whose implementation is based on SiLift; it uses the difference calculation function to obtain the edit steps which have been applied to evolve each of the interrelated models.

An example of two kinds of loosely coupled but semantically interrelated models are software architecture (SA) and fault tree models (FT) [112]. To understand the seman-

[3] http://pi.informatik.uni-siegen.de/qudimo/smg/se2013

tic relation between SA and FT models better, we adapted the difference calculation procedure of the analysis framework to both kinds of models:

Matcher: Both for SA and for FT models, correspondences are determined using a simple signature-based matching strategy. Similar to feature models and our SWML configuration, correspondences between named elements are established based on equality (and uniqueness) of names. Anonymous model elements representing relationships between named elements are matched if the related elements are matched.

Edit Rules: As shown in Table 9.2, our current analysis procedure which has been applied to a sample evolution of SA and FT models is based on sets of generated edit rules only. In sum, 112 mandatory edit rules have been generated by SERGe, 42 for SA models and 70 for FT models.

9.2.7 Conclusions and Critical Discussion

Achievement of main goals. Concerning the flexibility of the proposed framework (evaluation goal G_1), the case studies show that an MDE tool chain configurator can conveniently provide a set of versioning tools and differencing functions which are specifically tailored for a certain usage scenario. Our generic integration of SiLift GUI components with standard editors which are based on GMF, Xtext or the reflective EMF tree editor successfully worked for all configurations. Moreover, several important components for implementing technical variation points of our framework could be re-used from other existing model difference tools or general MDE tools.

The most important variation point is the modeling language for which a particular difference/versioning tool is being adapted. In this regard, the adaptability of the framework (evaluation goal G_2) has been demonstrated in several applications using a variety of different modeling languages and sub-languages. The typical effort to configure the generic components being used in a tool configuration ranges between 1 and 10 days, primarily depending on the size and the complexity of the (effective) meta-model. In particular, sets of mandatory edit rules could be generated to a large extent. An exception of this are edit operations on feature models which have been developed in Case Study III. The reason for this is more than 50% of the generated edit rules had to be supplemented by additional application conditions in order to statically classify them w.r.t. their semantic impact on the set of valid feature combinations. However, this is highly specific to this special usage scenario and does not apply to typical versioning scenarios and modeling languages.

One limitation of our approach pertains the implementation of complex edit operations: We only support edit operations which can be specified by one Henshin transformation rule or by an interaction scheme between a kernel rule and a set of (nested) multi-rules. Moreover, application conditions have to be specified in terms of static graph patterns. Thus, some refactoring operations from the catalogue of object-oriented refactorings presented in [107] could not be implemented at all.

Threats to validity. A threat concerning the external validity of our conclusions w.r.t. evaluation goal G_1 is that the set of tool configurations being used in our example applications represents a rather small subset of all possible configurations. Ideally, we would have to create an individual tool or tool function for each valid feature combination defined by our feature model (s. Figure 2.8) which formally documents the results of our requirements analysis. However, it is obvious that implementing and testing all possible configurations is infeasible. The issue one can argue about is that we did not apply a systematic methodology for selecting the set of tool configurations which have been implemented and tested (see e.g. [166, 167] for different strategies and coverage criteria in the field of SPL testing). Instead, the selection was mainly driven by the individual requirements of each case study. However, due to the diversity of the conducted case studies, we are convinced that the selected sample is a representative one.

A threat concerning the external validity of our conclusions w.r.t. evaluation goal G_2 is the selection of modeling languages for which our generic tool components have been adapted. However, the characteristics of the supported model types and the way how instances of these types are edited differ significantly from each other: Ecore is the de-facto standard for object-oriented modeling within the Eclipse Modeling Project. SysML is a visual modeling language which is a widely accepted and standardized notation in the context of model-based systems engineering [255]. It re-uses large parts of the UML and comes along with a comprehensive meta-model. Besides these standard languages, we implemented configurations for several domain-specific languages: Feature models are a widely used approach for modeling variability in the context of software product line engineering. SWML is a DSML that comes with a textual and visual notation and has been used in several case studies. Software architecture models used in [112] incorporate well-known concepts from architecture description languages (ADLs) [237]. Fault tree models are commonly used as quality evaluation models in safety-critical domains [118]. Thus, we are convinced that the chosen modeling languages are good representatives to show the adaptability of our approach.

9.3 Experimental Results

Table 9.3 illustrates the relations between quality aspects Q_1 - Q_5 as introduced in Section 9.1 and our motivating versioning scenarios SC1 - SC5 of Section 2.1 in which these these qualities are required. Quality aspects Q_1 - Q_4 have been evaluated using test models which have been obtained from model histories provided by empirical case studies on model evolution. The model histories gathered from these case studies perfectly meet our requirements since they provide a large set of test cases for differencing EMF-based models, thus serving as realistic real-world test data. Results are presented in Section 9.3.1 (for Q_1) and Section 9.3.2 (for Q_2 - Q_4). Quality aspect Q_5 has been evaluated using a conflict detection benchmark (s. Section 9.3.3).

	Usage scenario				
Quality aspect	SC1	SC2	SC3	SC4	SC5
Q_1: Understandability of model differences	✓		✓	✓	✓
Q_2: Correctness of edit scripts		✓	✓	✓	✓
Q_3: Consistency-preservation of edit scripts		✓	✓	✓	✓
Q_4: Abstraction level of edit scripts			✓	✓	✓
Q_5: Conflict detection					✓

Table 9.3: Quality aspects and related usage scenarios

9.3.1 Semantic Lifting of Model Differences

To evaluate our approach w.r.t. quality aspect Q_1, we performed an experiment using a model history from a case study for the domain of production systems in automation engineering. Each pair of successive model revisions $v_n \rightarrow v_{n+1}$ (in the following referred to as *evolution step*) in a model history provides a test case for which both a low-level difference $\delta_{v_n,v_{n+1}}$ and a semantically lifted difference $\Delta_{v_n,v_{n+1}}$ have been calculated.

Case study. Our test data set is taken from the PPU case study [254] (cf. Section 9.2.2). The PPU ("Pick and Place Unit") is a laboratory plant which consists of five main components: a stack, a crane, a stamp, a ramp and a conveyor. It can process different kinds of workpieces. The PPU case study provides 14 so-called evolution scenarios of the PPU. Each evolution scenario describes a specific configuration of the unit, and each configuration is documented using the SysML. Since the evolution scenarios of the PPU have been developed in a sequential order, we have a history of 14 revisions of a SysML model of the PPU.

Test procedure and results. Each evolution step $v_n \rightarrow v_{n+1}$ of the PPU model history serves as a test case for the calculation of a semantically lifted difference. To that end, our lifting engine has been adapted to SysML as shown in Table 9.2.

Basic measures for each test case[4] are shown in Table 9.4: Column *#Corresp.* refers to the number of correspondences in a matching $M_{v_n,v_{n+1}} : v_n \rightsquigarrow v_{n+1}$. Column *#Low-level* denotes the number of low-level changes in $\delta_{v_n,v_{n+1}}$ derived from the matching $M_{v_n,v_{n+1}}$, thus indicating the extent to which the compared models differ from each other. The number of semantic change sets in a semantically lifted difference $\Delta_{v_n,v_{n+1}}$ is listed in column *#Change-sets*.

To evaluate the extent to which semantically lifting a low-level model difference increases the understandability of a difference, the *compression factor* serves as an indicator of how much a user's perception of a difference is improved. It is defined as follows:

[4] Test cases 05 $\rightarrow$ 06 and 13 $\rightarrow$ 14 are missing due to missing SysML models for the evolution scenarios SC06 and SC14.

$$cf = \frac{|\delta|}{|\Delta|} \tag{9.3.1}$$

where $|\delta|$ refers to the number of low-level changes and $|\Delta|$ denotes the number of recognized semantic change sets. In other words, the compression factor measures how many low-level changes are, on average, grouped by a semantic change set. As shown in column *Compression* of Table 9.4, we observed compression factors between 2.00 and 3.68 for the evolving SysML models of the PPU.

Test case	#Corresp.	#Low-level	#Change-sets	Compression
SC00 → SC01	545	58	16	3.63
SC01 → SC02	545	203	86	2.36
SC02 → SC03	575	764	231	3.31
SC03 → SC04	774	6	3	2.00
SC04 → SC05	756	654	201	3.25
SC05 → SC07	904	165	102	1.62
SC07 → SC08	927	103	28	3.68
SC08 → SC09	943	298	94	3.17
SC09 → SC10	1008	367	111	3.31
SC10 → SC11	1099	83	28	2.96
SC11 → SC12	1107	436	143	3.04
SC12 → SC13	1216	95	40	2.38

Table 9.4: Experimental results for the semantic lifting of model differences

Interpretation of the results. Concerning quality aspect Q_1, the compression rates show that model differences can in fact be optimized significantly through semantically lifting to user-level edit operations.

Although most of the specified SysML edit operations appear to be quite simple, we measured fairly high compression rates in terms of our test cases. The high compression rates are mainly caused by the fact that the SysML meta-model is far from being optimized for model comparison and leads to a large number of pseudo changes. Most parts of the change set recognition rules are involved in collecting pseudo changes which result from the complex, often redundant structure of the SysML meta-model. Low compression rates occur in the context of edit operations which change attribute values, e.g. the name of a model element, which leads to one low-level change only.

Threats to validity. Concerning the construct validity of our results, it is debatable whether the compression factor is adequate for assessing the understandability of a model difference. However, it is generally assumed that the reduction of edit steps increases the quality of a difference in the sense that it is better understandable for tool users. Thus, the compression factor is a quantitative measure indicating how much the number of edit steps contained by a difference can be reduced by using high-level edit operations. On the contrary, differences are not unique, and there might

be alternative edit steps that describe model changes in a more or less convenient way from a user's point view. In fact, the domain of model versioning still lacks a commonly accepted set of quality measures for model differences [201]. We mitigate this threat by choosing only edit operations which are offered as editing commands by the SysML editor Papyrus for the configuration of our differencing engine used in this experiment. We deliberately omitted complex evolutionary operations which may lead to even higher compression rates but which might be useful only under certain conditions, e.g. in case of project-specific style guides.

9.3.2 Generation and Application of Edit Scripts

We have evaluated our approach with respect to quality aspects Q_2 - Q_4 of generated edit scripts using model histories obtained from an empirical case study on model evolution.

Case study. The test data set being used in our experiment originates from the GMF case study. The study was initially contributed by Herrmannsdörfer et al. [126], and later extended by Langer et al. [157] in order to evaluate some of the contributions of his dissertation (cf. Chapter 10). They extracted the revision history of three Ecore models defined within the GMF project, namely the *Graphical Definition Metamodel* (gmfgraph), the *Generator Metamodel* (gmfgen), and the *Mappings Metamodel* (gmfmap).

They studied the evolution of these models from GMF release 1.0 to release 2.1 covering a period of two years: Firstly, they checked-out each model version between releases 1.0 and 2.1 from the GMF Subversion repository. Secondly, they manually reverse engineered the edit operations that *presumably* have been applied between the revisions. The range of available edit operations includes elementary user-level operations as well as complex refactoring operations known from object-oriented programming. Additionally, model elements were assigned persistent identifiers according to the performed edit steps.

Evaluation setup. The tool functions which are used in this experiment have been adapted to Ecore as shown in Table 9.2. Since all model elements in the GMF histories have universally unique identifiers, a UUID-based matcher has been selected. The edit operation detection engine has been configured with two sets of edit rules implementing the mandatory and optional edit rules. In particular, all of the 32 complex edit operations given in [157] are covered by our complex edit rules. For the the resolution of operation arguments in terms of the non-interactive application of edit scripts, the matching-based argument manager which uses the UUID-based matcher for the resolution of arguments has been selected.

Similar to the evolution of the PPU, each evolution step $v_n \to v_{n+1}$ in the histories of the considered GMF models provides a test case for which an edit script $\Delta_{M_n \Rightarrow M_{n+1}}$ has been extracted (we use M_n here to refer to a model version v_n). Basic properties

of each evolution scenario are shown by column two and three of Table 9.5: *#Low-level* denotes the number of low-level changes in the difference $\delta_{M_n,M_{n+1}}$, the number of operation invocations contained by the extracted edit script $\Delta_{M_n \Rightarrow M_{n+1}}$ is shown by column *#Ops.*, which further distinguishes mandatory and optional complex operations *(#mandatory/#complex)*. In case of the *gmfgen* history, the 108 evolution scenarios are summarized by average values.

Correctness of edit scripts (Q$_2$). In order to assess quality attribute Q_2, each edit script $\Delta_{M_n \Rightarrow M_{n+1}}$ was applied to its original model M_n. The result of the application, called $M_{n'}$, was compared to M_{n+1}, expecting equal models $M_{n'}$ and M_{n+1}.

As shown by the column for test series Q_2 in Table 9.5, the edit scripts which were created in our case study are correct for all evolution scenarios.

