# Evolutionary Metamodeling

Markus Herrmannsdörfer

Technische Universität München

TECHNISCHE UNIVERSITÄT MÜNCHEN
Institut für Informatik

# Evolutionary Metamodeling

*Markus Herrmannsdörfer*

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

| | |
|---|---|
| Vorsitzender: | Univ.-Prof. Dr. Helmut Seidl |
| Prüfer der Dissertation: | |
| 1. | Univ.-Prof. Dr. Dr. h.c. Manfred Broy |
| 2. | Univ.-Prof. Dr. Oscar Nierstrasz,<br>Universität Bern, Schweiz |

Die Dissertation wurde am 07.07.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 12.09.2011 angenommen.

# Abstract

Model-based software development promises to increase productivity and quality through domain-specific modeling languages. In response, modeling languages are receiving increased adoption in industry. With the integration of modeling languages into industrial development practice, their maintenance is gaining importance. Like software, modeling languages and thus their metamodels are subject to evolution due to changing requirements. When a metamodel is adapted to the new requirements, existing models may no longer conform to it. To be able to use the existing models with the evolved modeling language, they need to be migrated.

Support for model migration in response to metamodel adaptation faces two challenges. First, to reduce migration effort, the model migration needs to be automated as far as possible. However, there is no empirical knowledge about the extent to which model migration can be automated in practice. Second, the model migration needs to ensure that the meaning of a possibly unknown set of models is preserved. However, existing approaches require to specify the migration after the complete metamodel adaptation, thereby losing the intention behind the changes.

This thesis contributes to both challenges. First, to determine the potential for automating model migration in practice, we performed an empirical study on the histories of two industrial metamodels. The study showed that models can be migrated automatically in practice. Moreover, we found out that effort can be significantly reduced by reuse of recurring migrations, while expressiveness is required to define custom migrations.

Second, we present our novel method COPE that provides the desired level of reuse and expressiveness. To not lose the intention behind the metamodel changes, COPE records the model migration together with the metamodel adaptation—we call this the coupled evolution of metamodels and models. COPE records the coupled evolution as a sequence of coupled operations in an explicit history model. Each coupled operation encapsulates both metamodel adaptation as well as reconciling model migration. Recurring coupled operations can be reused through a library to significantly reduce migration effort. Expressiveness is provided by custom coupled operations which need to be specified manually. Using the history model, existing models can be automatically migrated to the adapted version of the metamodel.

To demonstrate the applicability of COPE in practice, we used it in six real-life case studies to automate model migration in response to metamodel adaptation. We applied COPE to reverse engineer the coupled evolution, used it to directly evolve a modeling language, and compared it to other model migration and transformation tools. All the case studies show that more than 95% of the coupled evolution can be covered by reusable coupled operations and that only very few custom migrations are required. Moreover, the comparison case studies indicate that recording the changes in a history model is more likely to lead to a semantics-preserving model migration than specifying the migration after the changes occurred.

Finally, the case studies revealed that COPE supports an evolutionary process for developing a modeling language. To show that, we propose methods to recommend operations for metamodel improvement by analyzing the built models and to extend the operations to also adapt the semantics definition of the modeling language.

# Acknowledgements

I would like to thank all the people who helped me to make this dissertation a success. First, I want to express my gratitude to my supervisor Manfred Broy who has always been very supportive, although my topics are not very close to his main research interests. Only the close connections to industry and the multitude of opportunities that he offered me made this dissertation possible. I would also like to thank my co-supervisor Oscar Nierstrasz who cordially invited me to get to know his research group and who gave me valuable feedback.

In addition, my thanks also goes to all the people with which I had the pleasure to work over the last three and a half years. Special thanks go to Sebastian Benz and Elmar Jürgens for introducing me to scientific working and publishing; to Sabine Rittmann for helping me to get to know the chair of Prof. Broy and for providing me the template for this dissertation; to Stefano Merenda for our discussions on metamodeling; to Daniel Ratiu for giving my dissertation new directions; and to Maximilian Kögel for the joint work on model and metamodel evolution.