Consistency-preservation of edit steps (Q$_3$). The consistency-preservation of edit steps was evaluated by applying each edit script $\Delta_{M_n \Rightarrow M_{n+1}}$ to its original model M_n for all evolution scenarios. After each edit step, the resulting state of the model was checked for consistency violations by applying the EMF validation rules provided by Ecore.

Validation errors are already reported for the initial model versions of the observed histories, i.e. before any edit script was applied. This is not very surprising, since the applied validation rules check a model against the *perfect* level of consistency. Thus, the column for test series Q_3 in Table 9.5 reports the number of changes to the total amount of validation errors which are observed for a model. For example, the initial version of *gmfmap* violates the Ecore constraint that “a class that is an interface must also be abstract” 9 times. All of these invariant violations are corrected from revision 1.49 to 1.50. Similar changes to the total amount of validation errors can be observed for *gmfgraph* and *gmfgen*. We can conclude that none of these changes report an increase of validation errors which results from the application of an edit operation.

Abstraction-level of edit scripts (Q$_4$). Quality attribute Q_4 can be directly measured within the edit scripts. We are interested in the *recall*, i.e. which fraction of the “actually occurred” complex edit operations was detected with our approach. The reference values were given by the manually reverse engineered GMF model histories.

We observed average recall values between 0.61 and 1.00, i.e. some complex edit operation invocations were not found. This is due to the fact that their effect (or precondition) was only visible (or fulfilled) in a transient intermediate state, but not observable in the low-level difference (cf. Section 4.4). Instead, one or several operations which summarize the effect of a complex one are reported.

An example for this can be found in the evolution of the GMF Generator Metamodel (gmfgen). Here, the manually reversed edit sequence reports two relocations of a class attribute; first, it is pulled up to a superclass and subsequently moved to an associated class of this superclass. In the low-level difference, however, we can only observe a simple attribute movement which is finally recognized by the edit script generator.

gmfgraph					
			Q_2	Q_3	Q_4
Test case	#Low-level	#Ops.	Correct	δ(#valid.-errors)	Recall
1.23 → 1.24	9	2/1	✓	0	-
1.24 → 1.25	4	1/0	✓	0	-
1.25 → 1.26	8	1/1	✓	0	-
1.26 → 1.27	10	10/0	✓	0	-
1.27 → 1.28	10	10/0	✓	0	-
1.28 → 1.29	14	14/0	✓	14	-
1.29 → 1.30	202	55/17	✓	3	0.61
1.30 → 1.31	7	4/0	✓	0	-
1.31 → 1.32	6	3/0	✓	0	-
1.32 → 1.33	54	16/2	✓	0	-
gmfmap					
Test case	#Low-level	#Ops.	Correct	δ(#valid.-errors)	Recall
1.43 → 1.44	14	6/0	✓	0	-
1.44 → 1.45	2	2/0	✓	0	-
1.45 → 1.46	107	39/4	✓	0	1.00
1.46 → 1.47	9	3/0	✓	0	-
1.47 → 1.48	1	1/0	✓	0	-
1.48 → 1.49	33	8/4	✓	0	1.00
1.49 → 1.50	9	9/0	✓	9	-
1.50 → 1.51	34	12/0	✓	0	-
1.51 → 1.52	43	13/2	✓	0	1.00
1.52 → 1.53	10	4/0	✓	0	-
1.53 → 1.54	1	1/0	✓	0	-
1.54 → 1.55	35	20/0	✓	0	-
1.55 → 1.56	16	6/1	✓	0	1.00
1.56 → 1.57	1	1/0	✓	0	-
1.57 → 1.58	30	15/0	✓	0	-
avg. values					
	#Low-level	#Ops.	Correct	δ(#valid.-errors)	Recall
gmfgraph	32.40	11.80/1.90	100.00%	1.70	0.61
gmfmap	23.00	9.47/0.60	99.33%	0.60	1.00
gmfgen	24.52	8.75/0.71	99.07%	0.75	0.78

Table 9.5: Experimental results for the generation and application of edit scripts

The example shows that recall values have to be interpreted with caution. In this case, the manually reversed sequence of refactorings provides valuable information for the use case of the original case study [126], i.e. the automation of model migration in response to meta-model adaptation. With respect to model versioning, however, difference tool users most likely prefer the change to be explained as simple attribute movement.

As we have shown in test series Q_2 and Q_3, correctness and consistency-preservation of the extracted edit scripts were not affected by unrecognized complex operations.

Analogously, the precision of our approach can be deduced from the assessments of Q_2 and Q_3: false positives are shown to not occur because otherwise the target model would not have been reconstructed correctly.

Threats to validity. A threat to the internal validity is the method used in test series for Q_2 to check the correctness of edit script applications: The actual result $M_{n'}$ of the application of edit script $\Delta_{M_n \Rightarrow M_{n+1}}$ to its original model M_n needs to be compared with the expected result M_{n+1}. Obviously, the UUIDs of all model elements in $M_{n'}$ which are created by the application of $\Delta_{M_n \Rightarrow M_{n+1}}$ differ from the UUIDs assigned to the corresponding elements in M_{n+1}. Consequently, meaningful UUIDs are not available in $M_{n'}$ by construction. Thus, we used the similarity-based matching engine of the SiDiff model comparison framework [2, 138] to check the equality of $M_{n'}$ and M_{n+1}. Similarity-based matchers can produce correspondences which are generally considered sub-optimal or wrong. However, [259] has analyzed this error for class diagrams and the SiDiff framework; the total number of errors was typically below 2%.

9.3.3 Conflict Detection

We have chosen the Colex [56] benchmark set to evaluate the conflict detection facility of our approach to updating workspace copies of models in collaborative development scenarios. According to the quality aspect Q_5 (s. Section 9.1), the evaluation should show that the accuracy of the conflict detection facility of our approach is fairly equal to traditional 3-way merging. Thus, we compare the results being obtained in our approach with the results being expected in traditional 3-way merging for a set of carefully selected conflict scenarios defined by the Colex benchmark set.

Evaluation setup. Colex is an open, web-based, collaborative conflict lexicon which describes a collection of conflicting situations for UML class diagrams, state machines and sequence diagrams. Each example scenario consists of three versions of a model; the common base version and the two conflicting versions called left model and right model, respectively. Each conflict is classified according to the causal conflict categorization presented in [56], which assumes a traditional 3-way merging. Basically, this categorization distinguishes two main causes for conflicts; *overlapping changes* and *violations* of given specifications, e.g. operation contract violations and violations of consistency constraints defined by a given meta-model. Overlapping changes are further refined into *contradicting changes* (update/delete and update/update) and so called *equivalent changes*.

Some conflicts can only be detected based on specific domain knowledge. For example, if one modeler adds an attribute *name* to a class *Person*, and the attributes *firstName* and *lastName* are added to the same class by a second modeler, it is a matter of opinion whether this situation represents an equivalent change that shall be reported as conflict. In our evaluation, we have deliberately left out 10 scenarios that are based on domain knowledge. The remaining 33 benchmark scenarios were analyzed

for conflicts detected by our approach, assuming the generic warning configuration described in Section 9.2.1. For each of the scenarios, the update was simulated in the workspace of the left model (WS_{left}) and in the workspace of the right model (WS_{right}). Detailed results of this analysis including all edit scripts serving as as patches can be found at the accompanying web site[5] of [8].

	WS_{left}		WS_{right}	
	Exec.	Arg.	Exec.	Arg.
Contradict. (update/delete)	7	1	7	1
Contradict. (update/update)	1	5	2	4
Equivalent change	6	7	7	6
Violation	5	1	6	0
Sum	**19**	**14**	**22**	**11**

Table 9.6: Kinds of conflicts in the COLEX benchmark and their detection by our approach

Comparison to 3-way merging. Table 9.6 summarizes for each kind of conflict of the COLEX benchmark which kinds of warnings or errors are reported by our approach. All conflicts can be detected by our approach, regardless of the workspace in which the update was performed. Most scenarios lead to similar conflict reports as in case of traditional 3-way merging: 88% of the update/delete conflicts are detected as non-executable operations (*Exec.*), mostly due to missing operation arguments. 75% of the update/update conflicts lead to warnings due to modified operation arguments (*Arg.*). 92% of the violations lead to non-executable operations, most of the operations fail due to unfulfilled application conditions. No dominating method of conflict identification can be observed in case of equivalent changes. If equivalent changes lead to the violation of well-formedness rules, e.g. the parallel insertion of equally named elements into the same namespace, the resulting conflict is detected by non-executable operations. The remaining equivalent changes lead to warnings caused by modified operation arguments.

Threats to validity. A threat to the validity of our results concerns the way in which the accuracy of the conflict detection facility is being assessed. While all conflicts of the selected examples can be found using our approach (which indicates a high *recall*), the COLEX benchmark set is generally unsuited to quantitatively evaluate the *precision* of a conflict detection procedure for the following reason: Each of the examples actually leads to a conflicting situation and many of the examples are minimal in the sense that we have to propagate only a single edit step to the workspace version. Thus, false positives cannot occur in these cases. In this respect, experiments with larger models are needed to assess the precision of our conflict detection facility in a meaningful way. However, to the best of our knowledge, COLEX is the only publicly available benchmark which enables a uniform comparison of different approaches to model merging.

[5] http://pi.informatik.uni-siegen.de/Projekte/SiLift/ase2014.php

CHAPTER 10

Related Work

A lot of research into methods and algorithms for comparison and versioning of software models was stimulated recently; the CVSM online bibliography[1] compiles about 400 publications in this field, most of them dating from 2003 or later. While the state of the art has been already outlined in Section 2.3, approaches being closely related to ours will be investigated in more detail in the remainder of this chapter. Generic approaches that can be applied to many modeling languages will be discussed in Section 10.1, while language-specific approaches which cannot be transferred to other modeling (sub-)languages are briefly reviewed in Section 10.2. An overview of currently available model repositories is given in Section 10.3. Section 10.4 closes with a review of approaches from other domains being closely related to a subset of the problems addressed in this thesis.

10.1 Generic Model Versioning

AMOR. The AMOR ("Adaptable Model Versioning") project [26] is a cooperative research project which particularly addresses conflict detection and conflict resolution in the context of 3-way merging. The deficiencies inherent to this functional principle have been already discussed in Section 2.4. Nonetheless, the project has led to a number of publications, some of them are closely related to our work.

Brosch et al. [57, 58, 59] have addressed the specification of composite edit operations. The main goal is to provide an intuitive, yet informal, approach to specify edit

[1] `http://pi.informatik.uni-siegen.de/CVSM`

operations based on the principle of "model transformation by example". Their main contribution is a development tool which is known as Operation Recorder. Within this tool, composite operations are defined "by example". Basically, a tool developer creates a typical model serving as base version and edits the model to produce the effect of the composite operation. Subsequently, the old and new model states are compared, and the Operation Recorder derives change actions as well as pre- and postconditions which serve as a draft of a specification of this composite operation. This draft version of an operation specification is finally corrected and tailored by the tool developer.

Langer et al. [155, 157] use these operation specifications for detecting complex operations in differences which are obtained from EMF Compare, however with different goals and assumptions compared to ours. This approach does not intend to produce executable edit scripts being used as patches. Consequently, the identification of arguments, dependencies between operation invocations etc. are not directly addressed. This is no problem for the goals of their approach, namely to document occurrences of complex edit operations in order to make a difference better understandable. Moreover, composite operation definitions are used for annotating differences and to exploit these annotations to enable better conflict detection within 3-way merging [155]. Among the approaches providing support for high-level differencing, the one presented by Langer is the most similar to ours. Unlike our approach which is based on graph transformation concepts, complex edit operations are specified in a custom transformation language. Thus, formally reasoning about potential conflicts and dependencies or the semantic impact of edit operations as proposed in [20] is not possible with this approach.

Moreover, Langer et al. [155, 158] propose to adopt the notion of signs and signifiers in linguistics in order to detect language-specific conflicts in the context of 3-way merging. Two model elements which are mappings of the same original concept are supposed to have the same signifier. The idea is to analyze the result being obtained by a 3-way merge procedure for further conflicts which are issued as warnings. On the one hand, concurrent signifier changes of the same element indicate a potentially contradicting change of a model element's meaning. On the other hand, an issue arises if two distinct model elements in the revised versions share the same signifier, as this might indicate an overlapping meaning of different elements in the merged model. Although being used in a different context, [158] shares several ideas with our techniques to signature-based and comparison-based detection of modified operation arguments. Firstly, signifiers are similar to our notion of a model element's signature, i.e. a signifier includes local and non-local properties of a model element that convey its superior meaning. Secondly, the per-type specification of signifiers for a given meta-model roughly corresponds to our approach to configure a comparison-based detection of modified arguments. Relevant properties are specified using a slightly extended form of the Epsilon Comparison Language (ECL) [148], a DSL for developing language-specific model matching rules. Therefore, ECL provides a set of built-in compare functions being similar to those of the SiDiff framework. If a rule indicates a match for two model elements, the model elements share the same signifier. To summarize, [158] presents an orthogonal extension of generic model versioning systems; although signifiers are intended to be used in conjunction with a conventional approach to 3-way merge, there is no interaction

with the merge operator. As contradicting signifier changes and unexpected signifier matches are handled a posteriori, the technique can be also used as a complementary step to our workspace update procedure.

EMF Compare. We have also analyzed EMF Compare (EMFC) [63, 86], the currently most widely used differencing engine for EMF-based models, about how it deals with the problems addressed in this thesis. Documentation about these aspects is scarcely available, therefore we performed several tests. EMFC produces pseudo differences in some cases, but not always; there does not seem to be a clear strategy for dealing with them. The change of the association navigability as in our Example 2.1, for instance, produces two reported changes. Our hypothesis is that EMFC filters pseudo changes being caused by the underlying EMF-based model representation. For example creations and deletions of opposite references are reported only once.

Although language-specific edit operations cannot be recognized at all, EMFC exploits the underlying EMF model representation to partially lift model differences to a higher level of abstraction. Deletions and creations of containment references, for instance, are generally reported as move operations. Furthermore, deletions of subtrees of an ASG are reported as single edit step deleting the root node of a subtree; child nodes and all references being incident to deleted nodes are implicitly deleted, too. While this can be useful for the purpose of documenting model changes in a compact manner, implicit deletions are risky when changes shall be propagated to models being different from the original one (cf. scenarios SC3-5 in Section 2.1) because it can cause a considerably different effect which is likely to be unintended.

Nonetheless, the EMFC tooling offers several interesting features. The difference presentation GUI, for instance, enables the grouping and filtering of changes on a per-type (additions, deletions, moves, reference changes, etc.) basis, thus providing a rudimentary implementation of the idea of selectively displaying changes [269]. Moreover, changes between visual models can be illustrated using the external diagram representation. The available presentation technique combines concepts known from the interactive list of edit steps and the unified diagram (s. Section 2.2.3) in a single GUI.

Interactive merging. Most approaches to 3-way merging assume that all merge decisions are taken before the merge result is finally generated non-interactively. Some approaches [87, 161, 220] deviate from this structure by enabling users to manually trigger single edit steps in both asymmetric differences $\Delta_{V_b \to V_1}$ and $\Delta_{V_b \to V_2}$ of Figure 2.9. After each step, the intermediate state of the model can be edited; however, the current conflict analysis is invalidated and must be repeated. The stepwise interactive generation of the merge result is similar to our approach to interactive patching. The most important distinction is the more compact user interface of our approach: The graphical user interfaces of related merge tools consist of at least six main windows and several dialogues. In contrast to this, our interactive patch application tool displays only *one* difference and *one* model (and additional dialogues if needed).

Modeling of differences and patches. Most state-based differencing algorithms use a data structure which implements both a matching between an original model A and its revised version B as well as the changes from A to B. This data structure is often referred to as difference model, the corresponding schema is defined using meta-modeling techniques from modeling frameworks such as EMF. The use of a consistent technology stack for the representation of models and model differences is similar to our approach. However, most difference meta-models, e.g. [63, 72, 209, 235], support only primitive graph operations for the specification of changes. The grouping low-level changes to semantic change sets is proposed only in few approaches [149, 155]. However, these approaches are not sufficient for our purpose of model patching because they do not address dependencies between edit steps and the identification of operation arguments. Our approach to modeling of differences, in particular the representation of edit scripts, is thus a significant extension over previous proposals.

10.2 Language-specific Approaches

As already explained in Section 2.3, language-specific approaches to model versioning can only be applied to models of one specific type. This serious restriction also applies to commercial solutions that have been developed for a particular type of model, the tools *Medini unite*[2] and *SimDiff/SimMerge*[3] targeting differencing and merging of Matlab/Simulink [176] models are examples of this. We will not consider all of these approaches in detail, but review only those language-specific approaches being closely related to one or several contributions of this thesis. In this regard, we identified two directions; language-specific approaches detecting complex edit operations, and semantic differencing. The latter direction is an alternative, yet completely different, approach which aims at a better understanding how two versions of a model differ from each other.

Detection of complex edit operations. Gerth et al. [111, 154] present an approach for detecting complex edit operations in a given low-level difference between two versions of a business process model. Analogously to our approach, correspondences obtained in the first step of a differencing pipeline are used to identify the common model elements in both versions. The detection of edit operations, however, proceeds different than ours: Correspondences are enriched with the technique of so-called Single Entry Single Exit fragments (SESE fragments) [249]. These SESE fragments are utilized to associate certain low-level changes (called "action differences" and "fragment differences" [154]) with a pre-defined edit operation which causes the appropriate effect. SESE fragments and the resulting process model decomposition are beneficial since complex edit operations can be detected even if their effect is only partially visible in the low-level difference (i.e. we have a transient effect), which is not possible in our approach. However, the operation detection can be only applied to models having the

[2] http://www.ikv.de/index.php/en/products/unite
[3] http://www.ensoftcorp.com/simdiff

SESE property, which restricts the approach to business process models, limited forms of activity diagrams and similar types of models.

A few publications, e.g. [110, 252] address the detection of complex edit operations on evolving meta-models in the context of meta-model evolution and model co-evolution. Most notably, Vermolen et al. [252] propose a meta-modeling formalism which is used to precisely specify sets of primitive and complex edit operations on meta-models. The basic idea is similar to ours; each edit operation leads to a well-defined pattern in a difference which is to be detected. To that end, low-level changes in a difference are arranged according to pre-defined normal forms such that pattern instances are finally obvious to see. Dependencies between edit steps are established by analyzing their pre- and postconditions. Similar to [154], the approach is also capable of detecting edit operations leading to transient effects in an editing sequence, at least to a certain extent. The idea is to statically define a set of typical combinations of edit operations leading to transient effects. During operation detection, these combinations are treated as usual. In the obtained edit script, each combination can be finally replaced by the original operations.

Semantic differencing. Maoz et al. [172] present a technique for comparing activity diagrams which does not deliver a complete difference, but finds only single counterexamples, called "diff witnesses" to the proposition that the compared activity diagrams are equivalent. These counterexamples can be used as an additional documentation and for testing purposes. In general, semantic approaches cannot detect syntactic differences and cannot explain which edit steps can be used to implement a desired change, thus they are no basis for patching or 3-way merging. Moreover, semantic differencing requires the modeling language to have formal semantics, which is often not the case, notably with informal models used in requirements engineering. Refactorings, which provide valuable information about the evolution of a model, cannot be detected at all because they do not change a model's representation the semantic domain.

Basically the same problems apply to other semantic differencing approaches that have been proposed for class diagrams [174], feature models [102], and certain behavior models [156]. To summarize, semantic approaches are not intended to replace but rather to complement syntactic differencing techniques, their integration is an interesting field for future research [173].

10.3 Model Repositories

Most of the approaches to model versioning considered so far assume models to be stored in traditional, file-based versioning systems and typically re-implement only some basic services such as differencing and merging. On the contrary, some approaches have proposed to develop complete versioning systems which are based on alternative storage technologies from scratch. De Lucia et al. [80] have developed a management system called ADAMS which uses dependency links which support traceability in context-aware change management. The system is extensible in the sense that it can be customized

to different kinds of software artifacts. A specific extension named COMOVER [38, 81] has been developed to parse and handle models being serialized in the XMI format. Another approach which is based on the XMI standard is Odyssey-VCS [183, 199]. Its main feature is to enable a fine-grained definition of which types of model elements are to be treated as atomic configuration items from a software configuration management point of view. EMFStore [145], which is based on the former Sysiphus [62] project, provides a model repository for EMF-based models. CoObRA [218, 219] has been integrated into the Fujaba tool suite[4]. Oda and Saeki [197] propose to use generative techniques in order to develop version control systems for various types of visual models. General concepts for model repositories have also been developed in the AMOR (Advanced Model Repository) project[5], a collaborative project with partners from academia and industry. Commercial modeling tools such MagicDraw[6] often provide a central repository service as additional add-on[7].

Model repositories are rather complex distributed systems and often rely on client-side installations being tightly integrated into an MDE environment. The dominant approach to model differencing is operation-based, which is either achieved by tracking editing commands in standard editors of an MDE environment, or by exploiting the logging facilities of the underlying ooDBMS. Thus, the problem of semantically lifting low-level differences and creating executable edit scripts as addressed in this thesis virtually disappears. However, in addition to general problems of logging-based approaches (cf. Section 2.3.2), the obtained logs are just sequences of edit steps, i.e. "pure" directed deltas; information about dependencies is not available and may be difficult to detect since a precise specification of edit operations is often hidden in the source code of model editors.

Nonetheless, model repositories also provide some advantages. Most notably, they enable fine-grained traceability and locking of dedicated model elements. The latter feature makes pessimistic versioning slightly more attractive. Moreover, the unit of versioning can be shifted from files, which often include a complete model, to particular sub-models or even to single model elements. These features are largely orthogonal to the tool functions being developed in terms of this thesis, there is no principal limitation to integrate these functions with any of the available model repositories. However, the basic storage service of these systems is specifically designed for models and does not support the versioning of source code or other kinds of traditional development documents, which are usually not replaced completely. Using different versioning systems for models and traditional development documents is obviously not an attractive solution.

[4] http://www.fujaba.de
[5] http://www.model-repository.de
[6] http://www.nomagic.com/products/magicdraw.html
[7] http://www.nomagic.com/products/teamwork-server.html

10.4 Approaches from other Domains

In this section, we briefly review approaches from other domains being closely related to the some of the problems addressed in this thesis.

Design decision management. Könemann [149, 150, 151] presents an approach to the patching of models which considers the problem from a broader perspective. This approach aims at understanding and documenting the goals and design decisions manifested in the changes observed between the original and changed model. Könemann proposes a detailed process how a raw difference is transformed into a patch. This process combines several simple heuristics for grouping model changes and how the patch is applied later on a target model. The process resorts several times to interactive interventions of developers, e.g. in order to semantically enrich the contents of the patch, to control the resolution of references, or to correct and to control the effects of the application of the patch. On the one hand, the process is very flexible, in extreme cases, the patch is actually re-implemented for the target model. On the other hand, it is very labor-intensive since each group of related elementary modifications must be identified and treated manually.

In contrast to Könemann's approach, our approach is much more automated: the lifting of raw differences is fully automated, the application of the patch requires only limited user interaction. Our approach can exploit the additional knowledge about complex edit operations in order to detect arguments in editing steps and to protect against consistency violations.

Mining software repositories. A number of publications have addressed the reverse engineering of refactorings in the context of mining repositories.

Fadhel et al. [43] propose a search-based detection of model refactorings. Given an initial model, a revised model and a set of executable edit operations, they search for a sequence of operation invocations that transforms the initial model into the revised model. Due to the huge number of possible combinations of edit operation invocations, a heuristic method is used to explore the space of possible solutions; The approach uses a genetic algorithm that has been adapted to the problem domain: An initial population of candidate solutions, i.e. edit operation sequences, is iteratively evolved using simple implementations of the genetic selection, mutation and crossover operator. The fitness of a solution is assessed by its similarity to the revised model provided as input. An advantage of the approach is that it is able to detect sequences of complex operations which have transient effects. The algorithm seems to be practically applicable if original and revised model are very similar. In case of larger-scaled differences, however, the approach most likely is faced with serious performance problems. Moreover, as in the case of logging, the algorithm delivers pure directed deltas without any information about actual dependencies between edit steps.

Xing et al. [265] proposes to export relevant difference information in a database and to use queries for finding instances of refactorings. Our problem of actually annotating and lifting a difference is not directly addressed, the production of information

required to later re-execute found instances of refactorings is completely out of scope. Furthermore, the overhead of setting up a database and creating all required indexes, which is relevant when mining repositories, is a significant performance problem when comparing only two models.

Further analysis procedures, e.g. [82, 83, 205], have been proposed for detecting refactoring operations in evolving object-oriented software. The main intention is the same as with [43, 265], namely to document potential instances of refactorings, which shall be manually inspected or just counted.

Generation of edit operations. We are not aware of any other approaches for constructing consistency-preserving edit operations. There are approaches to create certain kinds of edit operations or grammars which can construct or modify graphs or models. However, these approaches either do not support all kinds of modifications (e.g. *setting attribute values* or *moving elements*) or they can lead to serious consistency violations.

Edit operations on models are indirectly addressed in some approaches which aim at generating instance models for a given meta-model. Virtually all of these approaches are based on the idea to systematically enumerate meta-model instances. Brottier et al. [61] describe an enumeration algorithm which is based on model fragments that must be specified manually. Other approaches use SAT-solvers such as the Alloy Analyzer [129] to systematically enumerate valid instances in a restricted search space. The approach of Alanen and Porres [23] assumes that a model is first converted into a string representation, then edited using a syntax-directed editor, and finally converted back to an ASG-based representation. Our actual problem of generating executable specifications/implementations of edit operations has been addressed only by few approaches.

In the context of engineering delta modeling languages [119], Seidl et al. [221] present an approach to generate executable edit operations (called "delta operations") from EMOF-based meta-models. They support six kinds of delta operations which are similar to ours. Set and unset operations are included as they support model element attributes having `null` values. However, in contrast to our CPERs, multiplicity constraints in meta-models are not supported, i.e. mandatory children and neighbors are not created (deleted) in a single transaction.