This dissertation is based on a number of already published papers. I am grateful to the co-authors which supported me to publish these papers: Sebastian Benz, Kelly Garcès, Elmar Jürgens, Maximilian Kögel, Dimitrios Kolovos, Richard Paige, Daniel Ratiu, Louis Rose, Sander Vermolen, Guido Wachsmuth, and James Williams. I would also like to thank the reviewers that tremendously helped me to improve the dissertation with their comments: Sebastian Benz, Peter Braun, Benedikt Hauptmann, Benjamin Hummel, Lars Heinemann, Florian Hölzl, Elmar Jürgens, Maximilian Junker, Thomas Kofler, Daniel Mendez-Fernandez, Daniel Ratiu, Andreas Vogelsang, and Doris Wild.

At the chair of Prof. Broy, I have worked together with many people in different research projects. Thank you for supporting this dissertation in these projects and making work so much fun: Sabine Rittmann (InServe); Wolfgang Haberl, Stefan Kugele, Stefano Merenda, Michael Tautschnig, Zhonglei Wang, and Doris Wild (Base.XT); Florian Deißenböck, Martin Feilkas, and Elmar Jürgens (SoQuo); Lars Heinemann, Klaus Lochmann, and Stefan Wagner (Quamoco); Alexander Harhurin, Florian Hölzl, Thomas Kofler, Christian Leuxner, Daniel Ratiu, and Judith Thyssen (SPES).

I also want to express my gratitude to the people who allowed me to perform interesting case studies: Steffen Becker and Klaus Krogmann for providing the Palladio Component Model (PCM); Maximilian Kögel and Jonas Helming for giving access to Unicase; Pieter van Gorp, Steffen Mazanek, and Arend Rensink for organizing the Transformation Tool Contest (TTC); Louis Rose, David Williams, Kelly Garcès, and Dimitrios Kolovos for asking me to participate in the comparison of migration tools. Further thanks go to Antonio Cicchetti, Thomas Goldschmidt, Steven Kelly, Anneke Kleppe, Ralf Lämmel, Ed Merks, Alfonso Pierantonio, Davide di Ruscio, Juha-Pekka Tolvanen, Markus Voelter, and Eelco Visser for valuable discussions.

Last but no least, I am grateful to my parents Luitgard and Uwe and my sister Martina for supporting me through all these years of education. I would also like to thank my girlfriend Michaela for the free time spent together that effectively distracted me from this work. Without their support, this work would never have been possible.

*"It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change."*

Charles Darwin

# Contents

**1 Introduction** **15**
1.1 Context: Modeling Languages . . . 15
1.2 Problem: Modeling Language Evolution . . . 17
1.3 Thesis: Recording Metamodel Adaptations . . . 19
1.4 Approach: Evolutionary Metamodeling . . . 20
1.5 Contributions of this Thesis . . . 21
1.6 Outline of this Thesis . . . 23

**2 Background: Engineering of Modeling Languages** **25**
2.1 Model-based Development . . . 25
2.1.1 Models and Modeling Languages . . . 26
2.1.2 Benefits and Risks . . . 27
2.1.3 The Quest for Abstraction . . . 28
2.1.4 Major Initiatives . . . 28
2.2 Metamodeling – Modeling the Abstract Syntax of Modeling Languages 29
2.2.1 Meta Object Facility . . . 30
2.2.2 Abstract Syntax of a Modeling Language . . . 31
2.2.3 Simplified E-MOF Metametamodel . . . 35
2.2.4 Complete E-MOF Metametamodel . . . 38
2.2.5 UML Object and Class Diagrams . . . 41
2.2.6 Eclipse Modeling Framework . . . 43
2.3 Concrete Syntax of Modeling Languages . . . 45
2.3.1 Concrete Syntax of a Modeling Language . . . 45
2.3.2 Implementing the Concrete Syntax . . . 46
2.4 Semantics of Modeling Languages . . . 48
2.4.1 Semantics of a Modeling Language . . . 49
2.4.2 Implementing the Semantics . . . 51
2.5 Evolution of Modeling Languages . . . 53
2.5.1 Reasons for Language Evolution . . . 54
2.5.2 Metamodel and Semantics Evolution . . . 55
2.5.3 Breaking Metamodel Changes . . . 57
2.5.4 Model Migration . . . 59
2.5.5 Model Transformation for Model Migration . . . 61
2.6 Summary . . . 64