Ehrig et al. [96] deduce graph grammar rules from meta-models. The generated set of rules is organized in three layers: Layer 1 rules create instances of meta-model classes, Layer 2 establishes mandatory relationships between elements. In this step additional elements are also created when necessary. Finally, Layer 3 rules establish optional relationships. The rules are applied randomly to an empty initial model to create a concrete instance of the meta-model. Taentzer [234] extends the approach from restricted multiplicities to arbitrary ones. However, using the concept of layered graph grammars obviously leads to inconsistent intermediate states because the different layers, in particular layers 1 and 2, are applied independently. Hence, the generated rules do not implement consistency-preserving edit operations. Moreover, rules which delete or move model elements or which change attribute values of elements are not generated

in this approach at all.

Hoffmann and Minas [127] describe how to translate a class diagram into a so called adaptive star grammar. They also support advanced modeling concepts of class diagrams, like association subsetting and redefinition. However, such grammars are only intended to produce new models. They do not address the problem of modifying existing models, e.g. using deletions, since their rules use non-terminal symbols which do not appear in existing models.

CHAPTER 11

Conclusions and Future Work

This chapter concludes the thesis with a summary in Section 11.1, an outlook on possible future work and research directions is given in Section 11.2.

11.1 Summary

Version management is a key functionality of software development environments. This applies in particular to software development environments which support MDE. While basic storage services such as storing models in files, repositories or databases are readily available, it soon became obvious that functions for comparing (differencing) and merging models are a much bigger challenge than similar, well-known functions for textual documents, notably source code. It is commonly agreed that model versioning needs a structural approach to model differencing and the propagation of model changes. Line-based approaches as used for textual software artifacts are inappropriate. To that end, approaches to structural model versioning have started to be developed, however, are still in their infancy. The main problem is that they are stuck in abstract syntax structures for models and operate on low-level, sometimes tool-specific model representations. The resulting differences, which are based on low-level edit operations such as creating/deleting single nodes/edges of an ASG, are often hard to understand and lead to serious consistency problems when being used for the propagation of model changes.

To tackle these problems, we presented concepts, techniques, and tools that are needed to systematically lift up structural model versioning to a higher level of abstraction. The key idea is to lift model differences to user-level edit operations, e.g.

operations available as editing commands in standard model editors or as offered by modern refactoring tools, such that modelers can easily recognize and manage changes. Obviously, formalisms to define such language-specific operations and methods to recognize the resulting changes mutually depend on each other. We use model transformation rules implemented in Henshin in order to specify edit operations in a precise and meaningful way. Our approach exploits the fact that Henshin rules are based on graph transformation concepts. These rules are declarative enough in order to be statically analyzed, e.g. for potential conflicts and dependencies, and to be dynamically interpreted by generic algorithms. In particular, we use the Henshin pattern matching engine in order to find occurrences of edit operations in a given low-level difference; change patterns to be found are specified by recognition rules which are automatically generated from their corresponding edit rules. Of course, edit rules can be used as executable specifications of the respective edit operations, i.e. edit scripts generated by our approach serve as executable model transformations. This way, the same set of edit rules can be used to specify and recognize model changes. Consequently, detectable edit operations being reported as model changes are always kept consistent with their executable specification.

Based on the developed foundations and technologies, another contribution of this thesis is an approach to model patching which prevents serious inconsistencies in the patched model: Patched models are guaranteed to meet the consistency constraints required by the standard editor of a development environment. The key idea is to use only consistency-preserving edit operations on models, which obviously depend on the modeling language and the required level of consistency. Engineering a set of executable specifications of CPEOs is supported both technically and methodologically. The idea is to first "transform" a perfect (or standard) meta-model into an effective meta-model which formally documents the required consistency-level of a particular modeling environment. Edit rules being implementations of CPEOs are finally obtained from the effective meta-model in a semi-automated way. Our approach to edit rule generation supports basic consistency-constraints and the commonly used combinations of multiplicity invariants.

In addition, we slightly extended our model patching facility to provide a completely new approach to update workspace copies of models. This approach is based on the principle of document patching and differs methodologically and technically from previous approaches based on 3-way merging. An interactive workspace update tool which is based on our approach fulfills the same requirements than classical merge tools, some of them in a better way: From a modeler's point of view, the user interface is less complex than GUIs of interactive 3-way merge tools, and the user interaction concept does not force a user to mentally return to the common base version. Nonetheless, we were able to detect almost the same conflicts, which is demonstrated by means of a model merging benchmark. From the point of view of a tool chain maintainer, a workspace update tool can be integrated into an existing MDE tool environment with minimal effort.

We finally presented a generic framework supporting the planned re-use and composition of a set of configurable tool components. A subset of these components has been

developed in terms of this thesis, while other components can be integrated and re-used from an existing MDE environment. The framework is based on a thorough domain analysis which conveys several requirements that arised in our industrial and academic collaborations. We demonstrated the feasibility of the proposed approach providing a reference implementation known as SiLift. The framework has been used to build several individual instances of a family of difference tools covering a broad range of different version and variant management scenarios.

The concepts and tools developed in this thesis have been evaluated in several case studies. In addition to our own experiments, the approach has been successfully used to solve concrete problems in other research projects dealing with model evolution, e.g. in the context of the German priority programme on managed software evolution [116]. Moreover, a brief overview of selected synergetic research results that have been collaboratively achieved in the context of software evolution for automated production systems is presented in [19]. Semantically lifting model differences to a higher level of abstraction plays a pivotal role for many of these results.

In this thesis, we emphasized the support of visual modeling languages since they impose the most challenging problems and requirements w.r.t. model versioning. However, we also showed that the approach is not limited to visual models but can be applied to textual models, too. This line of research is not only of great interest for the domain of model versioning but can be also exploited for change management of textual artifacts, notably source code used in classical programming languages. We are convinced that lifting low-level differences to a higher level of abstraction based on semantically rich edit operations can be advantageously used for the versioning of all kinds of structured documents.

11.2 Outlook

In this section, we briefly outline possible directions to future research, which can be roughly classified into two categories:

On the one hand, our approach can be improved w.r.t. several aspects. Handling of ordered element sets, optimizing the performance of the difference calculation, and the integration of our tool functionality into a version control system are examples of this. Another important aspect is to provide better meta-tool support in order to make the configuration of individual tools which are based on our framework more user-friendly.

On the other hand, tool functions for calculating and propagating model changes are very basic functions which are necessary in many application scenarios beyond classic versioning. Many of them can benefit from lifting these functions from low-level to high-level edit operations. In the context of analyzing model histories, for example, complex edit operations can be utilized as they implicitly create tracing information, i.e. information about successor relationships between model elements. Model co-evolution and delta-oriented SPL engineering are further example problem domains to which our techniques can be beneficially applied.

Handling of ordered element sets. Currently, we assume ASGs to be typed, attributed but not to be ordered graphs, which is consistent with the graph model of Henshin. However, models may contain various "collections" of ordered model elements. In some cases, the order is relevant for model versioning. For example, a parameter list of an operation of a UML class is an ordered set of parameters. Users can re-order the elements of a collection and insert or delete elements at a given position. Such editing effects should be reported correctly in model differences. If differences are transformed to edit scripts which are used as patches, elements should be arranged according to the specified order, i.e. positions must be specified in edit scripts. These requirements finally lead to the problems (i) to offer an approach for handling positions of inserted, deleted or moved elements and to define appropriate consistency-preserving edit operations on ordered sets, and (ii) to actually compute the permutation of a collection.

In [134], we proposed a first approach how to deal with these problems. As we are typically interested in the relative order in which elements are arranged to each other, the basic idea is to introduce additional edges in an ASG indicating the relative position of an element with respect to its predecessor/successor. Thus, re-orderings as well as insertions and deletions at dedicated positions can be implemented as normal Henshin rules. On the meta-model layer, MOF-based meta-models, in which ordered edge types are declared by simple data values, are mapped to type graphs such that linked list "implementations" are provided for each ordered edge type. Such a "refined" type graph is finally implemented in EMF Ecore. As a consequence, difference tools do not operate on instances of the original meta-model, but on a refined version. Thus, on the instance model layer, two additional transformations are required: An input transformation supplements predecessor and successor relationships. If the processed models are being modified, e.g. in case of patching scenarios, an output transformation arranges the order of runtime objects in terms of their original implementation structures according to the order which is induced by the predecessor/successor relationships. The predecessor/successor relationships are finally deleted.

However, we left an integration of the proposed approach into our model differencing tool chain as future work, as it is faced with several challenges which have yet been addressed only in an ad-hoc manner.

Tooling-related challenges are (a) how to synchronize instances of the refined meta-model with instances of the original meta-model while interactively applying edit scripts, and (b) how to simultaneously manage an original meta-model and its refined version in a single MDE environment.

A particular conceptual problem is that edit operations on ordered element sets can lead to unavoidable transient effects. In this regard, [134] suggests only a simple workaround mitigating this problem. The basic idea is to extend our differencing pipeline by a feedback loop such that the information gathered during edit operation recognition is taken into account by the creation of a low-level difference. In particular, correspondences between elements in corresponding element lists shall be iteratively removed. The iteration ends as soon as all low-level changes can be grouped to semantic change sets, i.e. all non-erasable transient effects have been eliminated. The feedback loop terminates as each iteration only leads to the removal of a correspondence, no

incremental calculation of a "better" matching is being triggered. In the worst case, none of the elements of the corresponding lists are considered to be corresponding. In this case, we can be sure to find a trivial sequence of edit steps, i.e. all elements are removed from the list and subsequently inserted in the correct order.

In general, the conceptual design and implementation of matching algorithms that allow a feedback loop in the conventional processing pipeline of state-based difference tools is worth to be investigated in more detail for several other problems. An example of this is the problem of how to handle model matchings violating soundness criteria being required by our approach to operation detection. These cases are hypothetical if a reliable matcher is used, but they may occur if the matching has to be calculated based on heuristics.

Performance optimizations. Experiences collected in the conducted case studies and experiments show that the performance of our approach to semantic lifting depends on a variety of factors, which is a general finding for rule-based model transformations [244]. Runtimes are acceptable for most configurations and medium-sized models. However, if models or complex edit rules are huge, difference calculation leads to runtimes which are hardly acceptable in typical usage scenarios. The same problem arises when our techniques shall be applied to structural differencing of program source code. Our hypothesis is that the following conceptual improvements have the potential for significant performance optimizations:

- **Incremental lifting of differences**: Optional complex edit rules are typically composed of smaller mandatory or other optional rules. Nonetheless, we currently treat such a complex rule as a single monolithic rule, which can lead to considerably large edit rules for some kinds of edit operations, notably in the case of refactorings. The generated recognition rules are treated as one monolithic rule, too. Obviously, a recognition rule is even slightly bigger than its corresponding edit rule as it comprises difference information *and* model structures (s. Chapter 4). The approach is sensible from the perspective of software design; during operation recognition, all rules can be processed in the same way, and all recognition rules can be applied to a low-level difference in parallel. From a performance point of view, however, incrementally lifting a difference is a less costly approach. The idea is to apply small rules first. Subsequently, composite rules are applied. Semantic change sets which have been already created in a previous iteration may serve as pre-match for a complex recognition rule. Thus, the depth of the recursion of the recursive graph pattern matching algorithm that is used to find a recognition rule match can be reduced in many cases.

- **Reduction of the scope of comparison**: Currently, the scope of comparison covers complete models which are to be compared with each other. This leads to unnecessarily large search spaces for models in which only small, typically local parts were actually changed. However, the reduction of the scope of comparison is not straightforward and leads to several non-trivial requirements. Simple

heuristics as proposed in [30], which decomposes the source code of programs on a "per-file basis" into a set of disjoint context-free syntax trees, are hardly applicable to ASGs of models. More sophisticated strategies are required in order to split large models into smaller, manageable parts. A particular challenge is that the splitting strategy must be compatible to the configuration of a difference calculation tool.

Integration into a version control system. The tool functions being developed in the context of this thesis are independent of a particular version control system (VCS). This is strongly required for some of our application scenarios, notably SC4, where different variants of a model are not located in a central repository due to organizational constraints. In other scenarios, in particular SC3 and SC5, our solutions become more powerful from a practical point of view if they are tightly integrated into an existing VCS. Such an integration is possible with a model repository (cf. Section 10.3) or any of the available mainstream versioning systems such as Subversion or Git. We aim at the integration with one of the latter ones such that all MDE and non-MDE artifacts are managed in a single repository.

A technical challenge is that basic commands, such as `SVN update`, are traditionally treated as batch procedures (cf. Section 2.3.3), while a particular feature of our approach is to interactively control the propagation of model changes. One option is to implement a client-side VCS adaptation layer which basically delegates to native commands, but resorts to locally installed difference tools if needed. Another option is to use hooks which are usually provided by complementary client VCS software. This option is very attractive if the VCS client is integrated with a development environment as, e.g., Subversive[1] or EGit[2] which are based on Eclipse. The basic idea is to listen for certain events in order to get noticed before a particular command is to be executed. Then, the execution of the command can be suppressed and passed to a dedicated difference tool instead.

Meta-tool support. Chapter 7 presents a general method and process for the creation of edit rules. However, this process resorts several times to development tasks which have to be performed manually and typically require a lot of experience. To that end, additional meta-tools are required in order to make the configuration of individual tools which are based on our framework more user-friendly

Firstly, a given standard meta-model currently has to be manually edited towards the effective meta-model serving as input for the edit rule generation. This step should be rather performed based on in-place model transformations which are permitted in this context.

A further important aspect is to statically check a set of edit rules for certain criteria, notably soundness and completeness, which can yet only be guaranteed for generated

[1]http://eclipse.org/subversive
[2]http://eclipse.org/egit

rules. However, manual adaptations to generated edit rules have to be carefully engineered. This approach is often acceptable, but tedious and prone to errors in cases where edit rules have to be manually created to a large extent or already exist. Another problem is that the set of edit rules used to configure the operation detection engine must be consistent with the properties of the matching produced by the matcher of a particular differencing pipeline. This problem is not directly addressed in this thesis. Instead, we rather rely on certain properties of the matcher. These assumptions are reasonable, but can not be guaranteed in general, notably in the case of similarity-based matchers. Here, consistent configurations of all components of a differencing pipeline are strongly required.

Finally, meta-tool support shall be provided to create and manage complex edit rules. Such a tool suite should at least offer a basic function to compose two (potentially dependent) transformation rules to a single composite rule having the same effect as the sequential application of its components. In this regard, the automated deduction of composite preconditions from component preconditions is another interesting feature. Here, we think of integrating recent advances in the field of model transformation composition, e.g. [36, 123, 250]. Managing variability in large sets of similar complex edit rules is another issue which needs to be addressed, the approach presented in [230] may be useful for this.

Analysis of model histories supporting complex edit operations. The evolution of a model cannot be fully understood if only information about "short-term changes", i.e. changes between successive revisions, is available. Typical questions which refer to a history as a whole are: When and in which context was a model element created? Which model elements were changed most frequently? Which groups of model elements were modified together? Answering these questions requires to identify long-living entities, i.e. single model elements, or groups of related model elements, which exist almost unchanged for some time in the history of a model. Previous approaches to the analysis of model histories such as [259] support only single model elements as long-living entities. The tracing of sets of related model elements, e.g. all model elements involved in a bug, is only supported in the form of ad-hoc queries. Complex edit operations, however, affect virtually always several model elements. They implicitly create groups of model elements which are affected together by one change and which should also be regarded as long-living entities.

Here, we aim at extending existing concepts by support for non-atomic long-living entities in histories. The basic idea is to utilize complex operations to investigate splits and merges of evolving model elements. For example, model elements which are deleted by a refactoring can "re-incarnate" in other created model elements. The refactoring pullUpAttribute in class diagrams, for instance, creates a new attribute at the superclass; it can be considered as the successor of all deleted attributes at the subclasses. More generally, refactorings and other kinds of complex edit operations can lead to non-trivial structures of successor relationships between model elements which should be exploited for the analysis of model histories.

Supporting model co-evolution. So far, our work focuses on "monolithic" models only. However, MDE typically involves a multitude of different models being used to describe different aspects such as structure, behavior, performance, reliability etc. of a system. Consequently, we have not only one model of a system but a network of interrelated (or coupled) models, which leads to several challenging problems.

Loosely coupled models lead to a synchronization problem: Different (sub-)models, each of them representing a dedicated view, are edited independently of each other since they are assigned to different developers or due to the fact that a developer concentrates on a single aspect at a specific point of time. Thus, changes in one model must be propagated to all related models in order to keep the views synchronized and to avoid inconsistencies. Solutions exist if the propagation is straightforward and can be fully automated (see e.g. [114, 124]). However, there are several types of semantically interrelated models for which no simple and straightforward co-evolution exists [112]. The only viable solution is to pre-define co-evolution rules which can be offered to developers as possible options. A first step towards this direction is presented in [18]. In this work, we address a particular challenge domain engineers are faced with in this context, namely to find the proper co-evolution rules for semantically interrelated models. To that end, we developed an extensive analysis framework to learn about co-evolution steps from a given co-evolution history and to finally predict them with a certain degree of probability. While basic assessments show that the approach is feasible, larger case studies are needed to evaluate how far we can push the generation of co-evolution rules and how much training data is needed to derive appropriate co-evolution rules.

In contrast, versioning tools for *tightly coupled models* are typically faced with the huge challenge of versioning single models independently of each other. In the context of the OMG, a standard modeling guideline for the definition of OMG-related metamodels such as the UML [190], SysML [191] etc. is to represent all information in a model redundancy-free and only once. Redundancy-freedom implies that if several models contain the same information from a users' point of view then the related data is contained in only one model and this unique representation is referenced from other models. For example, a state machine can be associated with as class in a class diagram; a transition in the state machine can be triggered by a call-event which refers to an operation of this class. This leads to the problem that our above state machine cannot be displayed graphically in isolation without the other referenced model, in our example the class diagram: The signature of the called operation is missing. From a users' point of view, this signature is a part of the state machine since it appears as part of the standard graphical representation of the state machine. This example shows that the notion of "a model" as defined in [190] is not identical to the notion of "a model" from a user's point of view. Hence, it does not support models as autonomous documents: One cannot check out an isolated model as a workspace copy and then display and edit this model using standard model editors. This challenge is insufficiently addressed by the current state of the art.

Integration with delta-oriented SPL engineering. Synergies can be achieved by integrating our techniques with delta-oriented SPL engineering [130, 214]. In this approach, there is a distinguished product (called base version) of a product line, which is a fully functional product, e.g. one with a large set of features which were tested together. All other products are generated by patching the base version, i.e. by removing, replacing or adding parts related to a particular feature. In delta-oriented modeling [120], a delta is a specification of how to transform one valid variant (called the source variant) of a model into a target variant. However, the manual specification of a large set of deltas is tedious and prone to errors. Thus, a first obvious usage scenario for applying our techniques is an approach to semi-automated delta extraction [19]. With this approach, the definition of a delta is achieved in two steps: First, a particular source variant is modified using standard editors such that it finally becomes the desired target variant. Secondly, a delta is automatically extracted, basically by comparing the original and the revised versions of the model. Thus, the definition of a delta becomes much easier and far more reliable.

Bibliography

[21] Mathieu Acher, Patrick Heymans, Philippe Collet, Clément Quinton, Philippe Lahire, and Philippe Merle. "Feature model differences." In: *Advanced Information Systems Engineering*. Springer. 2012, pp. 629–645.

[22] Marcus Alanen and Ivan Porres. *Difference and union of models*. Springer, 2003.

[23] Marcus Alanen and Ivan Porres. *A relation between context-free grammars and meta object facility metamodels*. Technical Report No. 606, Turku Centre for Computer Science, 2004.

[24] Marcus Alanen and Ivan Porres. *Subset and union properties in modeling languages*. Technical Report No. 731, Turku Centre for Computer Science, 2005.

[25] Kerstin Altmanninger, Petra Brosch, Gerti Kappel, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. "Why model versioning research is needed!? an experience report." In: *Proceedings of the MoDSE-MCCM 2009 Workshop@ MoDELS*. Vol. 9. 2009.

[26] Kerstin Altmanninger, Gerti Kappel, Angelika Kusel, Werner Retschitzegger, Martina Seidl, Wieland Schwinger, and Manuel Wimmer. "AMOR–towards adaptable model versioning." In: *1st International Workshop on Model Co-Evolution and Consistency Management, in conjunction with MODELS*. Vol. 8. 2008, pp. 4–50.

[27] Kerstin Altmanninger, Wieland Schwinger, and Gabriele Kotsis. "Semantics for accurate conflict detection in SMoVer: Specification, detection and presentation by example." In: *International Journal of Enterprise Information Systems (IJEIS)* 6.1 (2010), pp. 68–84.

[28] Kerstin Altmanninger, Martina Seidl, and Manuel Wimmer. "A survey on model versioning approaches." In: *International Journal of Web Information Systems* 5.3 (2009), pp. 271–304.

[29] Carsten Amelunxen, Alexander Königs, Tobias Rötschke, and Andy Schürr. "MOFLON: A standard-compliant metamodeling framework with graph transformations." In: *Model Driven Architecture–Foundations and Applications*. Springer. 2006, pp. 361–375.

[30] Sven Apel, Olaf Leßenich, and Christian Lengauer. "Structured merge with auto-tuning: balancing precision and performance." In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering.* ACM. 2012, pp. 120–129.

[31] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. "Semistructured merge: rethinking merge in revision control systems." In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.* ACM. 2011, pp. 190–200.

[32] Thorsten Arendt and Gabriele Taentzer. "A tool environment for quality assurance based on the Eclipse Modeling Framework." In: *Automated Software Engineering* 20.2 (2013), pp. 141–184.

[33] Thorsten Arendt. "Quality Assurance of Software Models-A Structured Quality Assurance Process Supported by a Flexible Tool Environment in the Eclipse Modeling Project." PhD thesis. Philipps-Universität Marburg, 2014.

[34] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. "Henshin: advanced concepts and tools for in-place EMF model transformations." In: *Model Driven Engineering Languages and Systems.* Springer, 2010, pp. 121–135.

[35] Thorsten Arendt, Florian Mantz, Lars Schneider, and Gabriele Taentzer. "Model refactoring in Eclipse by LTK, EWL, and EMF refactor: a case study." In: *Model-Driven Software Evolution, Workshop Models and Evolution.* 2009.

[36] Thorsten Arendt and Gabriele Taentzer. *Composite refactorings for EMF Models.* Technical report, Philipps-Universität Marburg, FB 12 - Mathematik und Informatik, 2012.

[37] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1999.

[38] Ivo Barone, Andrea De Lucia, Fausto Fasano, Esterino Rullo, Giuseppe Scanniello, and Genoveffa Tortora. "COMOVER: Concurrent model versioning." In: *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on.* IEEE. 2008, pp. 462–463.

[39] Victor R. Basili and Albert J. Turner. "Iterative enhancement: A practical technique for software development." In: *Software Engineering, IEEE Transactions on* 4 (1975), pp. 390–396.

[40] Lars Bendix and Pär Emanuelsson. "Diff and merge support for model based development." In: *Proceedings of the 2008 international workshop on Comparison and versioning of software models.* ACM. 2008, pp. 31–34.

[41] Lars Bendix and Pär Emanuelsson. "Collaborative work with software models–industrial experience and requirements." In: *Model-Based Systems Engineering, 2009. MBSE'09. International Conference on.* IEEE. 2009, pp. 60–68.

[42] Lars Bendix and Pär Emanuelsson. "Requirements for practical model merge–an industrial perspective." In: *Model Driven Engineering Languages and Systems.* Springer, 2009, pp. 167–180.

[43] Ameni ben Fadhel, Marouane Kessentini, Philip Langer, and Manuel Wimmer. "Search-based detection of high-level model changes." In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on.* IEEE. 2012, pp. 212–221.

[44] Keith H Bennett and Václav T Rajlich. "Software maintenance and evolution: a roadmap." In: *Proceedings of the Conference on the Future of Software Engineering.* ACM. 2000, pp. 73–87.

[45] Alexander Bergmayr and Manuel Wimmer. "Generating Metamodels from Grammars by Chaining Translational and By-Example Techniques." In: *MDEBE@ MoDELS.* 2013, pp. 22–31.

[46] Lasse Bergroth, Harri Hakonen, and Timo Raita. "A survey of longest common subsequence algorithms." In: *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on.* IEEE. 2000, pp. 39–48.

[47] Jean Bézivin and Olivier Gerbé. "Towards a precise definition of the OMG/MDA framework." In: *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on.* IEEE. 2001, pp. 273–280.

[48] Jean Bézivin. "On the unification power of models." In: *Software & Systems Modeling* 4.2 (2005), pp. 171–188.

[49] Enrico Biermann, Karsten Ehrig, Christian Köhler, Günter Kuhns, Gabriele Taentzer, and Eduard Weiss. "EMF model refactoring based on graph transformation concepts." In: *Electronic Communications of the EASST* 3 (2007).

[50] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. "Lifting parallel graph transformation concepts to model transformation based on the Eclipse modeling framework." In: *Electronic Communications of the EASST* 26 (2010).

[51] Enrico Biermann, Claudia Ermel, and Gabriele Taentzer. "Formal foundation of consistent EMF model transformations by algebraic graph transformation." In: *Software & Systems Modeling* 11.2 (2012), pp. 227–250.

[52] Daniel Bildhauer. "On the relationships between subsetting, redefinition and association specialization." In: *Ninth Conference on Databases and Information Systems.* 2010.