**3 State of the Practice: Automatability of Model Migration 65**
3.1 State of the Art . . . 66
3.2 Classification of Metamodel Changes . . . 67
3.2.1 Running Example . . . 68
3.2.2 Model-Specific Coupled Change . . . 69
3.2.3 Model-Independent, Metamodel-Specific Coupled Change . . . 70
3.2.4 Metamodel-Independent Coupled Change . . . 71
3.3 Study Design . . . 72
3.3.1 Study Goal . . . 73
3.3.2 Study Object . . . 73
3.3.3 Study Execution . . . 74
3.4 Study Implementation . . . 75
3.4.1 Study Result . . . 75
3.4.2 Study Discussion . . . 75
3.4.3 Threats to Validity . . . 77
3.5 Requirements for Automating Model Migration . . . 78
3.6 Summary . . . 78

**4 State of the Art: A Cross-Space Survey on Coupled Evolution 81**
4.1 Cross-Space Terminology . . . 82
4.2 Review Systematics . . . 84
4.2.1 Search Strategy . . . 84
4.2.2 Selection Criteria . . . 84
4.3 Classification of Approaches . . . 86
4.3.1 Technical Space . . . 86
4.3.2 Evolution . . . 86
4.3.3 Migration . . . 87
4.3.4 Evaluation . . . 88
4.4 Dataware . . . 89
4.4.1 Relational Dataware . . . 89
4.4.2 Object-Oriented Dataware . . . 91
4.5 Grammarware . . . 95
4.6 XMLware . . . 96
4.7 Modelware . . . 97
4.8 Cross-Space Comparison . . . 99
4.9 Motivation of our Approach . . . 101
4.9.1 Requirements . . . 101
4.9.2 Classification . . . 102
4.10 Summary . . . 103

**5 COPE – Coupled Evolution of Metamodels and Models 105**
5.1 COPE in a Nutshell . . . 106
5.1.1 Running Example . . . 106
5.1.2 Incremental Coupled Evolution . . . 107
5.1.3 Coupled Operations . . . 108
5.1.4 Custom Coupled Operations . . . 110
5.1.5 Reusable Coupled Operations . . . 112
5.1.6 Classification of Coupled Operations . . . 114

5.2 Library of Reusable Coupled Operations . . . 116
5.2.1 Origins of Reusable Coupled Operations . . . 116
5.2.2 Overview of the Library . . . 117
5.2.3 Structural Primitives . . . 118
5.2.4 Non-Structural Primitives . . . 119
5.2.5 Specialization / Generalization Operations . . . 121
5.2.6 Inheritance Operations . . . 123
5.2.7 Delegation Operations . . . 124
5.2.8 Replacement Operations . . . 126
5.2.9 Merge / Split Operations . . . 128
5.2.10 Discussion . . . 129
5.3 Language to Specify the Coupled Evolution . . . 131
5.3.1 Decoupling Metamodel and Model . . . 131
5.3.2 Breaking Metamodel Changes Revisited . . . 133
5.3.3 Primitives for Metamodel Adaptation and Model Migration . . . 135
5.3.4 Implementing Coupled Operations . . . 136
5.4 Limitations of Automating Model Migration . . . 140
5.4.1 Considering Semantics of Modeling Languages . . . 140
5.4.2 Characterizing Model-Specific Migration . . . 144
5.4.3 Coping with Model-Specific Migration . . . 146
5.5 Summary . . . 148

**6 Tool Support** **149**
6.1 Recording the Coupled Evolution . . . 149
6.1.1 Tool Workflow . . . 150
6.1.2 User Interface . . . 151
6.2 Maintaining the Coupled Evolution . . . 152
6.2.1 Inspecting the Coupled Evolution . . . 152
6.2.2 Refactoring the Coupled Evolution . . . 154
6.2.3 Recovering the Coupled Evolution . . . 157
6.3 Operation-based Metamodel Versioning . . . 158
6.3.1 History Metamodel . . . 158
6.3.2 Recording and Interpreting the History . . . 162
6.3.3 Preserving the History . . . 164
6.4 Summary . . . 166

**7 Case Studies** **167**
7.1 GMF Generator Model and Palladio Component Model . . . 168
7.1.1 Study Goal . . . 168
7.1.2 Study Object . . . 169
7.1.3 Study Execution . . . 170
7.1.4 Study Result . . . 171
7.1.5 Study Discussion . . . 172
7.1.6 Threats to Validity . . . 174
7.2 Graphical Modeling Framework . . . 175
7.2.1 Study Goal . . . 175
7.2.2 Study Object . . . 176
7.2.3 Study Execution . . . 178
7.2.4 Study Result . . . 182