[53] Kristopher Born, Thorsten Arendt, Florian Heß, and Gabriele Taentzer. "Analyzing Conflicts and Dependencies of Rule-Based Transformations in Henshin." In: *Fundamental Approaches to Software Engineering.* Springer, 2015.

[54] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. "Model-driven software engineering in practice." In: *Synthesis Lectures on Software Engineering* 1.1 (2012), pp. 1–182.

[55] P. Brosch, H. Kargl, P. Langer, M. Seidl, K. Wieland, M. Wimmer, and G. Kappel. "Representation and Visualization of Merge Conflicts with UML Profiles." In: *Proceedings of the International Workshop on Models and Evolution (ME 2010)* (2010).

[56] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, and Manuel Wimmer. "Colex: a web-based collaborative conflict lexicon." In: *Proceedings of the 1st International Workshop on Model Comparison in Practice.* ACM. 2010, pp. 42–49.

[57] Petra Brosch, Philip Langer, Martina Seidl, Konrad Wieland, Manuel Wimmer, Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger. "An example is worth a thousand words: Composite operation modeling by-example." In: *Model Driven Engineering Languages and Systems*. Springer, 2009, pp. 271–285.

[58] Petra Brosch, Philip Langer, Martina Seidl, and Manuel Wimmer. "Towards end-user adaptable model versioning: The by-example operation recorder." In: *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*. IEEE Computer Society. 2009, pp. 55–60.

[59] Petra Brosch, Martina Seidl, Konrad Wieland, Manuel Wimmer, and Philip Langer. "The operation recorder: specifying model refactorings by-example." In: *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM. 2009, pp. 791–792.

[60] Petra Brosch. "Conflict Resolution in Model Versioning." PhD thesis. Vienna University of Technology, Institute of Software Technology and Interactive Systems, 2012.

[61] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. "Metamodel-based test generation for model transformations: an algorithm and a tool." In: *Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on*. IEEE. 2006, pp. 85–94.

[62] Bernd Bruegge, Allen H Dutoit, and Timo Wolf. "Sysiphus: Enabling informal collaboration in global software development." In: *Global Software Engineering, 2006. ICGSE'06. International Conference on*. IEEE. 2006, pp. 139–148.

[63] Cédric Brun and Alfonso Pierantonio. "Model differences in the eclipse modeling framework." In: *UPGRADE, The European Journal for the Informatics Professional* 9.2 (2008), pp. 29–34.

[64] Greg Brunet, Marsha Chechik, Steve Easterbrook, Shiva Nejati, Nan Niu, and Mehrdad Sabetzadeh. "A manifesto for model merging." In: *Proceedings of the 2006 international workshop on Global integrated model management*. ACM. 2006, pp. 5–12.

[65] Thomas Buchmann, Alexander Dotor, and Bernhard Westfechtel. "MOD2-SCM: A model-driven product line for software configuration management systems." In: *Information and Software Technology* 55.3 (2013), pp. 630–650.

[66] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. "Towards a taxonomy of software change." In: *Journal of Software Maintenance and Evolution: Research and Practice* 17.5 (2005), pp. 309–332.

[67] Jim Buffenbarger. "Syntactic software merging." In: *Software Configuration Management*. Springer, 1995, pp. 153–172.

[68] Horst Bunke. "Error-tolerant graph matching: a formal framework and algorithms." In: *Advances in Pattern Recognition*. Springer, 1998, pp. 1–14.

[69] Sudarshan S Chawathe and Hector Garcia-Molina. "Meaningful change detection in structured data." In: *ACM SIGMOD Record*. Vol. 26. 2. ACM. 1997, pp. 26–37.

[70] Sudarshan S Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. "Change detection in hierarchically structured information." In: *ACM SIGMOD Record*. Vol. 25. 2. ACM. 1996, pp. 493–504.

[71] Shyam R Chidamber and Chris F Kemerer. "A metrics suite for object oriented design." In: *Software Engineering, IEEE Transactions on* 20.6 (1994), pp. 476–493.

[72] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. "A Metamodel Independent Approach to Difference Representation." In: *Journal of Object Technology* 6.9 (2007), pp. 165–185.

[73] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. "Managing model conflicts in distributed development." In: *Model Driven Engineering Languages and Systems*. Springer, 2008, pp. 311–325.

[74] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. "Model patches in model-driven engineering." In: *Models in Software Engineering*. Springer, 2010, pp. 190–204.

[75] Gregory Cobena, Serge Abiteboul, and Amelie Marian. "Detecting changes in XML documents." In: *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE. 2002, pp. 41–52.

[76] Ben Collins-Sussman, Brian Fitzpatrick, and Michael Pilato. *Version control with subversion*. O'Reilly Media, Inc., 2004.

[77] Reidar Conradi and Bernhard Westfechtel. "Version models for software configuration management." In: *ACM Computing Surveys (CSUR)* 30.2 (1998), pp. 232–282.

[78] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. "Thirty years of graph matching in pattern recognition." In: *International journal of pattern recognition and artificial intelligence* 18.03 (2004), pp. 265–298.

[79] Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. "Algebraic Approaches to Graph Transformation-Part I: Basic Concepts and Double Pushout Approach." In: *Handbook of Graph Grammars*. 1997, pp. 163–246.

[80] Andrea De Lucia, Fausto Fasano, Rocco Oliveto, and Genoveffa Tortora. "ADAMS: ADvanced Artefact Management System." In: *CSMR*. 2006, pp. 349–350.

[81] Andrea De Lucia, Fausto Fasano, Giuseppe Scanniello, and Genoveffa Tortora. "Concurrent fine-grained versioning of UML models." In: *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*. IEEE. 2009, pp. 89–98.

[82] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. "Finding refactorings via change metrics." In: *ACM SIGPLAN Notices*. Vol. 35. 10. ACM. 2000, pp. 166–177.

[83] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. "Automated detection of refactorings in evolving components." In: *ECOOP 2006–Object-Oriented Programming*. Springer, 2006, pp. 404–428.

[84] Danny Dig, Kashif Manzoor, Ralph Johnson, and Tien N Nguyen. "Effective software merging in the presence of object-oriented refactorings." In: *Software Engineering, IEEE Transactions on* 34.3 (2008), pp. 321–335.

[85] Eclipse Foundation. *EPatch*. 2014. URL: http://wiki.eclipse.org/EMF_Compare/Epatch.

[86] Eclipse Foundation. *EMF Compare*. 2015. URL: http://www.eclipse.org/emf/compare.

[87] Eclipse Foundation. *EMF Diff/Merge*. 2015. URL: http://eclipse.org/diffmerge.

[88] Eclipse Foundation. *Eclipse Modeling Framework (EMF)*. 2015. URL: http://eclipse.org/modeling/emf.

[89] Eclipse Foundation. *Eclipse Modeling Project (EMP)*. 2015. URL: http://eclipse.org/modeling.

[90] Eclipse Foundation. *Graphical Modeling Project (GMP)*. 2015. URL: http://eclipse.org/modeling/gmp.

[91] Eclipse Foundation. *Graphical Editing Framework (GEF)*. 2015. URL: http://www.eclipse.org/gef.

[92] Eclipse Foundation. *Xtext*. 2015. URL: http://eclipse.org/Xtext.

[93] Eclipse Foundation. *Papyrus*. 2015. URL: http://eclipse.org/modeling/mdt/?project=papyrus.

[94] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN: 3540311874.

[95] Hartmut Ehrig, Claudia Ermel, and Gabriele Taentzer. “A Formal Resolution Strategy for Operation-Based Conflicts in Model Versioning Using Graph Modifications.” In: *Fundamental Approaches to Software Engineering*. Ed. by Dimitra Giannakopoulou and Fernando Orejas. Vol. 6603. Lecture Notes in Computer Science. Springer, 2011, pp. 202–216.

[96] Karsten Ehrig, Jochen Malte Küster, and Gabriele Taentzer. “Generating instance models from meta models.” In: *Software & Systems Modeling* 8.4 (2009), pp. 479–500.

[97] Holger Eichelberger, Yilmaz Eldogan, Klaus Schmid, and Marienburger Platz. “A Comprehensive Survey of UML Compliance in Current Modelling Tools.” In: *Software Engineering* 143 (2009), pp. 39–50.

[98] Pär Emanuelsson. “There is a strong need for diff/merge tools on models.” In: (2010).

[99] Jacky Estublier, David Leblang, André van der Hoek, Reidar Conradi, Geoffrey Clemm, Walter Tichy, and Darcy Wiborg-Weber. “Impact of software engineering research on the practice of software configuration management.” In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 14.4 (2005), pp. 383–430.

[100] ETAS. *ASCET Software Products*. 2015. URL: http://www.etas.com/en/products/ascet_software_products.php.

[101] Juergen Etzlstorfer, Angelika Kusel, Elisabeth Kapsammer, Philip Langer, Werner Retschitzegger, Johannes Schoenboeck, Wieland Schwinger, and Manuel Wimmer. “A Survey on Incremental Model Transformation Approaches.” In: *Proceedings of the Workshop on Models and Evolution*. Ed. by Alfonso Pierantonio and Bernhard Schätz. Vol. 1090. CEUR Workshop Proceedings. 2013, pp. 4–13.

[102] Uli Fahrenberg, Axel Legay, and Andrzej Wasowski. “Vision Paper: Make a Difference! (Semantically).” In: *Proceedings of the 14th International Conference on Model Driven Engineering Languages and Systems*. Springer, 2011, pp. 490–500.

[103] Stefan Feldmann, Julia Fuchs, and Birgit Vogel-Heuser. "Modularity, variant and version management in plant automation–future challenges and state of the art." In: *Proceedings of DESIGN 2012, the 12th International Design Conference, Dubrovnik, Croatia*. 2012.

[104] Beat Fluri, Michael Wursch, Martin PInzger, and Harald C Gall. "Change distilling: Tree differencing for fine-grained source code change extraction." In: *Software Engineering, IEEE Transactions on* 33.11 (2007), pp. 725–743.

[105] Karl Fogel and Moshe Bar. *Open source development with CVS*. Paraglyph Inc., 2003.

[106] Sabrina Förtsch and Bernhard Westfechtel. "Differencing and merging of software diagrams state of the art and challenges." In: *Proc. Second Intl. Conf. Software and Data Technologies (ICSOFT)*. 2007.

[107] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999. ISBN: 0201485672.

[108] Robert France and Bernhard Rumpe. "Model-driven development of complex software: A research roadmap." In: *2007 Future of Software Engineering*. IEEE Computer Society. 2007, pp. 37–54.

[109] David Gale and Lloyd S Shapley. "College admissions and the stability of marriage." In: *American mathematical monthly* (1962), pp. 9–15.

[110] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. "Managing model adaptation by precise detection of metamodel changes." In: *Model Driven Architecture-Foundations and Applications*. Springer. 2009, pp. 34–49.

[111] Christian Gerth. *Business Process Models. Change Management*. Springer, 2013.

[112] Sinem Getir, André Van Hoorn, Lars Grunske, and Matthias Tichy. "Co-Evolution of Software Architecture and Fault Tree models: An Explorative Case Study on a Pick and Place Factory Automation System." In: *NiM-ALP@ MoDELS*. 2013, pp. 32–40.

[113] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall PTR, 2002.

[114] Holger Giese and Robert Wagner. "Incremental Model Synchronization with Triple Graph Grammars." In: *Intl. Conf. on Model Driven Engineering Languages and Systems*. 2006, pp. 543–557.

[115] Tom Gilb. "Evolutionary development." In: *ACM SIGSOFT Software Engineering Notes* 6.2 (1981), pp. 17–17.

[116] Ursula Goltz, Ralf H Reussner, Michael Goedicke, Wilhelm Hasselbring, Lukas Märtin, and Birgit Vogel-Heuser. "Design for future: managed software evolution." In: *Computer Science-Research and Development* (2014), pp. 1–11.

[117] Jack Greenfield and Keith Short. "Software factories: assembling applications with patterns, models, frameworks and tools." In: *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM. 2003, pp. 16–27.

[118] David F Haasl, NH Roberts, WE Vesely, and FF Goldberg. *Fault tree handbook*. Nuclear Regulatory Commission, Washington, DC (USA). Office of Nuclear Regulatory Research, 1981.

[119] Arne Haber, Katrin Hölldobler, Carsten Kolassa, Markus Look, Bernhard Rumpe, Klaus Müller, and Ina Schaefer. "Engineering delta modeling languages." In: *Proceedings of the 17th International Software Product Line Conference*. ACM. 2013, pp. 22–31.

[120] Arne Haber, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. "Delta modeling for software architectures." In: *arXiv preprint arXiv:1409.2358* (2014).

[121] David Harel and Bernhard Rumpe. "Meaningful modeling: what's the semantics of "semantics"?" In: *Computer* 37.10 (2004), pp. 64–72.

[122] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. "Confluence of typed attributed graph transformation systems." In: *Graph Transformation*. Springer, 2002, pp. 161–176.

[123] Florian Heidenreich, Jan Kopcsek, and Uwe Aßmann. "Safe composition of transformations." In: *Theory and Practice of Model Transformations*. Springer, 2010, pp. 108–122.