7.2.5 Study Discussion . . . . . . . . . 187
7.2.6 Threats to Validity . . . . . . . . . 188
7.3 Quamoco Quality Metamodel . . . . . . . . . 189
7.3.1 Study Goal . . . . . . . . . 189
7.3.2 Study Object . . . . . . . . . 189
7.3.3 Study Execution . . . . . . . . . 190
7.3.4 Study Result . . . . . . . . . 191
7.3.5 Study Discussion . . . . . . . . . 195
7.3.6 Threats to Validity . . . . . . . . . 196
7.4 Unicase Unified Model . . . . . . . . . 197
7.4.1 Study Goal . . . . . . . . . 197
7.4.2 Study Object . . . . . . . . . 197
7.4.3 Study Execution . . . . . . . . . 199
7.4.4 Study Result . . . . . . . . . 200
7.4.5 Study Discussion . . . . . . . . . 202
7.4.6 Threats to Validity . . . . . . . . . 203
7.5 Transformation Tool Contest . . . . . . . . . 204
7.5.1 Study Goal . . . . . . . . . 204
7.5.2 Study Object . . . . . . . . . 205
7.5.3 Study Execution . . . . . . . . . 205
7.5.4 Study Result . . . . . . . . . 208
7.5.5 Study Discussion . . . . . . . . . 213
7.5.6 Threats to Validity . . . . . . . . . 214
7.6 Comparison of Model Migration Tools . . . . . . . . . 215
7.6.1 Study Goal . . . . . . . . . 215
7.6.2 Study Object . . . . . . . . . 215
7.6.3 Study Execution . . . . . . . . . 218
7.6.4 Study Result . . . . . . . . . 219
7.6.5 Study Discussion . . . . . . . . . 225
7.6.6 Threats to Validity . . . . . . . . . 227
7.7 Summary . . . . . . . . . 228

**8 Beyond Model Migration: Evolutionary Metamodeling 231**
8.1 The Process of Evolutionary Metamodeling . . . . . . . . . 232
8.1.1 Elicit Metamodel Changes . . . . . . . . . 233
8.1.2 Implement Metamodel Changes . . . . . . . . . 233
8.1.3 Migrate dependent Artifacts . . . . . . . . . 234
8.1.4 Verify Model Migration . . . . . . . . . 235
8.1.5 Release Modeling Language . . . . . . . . . 235
8.2 Metamodel Usage Analysis for Identifying Metamodel Improvements 236
8.2.1 Templates for defining Usage Analyses . . . . . . . . . 236
8.2.2 Towards a Catalog of Usage Analyses . . . . . . . . . 238
8.2.3 Prototypical Implementation . . . . . . . . . 242
8.2.4 Study Goal . . . . . . . . . 242
8.2.5 Study Execution . . . . . . . . . 242
8.2.6 Study Object . . . . . . . . . 243
8.2.7 Study Result . . . . . . . . . 245
8.2.8 Study Discussion . . . . . . . . . 252

8.2.9 Threats to Validity . . . . . . . . 252
8.2.10 State of the Art . . . . . . . . 253
8.3 Towards Semantics-Preserving Model Migration . . . . . . . . 254
8.3.1 Adaptation of the Semantics Definition . . . . . . . . 254
8.3.2 Ensuring Semantics Preservation . . . . . . . . 256
8.3.3 Case Study . . . . . . . . 257
8.3.4 Revisiting the Library . . . . . . . . 260
8.3.5 State of the Art . . . . . . . . 262
8.4 Summary . . . . . . . . 263

**9 Summary 265**
9.1 Contributions . . . . . . . . 265
9.2 Outlook . . . . . . . . 268

**A Papers Excluded from the Survey 273**
A.1 Excluded Papers Within the Relevant Domain . . . . . . . . 273
A.2 Excluded Papers Outside the Relevant Domain . . . . . . . . 278
A.2.1 Process Evolution . . . . . . . . 278
A.2.2 Software Evolution . . . . . . . . 280
A.2.3 Ontology Evolution . . . . . . . . 281
A.2.4 Difference Calculation & Representation . . . . . . . . 281
A.2.5 Schema Matching & Integration . . . . . . . . 281

**Bibliography 283**

**Index 299**

Chapter 1

# Introduction

The topic of this thesis is an approach for the evolutionary development of the syntax of modeling languages based on metamodels.