[124] Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, Yingfei Xiong, Susann Gottmann, and Thomas Engel. "Model synchronization based on triple graph grammars: correctness, completeness and invertibility." In: *Software & Systems Modeling* (2013), pp. 1–29.

[125] Markus Herrmannsdoerfer and Maximilian Koegel. "Towards a generic operation recorder for model evolution." In: *Proceedings of the 1st International Workshop on Model Comparison in Practice*. ACM. 2010, pp. 76–81.

[126] Markus Herrmannsdoerfer, Daniel Ratiu, and Guido Wachsmuth. "Language evolution in practice: The history of GMF." In: *Software Language Engineering*. Springer, 2010, pp. 3–22.

[127] Berthold Hoffmann and Mark Minas. "Generating instance graphs from class diagrams with adaptive star grammars." In: *GCM 2010* (2011), p. 49.

[128] James J Hunt, Kiem-Phong Vo, and Walter F Tichy. "Delta algorithms: An empirical analysis." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 7.2 (1998), pp. 192–214.

[129] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006. ISBN: 0262101149.

[130] Praveen Jayaraman, Jon Whittle, Ahmed M Elkhodary, and Hassan Gomaa. "Model composition in product lines and feature interaction detection using critical pair analysis." In: *Model Driven Engineering Languages and Systems*. Springer, 2007, pp. 151–165.

[131] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, Ivan Kurtev, and Patrick Valduriez. "ATL: a QVT-like transformation language." In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. ACM. 2006, pp. 719–720.

[132] Frédéric Jouault and Jean Bézivin. "KM3: a DSL for Metamodel Specification." In: *Formal Methods for Open Object-Based Distributed Systems*. Springer, 2006, pp. 171–185.

[133] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990.

[134] Timo Kehrer and Udo Kelter. "Versioning of Ordered Model Element Sets." In: *Softwaretechnik-Trends* 34.2 (2014).

[135] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[136] Udo Kelter, Maik Schmidt, and Sven Wenzel. "Architekturen von Differenzwerkzeugen für Modelle." In: *Software Engineering*. 2008, pp. 155–168.

[137] Udo Kelter and Maik Schmidt. "Comparing state machines." In: *Proceedings of the 2008 international workshop on Comparison and versioning of software models*. ACM. 2008, pp. 1–6.

[138] Udo Kelter, Jürgen Wehren, and Jörg Niere. "A Generic Difference Algorithm for UML Models." In: *Software Engineering* 64.105-116 (2005), pp. 4–9.

[139] Udo Kelter. "Repository-Dienste fur die modellbasierte Entwicklung." In: *Design for Future–Langlebige Softwaresysteme* (2009), p. 76.

[140] Udo Kelter. "Pseudo-Modelldifferenzen und die Phasenabhängigkeit von Metamodellen." In: *Software Engineering*. 2010, pp. 117–128.

[141] Sanjeev Khanna, Keshav Kunal, and Benjamin C Pierce. "A formal investigation of diff3." In: *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science*. Springer, 2007, pp. 485–496.

[142] Doug Kimelman, Marsha Kimelman, David Mandelin, and Daniel M Yellin. "Bayesian approaches to matching architectural diagrams." In: *Software Engineering, IEEE Transactions on* 36.2 (2010), pp. 248–274.

[143] Maximilian Kögel. "TIME-Tracking Intra-and Inter-Model Evolution." In: *Software Engineering (Workshops)*. 2008, pp. 157–164.

[144] Maximilian Kögel, Jonas Helming, and Stephan Seyboth. "Operation-based conflict detection and resolution." In: *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*. IEEE Computer Society. 2009, pp. 43–48.

[145] Maximilian Kögel and Jonas Helming. "EMFStore: a model repository for EMF models." In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM. 2010, pp. 307–308.

[146] Dimitrios S Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F Paige. "Different models for model matching: An analysis of approaches to support model differencing." In: *Comparison and Versioning of Software Models, 2009. CVSM'09. ICSE Workshop on*. IEEE. 2009, pp. 1–6.

[147] Dimitrios S Kolovos, Richard F Paige, Fiona AC Polack, and Louis M Rose. "Update transformations in the small with the epsilon wizard language." In: *Journal of Object Technology (JOT)* (2003).

[148] Dimitrios S Kolovos. “Establishing correspondences between models with the epsilon comparison language.” In: *Model Driven Architecture-Foundations and Applications*. Springer. 2009, pp. 146–157.

[149] Patrick Könemann. “Capturing the intention of model changes.” In: *Model Driven Engineering Languages and Systems*. Springer, 2010, pp. 108–122.

[150] Patrick Könemann. “Semantic grouping of model changes.” In: *Proceedings of the 1st International Workshop on Model Comparison in Practice*. ACM. 2010, pp. 50–55.

[151] Patrick Könemann and Olaf Zimmermann. “Linking design decisions to design models in model-based software development.” In: *Software Architecture*. Springer, 2010, pp. 246–262.

[152] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Giuliano Antoniol, and Y-G Gueheneuc. “Madmatch: Many-to-many approximate diagram matching for design comparison.” In: *Software Engineering, IEEE Transactions on* 39.8 (2013), pp. 1090–1111.

[153] Thomas Kühne. “Matters of (meta-) modeling.” In: *Software & Systems Modeling* 5.4 (2006), pp. 369–385.

[154] Jochen M Küster, Christian Gerth, Alexander Förster, and Gregor Engels. “Detecting and resolving process model differences in the absence of a change log.” In: *Business Process Management*. Springer, 2008, pp. 244–260.

[155] Philip Langer. “Adaptable Model Versioning based on Model Transformation By Demonstration.” PhD thesis. Vienna University of Technology, Institute of Software Technology and Interactive Systems, 2012.

[156] Philip Langer, Tanja Mayerhofer, and Gerti Kappel. “Semantic Model Differencing Utilizing Behavioral Semantics Specifications.” In: *Model-Driven Engineering Languages and Systems*. Springer, 2014, pp. 116–132.

[157] Philip Langer, Manuel Wimmer, Petra Brosch, Markus Herrmannsdörfer, Martina Seidl, Konrad Wieland, and Gerti Kappel. “A posteriori operation detection in evolving software models.” In: *Journal of Systems and Software* 86.2 (2013), pp. 551–566.

[158] Philip Langer, Manuel Wimmer, Jeff Gray, Gerti Kappel, and Antonio Vallecillo. “Language-Specific Model Versioning Based on Signifiers.” In: *Journal of Object Technology* 11.3 (2012), pp. 4–1.

[159] Craig Larman and Victor R Basili. “Iterative and incremental development: A brief history.” In: *Computer* 36.6 (2003), pp. 47–56.

[160] Meir M Lehman. “Programs, life cycles, and laws of software evolution.” In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076.

[161] Kim Letkeman. “Comparing and merging UML models in IBM Rational Software Architect.” In: *IBM Rational, July* 39 (2005).

[162] Yuehua Lin, Jeff Gray, and Frédéric Jouault. “DSMDiff: a differentiation tool for domain-specific models.” In: *European Journal of Information Systems* 16.4 (2007), pp. 349–361.

[163] Tancred Lindholm. “A three-way merge for XML documents.” In: *Proceedings of the 2004 ACM symposium on Document engineering*. ACM. 2004, pp. 1–10.

[164] Ernst Lippe and Norbert Van Oosterom. "Operation-based merging." In: *ACM SIGSOFT Software Engineering Notes*. Vol. 17. 5. ACM. 1992, pp. 78–87.

[165] Jon Loeliger and Matthew McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, Inc., 2012.

[166] Malte Lochau, Sebastian Oster, Ursula Goltz, and Andy Schürr. "Model-based pairwise testing for feature interaction coverage in software product line engineering." In: *Software Quality Journal* 20.3-4 (2012), pp. 567–604.

[167] Roberto E Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Alexander Egyed, and Enrique Alba. "Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of software product lines." In: *Evolutionary Computation (CEC), 2014 IEEE Congress on*. IEEE. 2014, pp. 387–396.

[168] Jochen Ludewig. "Models in software engineering–an introduction." In: *Software and Systems Modeling* 2.1 (2003), pp. 5–14.

[169] David MacKenzie, Paul Eggert, and Richard Stallman. *Comparing and Merging Files with GNU diff and patch*. Network Theory Ltd., 2003.

[170] Miroslaw Malek. "The Art of Creating Models and Models Integration." In: *Model-Based Software and Data Integration*. Springer, 2008, pp. 1–7.

[171] David Mandelin, Doug Kimelman, and Daniel Yellin. "A Bayesian approach to diagram matching with application to architectural models." In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 222–231.

[172] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. "ADDiff: semantic differencing for activity diagrams." In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM. 2011, pp. 179–189.

[173] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. "A manifesto for semantic model differencing." In: *Models in Software Engineering*. Springer, 2011, pp. 194–203.

[174] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. "CDDiff: Semantic differencing for class diagrams." In: *ECOOP 2011–Object-Oriented Programming*. Springer, 2011, pp. 230–254.

[175] Slaviša Marković and Thomas Baar. "Refactoring OCL annotated UML class diagrams." In: *Model Driven Engineering Languages and Systems*. Springer, 2005, pp. 280–294.

[176] MathWorks. *MATLAB/Simulink*. 2015. URL: `http://www.mathworks.com`.

[177] Akhil Mehra, John Grundy, and John Hosking. "A generic approach to supporting diagram differencing and merging for collaborative design." In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. ACM. 2005, pp. 204–213.

[178] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. "Similarity flooding: A versatile graph matching algorithm and its application to schema matching." In: *Data Engineering, 2002. Proceedings. 18th International Conference on*. IEEE. 2002, pp. 117–128.

[179] Sergey Melnik, Erhard Rahm, and Philip A Bernstein. "Rondo: A programming platform for generic model management." In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM. 2003, pp. 193–204.

[180] Tom Mens. "A state-of-the-art survey on software merging." In: *Software Engineering, IEEE Transactions on* 28.5 (2002), pp. 449–462.

[181] Tom Mens. "On the use of graph transformations for model refactoring." In: *Generative and transformational techniques in software engineering*. Springer, 2006, pp. 219–257.

[182] Bertrand Meyer. "Applying "Design by Contract"." In: *IEEE Computer* 25.10 (1992), pp. 40–51.

[183] Leonardo Murta, Chessman Corrêa, João Gustavo Prudêncio, and Cláudia Werner. "Towards odyssey-VCS 2: improvements over a UML-based version control system." In: *Proceedings of the 2008 international workshop on Comparison and versioning of software models*. ACM. 2008, pp. 25–30.

[184] Shiva Nejati, Mehrdad Sabetzadeh, Marsha Chechik, Steve Easterbrook, and Pamela Zave. "Matching and merging of statecharts specifications." In: *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society. 2007, pp. 54–64.

[185] Hoan Anh Nguyen, Tung Thanh Nguyen, Hung Viet Nguyen, and Tien N Nguyen. "iDiff: Interaction-based program differencing tool." In: *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE. 2011, pp. 572–575.

[186] Jörg Niere. "Visualizing differences of UML diagrams with Fujaba." In: *Proceedings of the International Fujaba Days*. 2004, pp. 31–34.

[187] No Magic. *MagicDraw*. 2015. URL: `http://www.nomagic.com/products/magicdraw.html`.

[188] Object Management Group. *Human-Usable Textual Notation (HUTN) Specification, version 1.0*. OMG document number: formal/04-08-01. 2004.

[189] Object Management Group. *UML 2.4.1 Infrastructure Specification*. OMG Document Number: formal/2011-08-05. 2011.

[190] Object Management Group. *UML 2.4.1 Superstructure Specification*. OMG Document Number: formal/2011-08-06. 2011.

[191] Object Management Group. *Systems Modeling Language (SysML), version 1.3*. OMG Document Number: formal/2012-06-01. 2012.

[192] Object Management Group. *The UML Profile for MARTE: Modeling and Analysis of Real-Time and Embedded Systems*. 2013. URL: `http://www.omgmarte.org/`.

[193] Object Management Group. *Meta Object Facility (MOF) Core Specification, version 2.4.1*. OMG document number: formal/2013-06-01. 2013.

[194] Object Management Group. *Model Driven Architecture (MDA), MDA Guide rev. 2.0*. OMG document number: ormsc/2014-06-01. 2014.

[195] Object Management Group. *Object Constraint Language (OCL), version 2.4*. OMG document number: formal/2014-02-03. 2014.

[196] Object Management Group. *XML Metadata Interchange (XMI), version 2.4.2*. OMG document number: formal/2014-04-04. 2014.

[197] Takafumi Oda and Motoshi Saeki. "Generative technique of version control systems for software diagrams." In: *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE. 2005, pp. 515–524.

[198] Dirk Ohst, Michael Welle, and Udo Kelter. "Differences between versions of UML diagrams." In: *ACM SIGSOFT Software Engineering Notes* 28.5 (2003), pp. 227–236.

[199] Hamilton Oliveira, Leonardo Murta, and Cláudia Werner. "Odyssey-VCS: a flexible version control system for UML model elements." In: *Proceedings of the 12th international workshop on Software configuration management*. ACM. 2005, pp. 1–16.