**Contents**


**1.1 Context: Modeling Languages** . . . . . . . . . . . . . . . . . . . . . . 15
**1.2 Problem: Modeling Language Evolution** . . . . . . . . . . . . . . . . 17
**1.3 Thesis: Recording Metamodel Adaptations** . . . . . . . . . . . . . 19
**1.4 Approach: Evolutionary Metamodeling** . . . . . . . . . . . . . . . 20
**1.5 Contributions of this Thesis** . . . . . . . . . . . . . . . . . . . . . . . 21
**1.6 Outline of this Thesis** . . . . . . . . . . . . . . . . . . . . . . . . . . . 23


In Section 1.1 (*Context: Modeling Languages*), we first establish the context of our approach. We state the problem motivating the approach in Section 1.2 (*Problem: Modeling Language Evolution*). In Section 1.3 (*Thesis: Recording Metamodel Adaptations*), we formulate the central thesis of this dissertation. We present the approach for evolutionary metamodeling in Section 1.4 (*Approach: Evolutionary Metamodeling*). In Section 1.5 (*Contributions of this Thesis*), we list the major contributions of this dissertation. We finally provide an outline of this dissertation in Section 1.6 (*Outline of this Thesis*).

## 1.1 Context: Modeling Languages

Model-based development promises to raise the abstraction level of today's software development with the help of the pervasive use of models [France and Rumpe, 2007, Pretschner et al., 2007]. Ideally, models are built by using adequate modeling languages [Bézivin and Heckel, 2006] that allow their users to directly express the abstractions from their problem domain [Guizzardi, 2005]. Implementation code can be automatically generated from these models, using generators based on the modeling language [Czarnecki and Eisenecker, 2000]. Due to a higher level of abstraction compared to traditional code-based software development, model-based development promises to improve both the productivity and quality [Kelly and Tolvanen, 2007].

The development productivity can be improved, as the higher abstraction level of modeling languages allows their users to express the same piece of software with fewer constructs. The software quality can be improved, as the higher abstraction level of modeling languages allows their users and tools to better analyze the models for quality issues. It is a challenge to determine the adequate abstractions and hence the appropriate level of abstraction, when developing a modeling language [France and Rumpe, 2007].

Recent approaches such as Model-Driven Architecture [Kleppe et al., 2003], Software Factories [Greenfield et al., 2004] and Domain-Specific Modeling [Kelly and Tolvanen, 2007] advocate to also define modeling languages in a model-based manner. Figure 1.1 illustrates the typical process of development and use of a modeling language [Kleppe, 2008]. Language engineers build a model of the syntax of the modeling language—a so-called metamodel—which is in the center of the definition of a modeling language. The metamodel defines the constructs that the modeling language provides as well as how to compose them to models. Based on the metamodel, the language engineers build editors and code generators to support the use of the modeling language. Using these tools, the language users can build models that conform to the metamodel, i.e. obey the syntactical rules defined by the metamodel. Language workbenches [Fowler, 2005] such as the Eclipse Modeling Framework (EMF) [Steinberg et al., 2009], Microsoft DSL Tools [Cook et al., 2007] and MetaCase MetaEdit+ [Kelly and Tolvanen, 2007] significantly reduce the effort to build tool support for modeling languages around the metamodels.

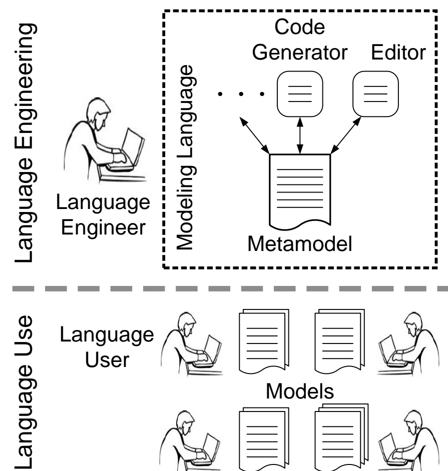



**Figure 1.1:** Development of modeling languages

In response, modeling languages are receiving increased attention in industry. The UML standard [Object Management Group, 2009], for instance, defines a general-purpose modeling language to specify object-oriented designs which is widely applied in industry. Since general-purpose modeling languages can be applied to design a wide range of software, they often do not effectively increase the abstraction level [Kelly and Tolvanen, 2007]. To increase the abstraction level, UML provides mechanisms like UML Profiles to extend the modeling language with domain-specific constructs [Selic, 2007]. Whereas UML profiles allow their users to reuse existing tool support for UML, a more clean way is to define domain-specific modeling languages from scratch. The AUTOSAR stan-

dard [AUTOSAR Development Partnership, 2008], for instance, defines a domain-specific modeling language to specify automotive software architectures. Since domain-specific modeling languages can only be applied to design software of a certain domain, they have the potential to significantly increase the abstraction level [van Deursen et al., 2000]. With the integration of modeling languages into industrial development practice, their evolution is gaining importance. Although significant work in both academia and industry has been invested into tool support for the initial development of modeling languages, issues related to their evolution are still largely disregarded [Mens and Demeyer, 2008].

## 1.2 Problem: Modeling Language Evolution

During the initial development of a modeling language, it is often a challenge to determine the appropriate level of abstraction [France and Rumpe, 2007]. If the abstraction level is too low, the modeling language may not lead to a significant gain in productivity and quality. If the abstraction level is too high, the modeling language may not be expressive enough to specify all required aspects of the software. Therefore, even though often neglected, a modeling language is subject to change like any other software artifact [Favre, 2005]. This holds for both general-purpose and domain-specific modeling languages. For instance, the general-purpose modeling language UML [Object Management Group, 2009] already has a rich evolution history, although being relatively young. Domain-specific modeling languages like AUTOSAR [AUTOSAR Development Partnership, 2008] are even more prone to change, as they have to be adapted, whenever their domain changes due to technological progress or evolving requirements [Sprinkle, 2003].

Figure 1.2 illustrates the typical process of evolving a modeling language [Kleppe, 2008]. Suppose that the language engineers have built a first version of the modeling language by defining a metamodel and creating editors and code generators around the metamodel. When the language users employ the editors to build models conforming to the metamodel, they often identify new requirements for the modeling language. The language engineers evolve the modeling language to a second version by first adapting its metamodel to the additional requirements. Metamodel adaptation may invalidate existing artifacts like editors and code generators that depend on the metamodel [Sprinkle, 2003]. Most importantly, existing models built by the language users may no longer conform to the adapted metamodels. The existing artifacts need to be migrated to conform to the metamodel again, so that they can be used with the evolved modeling language. In this thesis, we focus on the migration of models which is probably the most challenging, since models typically outnumber the other artifacts by far and are usually not under control of the language engineers.

In current practice, the migration of models is often performed manually which is tedious and error-prone. Consequently, missing tool support for modeling language evolution heavily hampers cost-efficient model-based development in practice. Providing appropriate tool support for model migration has been identified as one of the central challenges of software evolution [Mens et al., 2005]. There are two major challenges for building tools to support the migration of models in response to

metamodel adaptation.