[200] Pit Pietsch, Hamed Shariat Yazdi, and Udo Kelter. "Generating realistic test models for model processing tools." In: *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*. IEEE. 2011, pp. 620–623.

[201] Pit Pietsch, Hamed Shariat Yazdi, Udo Kelter, and Timo Kehrer. "Assessing the quality of model differencing engines." In: *Softwaretechnik-Trends* 32.4 (2012), pp. 47–48.

[202] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN: 3540243720.

[203] Ivan Porres. *Model refactorings as rule-based update transformations*. Springer, 2003.

[204] Rachel A Pottinger and Philip A Bernstein. "Merging models based on given correspondences." In: *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment. 2003, pp. 862–873.

[205] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. "Template-based reconstruction of complex refactorings." In: *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE. 2010, pp. 1–10.

[206] Erhard Rahm and Philip A Bernstein. "A survey of approaches to automatic schema matching." In: *the VLDB Journal* 10.4 (2001), pp. 334–350.

[207] Raghu Reddy, Robert France, Sudipto Ghosh, Franck Fleurey, and Benoit Baudry. "Model composition–a signature-based approach." In: *Aspect Oriented Modeling (AOM) Workshop*. 2005.

[208] Dennis Reuling. "Integration of UML Profiles into the SiDiff and SiLift tools." MA thesis. University of Siegen, 2013.

[209] José E Rivera and Antonio Vallecillo. "Representing and operating with model differences." In: *Objects, Components, Models and Patterns*. Springer, 2008, pp. 141–160.

[210] Marc J. Rochkind. "The source code control system." In: *Software Engineering, IEEE Transactions on* 4 (1975), pp. 364–370.

[211] Grzegorz Rozenberg, ed. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume I. Foundations*. River Edge, NJ, USA: World Scientific Publishing Co., Inc., 1997. ISBN: 98-102288-48.

[212] Thomas Ruhroth, Stefan Gärtner, Jens Bürger, Jan Jürjens, and Kurt Schneider. "Versioning and Evolution Requirements for Model-Based System Development." In: *Softwaretechnik-Trends* 34.2 (2014).

[213] Mehrdad Sabetzadeh, Shiva Nejati, Steve Easterbrook, and Marsha Chechik. "A relationship-driven framework for model merging." In: *Modeling in Software Engineering, 2007. MISE'07: ICSE Workshop 2007. International Workshop on.* IEEE. 2007, pp. 2–2.

[214] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. "Delta-oriented programming of software product lines." In: *Software Product Lines: Going Beyond.* Springer, 2010, pp. 77–91.

[215] Douglas C Schmidt. "Guest editor's introduction: Model-driven engineering." In: *Computer* 39.2 (2006), pp. 0025–31.

[216] Maik Schmidt and Tilman Gloetzner. "Constructing difference tools for models using the SiDiff framework." In: *Companion of the 30th international conference on Software engineering.* ACM. 2008, pp. 947–948.

[217] Maik Schmidt, Sven Wenzel, Timo Kehrer, and Udo Kelter. "History-based merging of models." In: *Comparison and Versioning of Software Models, 2009. CVSM'09. ICSE Workshop on.* IEEE. 2009, pp. 13–18.

[218] Christian Schneider, Albert Zündorf, and Jörg Niere. "CoObRA-a small step for development tools to collaborative environments." In: *Workshop on Directions in Software Engineering Environments.* 2004.

[219] Christian Schneider. "CoObRA: Eine Plattform zur Verteilung und Replikation komplexer Objektstrukturen mit optimistischen Sperrkonzepten." PhD thesis. Universität Kassel, Fachgebiet Software Engineering, 2007.

[220] Felix Schwägerl, Sabrina Uhrig, and Bernhard Westfechtel. "Demonstration of a Tool for Consistent Three-way Merging of EMF Models." In: *Joint track on Tools, Demos & Posters of ECOOP, ECSA and ECMFA* (2013), p. 26.

[221] Christoph Seidl, Ina Schaefer, and Uwe Aßmann. "DeltaEcore-A Model-Based Delta Language Generation Framework." In: *Modellierung.* 2014, pp. 81–96.

[222] Bran Selic. "The pragmatics of model-driven development." In: *IEEE software* 20.5 (2003), pp. 19–25.

[223] Petri Selonen. "A review of UML model comparison approaches." In: *Nordic Workshop on Model Driven Engineering.* 2007, pp. 37–51.

[224] Petri Selonen and Markus Kettunen. "Metamodel-based inference of inter-model correspondence." In: *Software Maintenance and Reengineering, 2007. CSMR'07. 11th European Conference on.* IEEE. 2007, pp. 71–80.

[225] Danhua Shao, Sarfraz Khurshid, and Dewayne E Perry. "Evaluation of semantic interference detection in parallel changes: an exploratory experiment." In: *Software Maintenance, 2007. ICSM 2007. IEEE International Conference on.* IEEE. 2007, pp. 74–83.

[226] Prawee Sriplakich, Xavier Blanc, and M-P Gervais. "Supporting collaborative development in an open MDA environment." In: *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on*. IEEE. 2006, pp. 244–253.

[227] Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973. ISBN: 978-3211811061.

[228] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[229] Matthew Stephan and James R Cordy. "A Survey of Model Comparison Approaches and Applications." In: *Modelsward*. 2013, pp. 265–277.

[230] Daniel Strüber, Julia Rubin, Marsha Chechik, and Gabriele Taentzer. "A Variability-Based Approach to Reusable and Efficient Model Transformations." In: *Fundamental Approaches to Software Engineering*. Springer, 2015.

[231] Gerson Sunyé, Damien Pollet, Yves Le Traon, and Jean-Marc Jézéquel. "Refactoring UML models." In: *UML 2001–The Unified Modeling Language. Modeling Languages, Concepts, and Tools*. Springer, 2001, pp. 134–148.

[232] Gabriele Taentzer. "AGG: A graph transformation environment for modeling and validation of software." In: *Applications of Graph Transformations with Industrial Relevance*. Springer, 2004, pp. 446–453.

[233] Gabriele Taentzer, André Crema, René Schmutzler, and Claudia Ermel. "Generating domain-specific model editors with complex editing commands." In: *Applications of Graph Transformations with Industrial Relevance*. Springer, 2008, pp. 98–103.

[234] Gabriele Taentzer. "Instance generation from type graphs with arbitrary multiplicities." In: *Electronic Communications of the EASST* 47 (2012).

[235] Gabriele Taentzer, Claudia Ermel, Philip Langer, and Manuel Wimmer. "A fundamental approach to model versioning based on graph modifications: from theory to implementation." In: *Software & Systems Modeling* 13.1 (2014), pp. 239–272.

[236] Kuo-Chung Tai. "The tree-to-tree correction problem." In: *Journal of the ACM (JACM)* 26.3 (1979), pp. 422–433.

[237] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[238] G Lorenzo Thione and Dewayne E Perry. "Parallel changes: Detecting semantic interferences." In: *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*. Vol. 1. IEEE. 2005, pp. 47–56.

[239] Thomas Thüm, Don Batory, and Christian Kästner. "Reasoning about edits to feature models." In: *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE. 2009, pp. 254–264.

[240] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. "FeatureIDE: An extensible framework for feature-oriented software development." In: *Science of Computer Programming* 79 (2014), pp. 70–85.

[241] Walter F Tichy. "Design, implementation, and evaluation of a revision control system." In: *Proceedings of the 6th international conference on Software engineering*. IEEE Computer Society Press. 1982, pp. 58–67.

[242] Walter F Tichy. "The string-to-string correction problem with block moves." In: *ACM Transactions on Computer Systems (TOCS)* 2.4 (1984), pp. 309–321.

[243] Walter F Tichy. "Tools for Software Configuration Management." In: *SCM* 30 (1988), pp. 1–20.

[244] Matthias Tichy, Christian Krause, and Grischa Liebel. "Detecting Performance Bad Smells for Henshin Model Transformations." In: *AMT@ MoDELS* 1077 (2013).

[245] Christoph Treude, Stefan Berlik, Sven Wenzel, and Udo Kelter. "Difference computation of large models." In: *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM. 2007, pp. 295–304.

[246] University of Siegen, Software Engineering Group. *The SiDiff Project*. 2015. URL: `http://www.sidiff.org`.

[247] University of Siegen, Software Engineering Group. *The SiLift Project*. 2015. URL: `http://pi.informatik.uni-siegen.de/Projekte/SiLift`.

[248] Ragnhild Van Der Straeten, Tom Mens, and Stefan Van Baelen. "Challenges in model-driven software engineering." In: *Models in Software Engineering*. Springer, 2009, pp. 35–47.

[249] Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. *Faster and more focused control-flow analysis for business process models through SESE decomposition*. Springer, 2007.

[250] Bert Vanhooff, Dhouha Ayed, Stefan Van Baelen, Wouter Joosen, and Yolande Berbers. "Uniti: A unified transformation infrastructure." In: *Model Driven Engineering Languages and Systems*. Springer, 2007, pp. 31–45.

[251] Rajesh Vasa, Markus Lumpe, and Allan Jones. *Helix-Software Evolution Data Set*. 2010. URL: `http://www.ict.swin.edu.au/research/projects/helix`.

[252] Sander D Vermolen, Guido Wachsmuth, and Eelco Visser. "Reconstructing complex metamodel evolution." In: *Software Language Engineering*. Springer, 2012, pp. 201–221.

[253] Birgit Vogel-Heuser, Jens Folmer, and Christoph Legat. "Anforderungen an die Softwareevolution in der Automatisierung des Maschinen- und Anlagenbaus." In: *at–Automatisierungstechnik* 62.3 (2014), pp. 163–174.

[254] B Vogel-Heuser, C Legat, J Folmer, and S Feldmann. *Researching evolution in industrial plant automation: Scenarios and documentation of the pick and place unit*. Technical Report TUM-AIS-TR-01-14-02, Institute of Automation and Information Systems, Technische Universität München, 2014.

[255] Valeriy Vyatkin. "Software engineering in industrial automation: State-of-the-art review." In: *Industrial Informatics, IEEE Transactions on* 9.3 (2013), pp. 1234–1249.

[256] Holger Wache, Thomas Voegele, Ubbo Visser, Heiner Stuckenschmidt, Gerhard Schuster, Holger Neumann, and Sebastian Hübner. "Ontology-based integration of information – a survey of existing approaches." In: *IJCAI-01 workshop: ontologies and information sharing*. Vol. 2001. 2001, pp. 108–117.

[257] Robert A Wagner and Michael J Fischer. "The string-to-string correction problem." In: *Journal of the ACM (JACM)* 21.1 (1974), pp. 168–173.

[258] Yuan Wang, David J DeWitt, and J-Y Cai. “X-Diff: An effective change detection algorithm for XML documents.” In: *Data Engineering, 2003. Proceedings. 19th International Conference on.* IEEE. 2003, pp. 519–530.

[259] Sven Wenzel. “Unique identification of elements in evolving models: towards fine-grained traceability in model-driven engineering.” PhD thesis. Universität Siegen, Fachbereich 12, Elektrotechnik und Informatik, 2011.

[260] Bernhard Westfechtel. “Merging of EMF models.” In: *Software & Systems Modeling* 13.2 (2014), pp. 757–788.

[261] Konrad Wieland, Philip Langer, Martina Seidl, Manuel Wimmer, and Gerti Kappel. “Turning conflicts into collaboration.” In: *Computer Supported Cooperative Work (CSCW)* 22.2-3 (2013), pp. 181–240.

[262] Manuel Wimmer and Gerhard Kramler. “Bridging grammarware and modelware.” In: *Satellite Events at the MoDELS 2005 Conference.* Springer. 2006, pp. 159–168.

[263] World Wide Web Consortium (W3C). *XML Path Language (XPath) 3.0.* 2014. URL: `http://www.w3.org/TR/2014/REC-xpath-30-20140408`.

[264] Zhenchang Xing and Eleni Stroulia. “UMLDiff: an algorithm for object-oriented design differencing.” In: *Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE).* ACM. 2005, pp. 54–65.

[265] Zhenchang Xing and Eleni Stroulia. “Refactoring detection based on umldiff change-facts queries.” In: *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on.* IEEE. 2006, pp. 263–274.

[266] Zhenchang Xing and Eleni Stroulia. “Differencing logical UML models.” In: *Automated Software Engineering* 14.2 (2007), pp. 215–259.

[267] Zhenchang Xing. “Model comparison with GenericDiff.” In: *Proc. of the IEEE/ACM International Conference on Automated Software Engineering.* ACM. 2010, pp. 135–138.

[268] Wuu Yang. “Identifying syntactic differences between two programs.” In: *Software: Practice and Experience* 21.7 (1991), pp. 739–755.

[269] Yijun Yu, Thein Than Tun, and Bashar Nuseibeh. “Specifying and detecting meaningful changes in programs.” In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering.* IEEE Computer Society. 2011, pp. 273–282.