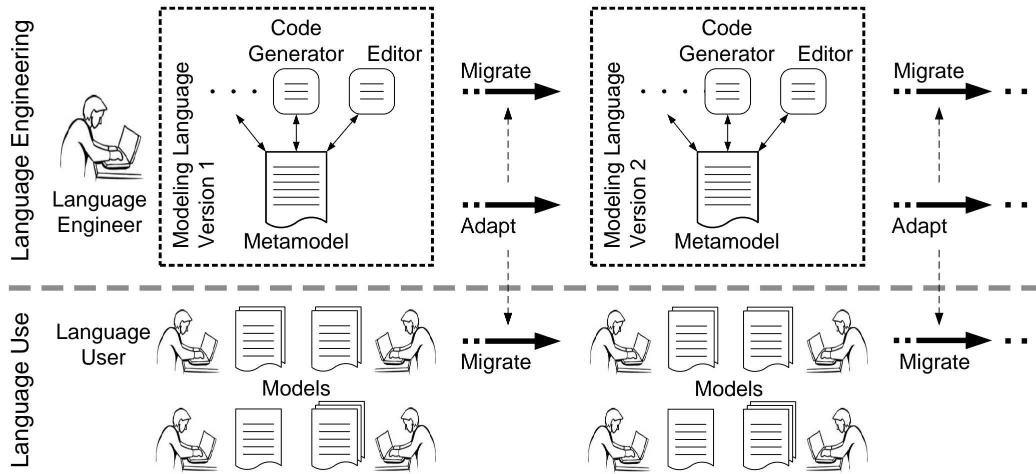



**Figure 1.2:** Evolution of modeling languages

**Automation Challenge.** The first challenge for creating tool support is to automate the migration of existing models as far as possible [Klint et al., 2005]. Due to the enormous effort for model migration, modeling language evolution is often performed in a backwards-compatible fashion. In other words, the language engineers adapt the metamodel in a way that the existing models can still be used with the evolved modeling language without migration. However, backwards compatibility heavily constrains the way in which a metamodel can be adapted. Furthermore, the preservation of old constructs can unnecessarily clutter and complicate a metamodel [Meyer, 2000]. This approach can be further refined by using deprecation to signal metamodel changes which is known from API evolution [Dig and Johnson, 2006]. More precisely, language engineers mark constructs as deprecated, before they actually remove them from the metamodel. Language users are then informed about the deprecated constructs which should no longer be used. However, deprecation shifts the responsibility for model migration from the language engineer to the language users. In addition, deprecation also clutters and complicates the metamodel, as it leads to non-orthogonal constructs being available at the same time. A prominent example is the introduction of generics to Java, where backwards compatibility is achieved by erasing all information about generic types during compilation. However, this technique leads to several limitations and exceptions in applying generic types, as different types are treated uniformly at runtime by the generics-unaware Java virtual machine [Allen and Cartwright, 2002]. In a nutshell, both backwards compatibility and deprecation heavily threaten the simplicity and quality of the metamodel. As a lot of artifacts like editors and code generators depend on the metamodel, these approaches also affect their simplicity and quality. **To avoid decay of modeling languages due to non-backwards-compatible evolution, the migration of existing models should be automated as far as possible.**

**Semantics Preservation Challenge.** The second challenge for creating tool support is to ensure that the migration preserves the meaning of the models as far as possible [Sprinkle and Karsai, 2004]. In practice, it is often difficult to prove semantics

preservation, since the semantics of a modeling language is often not defined explicitly, but only implicitly by e.g. code generators. A popular approach to automate the migration of existing models is to first perform all metamodel adaptations for the new version of the modeling language and later manually implement a migrator. The purpose of the migrator is to preserve the information of an existing model by transforming it into a new version that conforms to the adapted metamodel. This approach has the advantage that the metamodel can be adapted in a clean manner, because legacy constructs can be removed. However, implementation of a migrator after a number of metamodel adaptations is tedious, as the intention behind these metamodel adaptations is already lost. This missing intention makes it difficult for the language engineers to ensure that the migrator preserves the meaning of a possibly unknown number of models. The cable length problem [Steinke, 2006] which emerged during the development of the Airbus A380 illustrates that the loss of information during model migration can lead to high costs. For designing the Airbus A380, different groups of language users were employing different versions of the 3D modeling tool Catia [Dassault Systèmes, 2010]. To ensure model exchange between the two versions of the Catia modeling language, the language engineers implemented an automatic migrator. However, the migrator did not correctly migrate the information concerning the cables which resulted in cables being too short. All in all, the cable problem led to 2.8 billion Euros additional costs which is quite high compared to the overall design cost for the Airbus A380 of 12 billion Euros. **Semantics preservation is an important property to ensure that meaningful information is not lost during model migration.**

> **Problem Statement:**
> Modeling language evolution is often avoided in practice, since there is no adequate tool support to automate the migration of existing models and to ensure the preservation of their semantics during migration.

## 1.3 Thesis: Recording Metamodel Adaptations

In practice, a modeling language is evolved by incremental adaptations to the metamodel. There are a number of primitive metamodel changes like create element, rename element, delete element, and so on. One or more such primitive changes compose a well-defined metamodel adaptation, which preserves the overall consistency of the metamodel. Usually, these well-defined adaptations imply a certain intention about how to migrate existing models. In current practice, however, the occurrence of well-defined metamodel adaptations is lost when adapting a metamodel.

> **Thesis:**
> By recording the metamodel adaptations throughout the evolution of the modeling language, we can automate the migration of existing models and ensure the preservation of their semantics during migration.

## 1.4 Approach: Evolutionary Metamodeling

Figure 1.3 illustrates the integrated approach COPE for the evolutionary development of modeling languages which we developed to address the challenges. We explicitly model these metamodel adaptations as operations performed on the metamodel and record them in a history model of the metamodel. An operation can be characterized in terms of how to migrate existing models in response to the encapsulated metamodel adaptation. In this thesis, such an operation is called a coupled operation, as it couples the metamodel adaptation with the model migration. Consequently, a coupled operation also preserves the information of the models, as it provides an appropriate migration. Coupled operations can be easily composed by simply sequencing them. They are modular in the sense that the corresponding migration can be specified independently of any neighboring coupled operation. Due to their modularity, a comprehensive evolution can be decomposed into manageable coupled operations, thus ensuring scalability. The resulting history model further serves as a documentation of the evolution of the modeling language and can be used to later understand it.

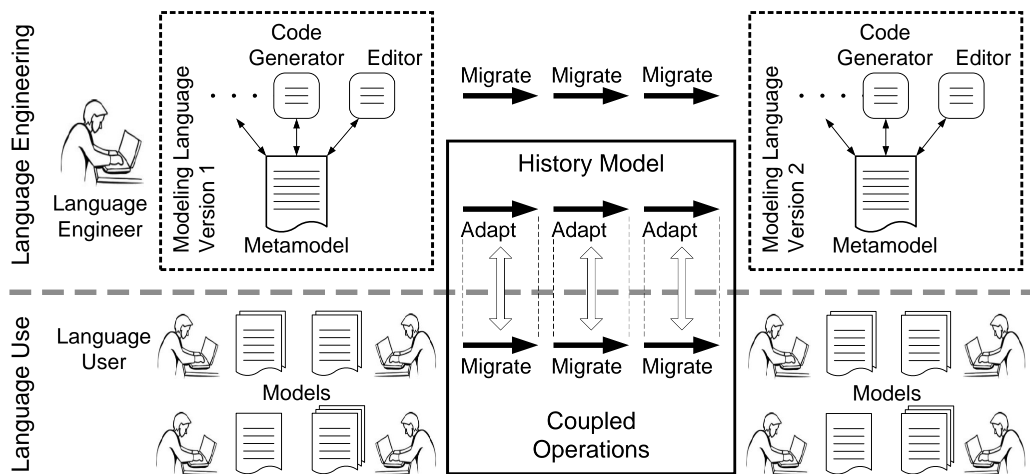



**Figure 1.3:** Evolutionary metamodeling

**Providing Automation.** The first challenge of automating the migration of existing models is addressed by different kinds of coupled operations. Coupled operations allow the language engineers to attach a model migration to the metamodel adaptation during modeling language evolution. The attached information can later be used to automatically migrate existing models so that they conform to the metamodel again. The migration can be further automated by reusing coupled operations that encapsulate recurring migrations. More specifically, our approach thus distinguishes between two basic kinds of coupled operations: reusable and custom coupled operations. Reusable coupled operations allow the language engineers to reuse migrations across metamodels, thereby further automating modeling language evolution. Reusable coupled operations are organized in a library through which they are made available to the language engineers. Custom coupled operations allow the language engineers to express complex migrations which cannot be reused across metamodels. A custom coupled operation can thus be used, in case no reusable coupled operation

captures the required model migration.

**Ensuring Semantics Preservation.** The second challenge of building a semantics-preserving migration is addressed by recording the evolution in the history model. Building an automated migrator after all the metamodel adaptations have been performed is a non-trivial task, as it has to ensure the preservation of the meaning of a possibly unknown set of models. This task is further complicated by the issue that, in current practice, the intention behind the metamodel changes is lost in the evolution process. To not lose the intention behind the adaptation, our approach immediately records the coupled operations in the history model, when they are performed to adapt the metamodel. Thereby, the approach tries to constructively ensure that the recorded history model provides a semantics-preserving migration. However, the approach requires the language engineers to choose the appropriate coupled operations, i.e. those preserving the semantics. We believe that the language engineer is the right stakeholder for this job, as he or she is also the one defining the semantics of the modeling language—either explicitly by basing the language on a semantic theory, or implicitly by building a code generator. In addition, proving semantics preservation is eased by decomposing the evolution into modular coupled operations and proving it for each coupled operation.

## 1.5 Contributions of this Thesis

This thesis develops an evolutionary method to automate and secure modeling language evolution. This method records the adaptations to the metamodel as coupled operations in a history model which can later be used to automatically migrate models conforming to the metamodel. This thesis presents the following contributions to the current state of the art. Where applicable, we cite previously published material.

**Automatability of Model Migration in Practice.** Unfortunately, little is known about the potential for automating the model migration in practice. To be able to characterize coupled operations according to their automatability, we developed a classification of coupled operations. To quantify the potential for automation, we have performed an empirical study that applies the classification to the evolution of two industrial metamodels [Herrmannsdoerfer et al., 2008a]. From this empirical study, we derive requirements for an effective approach to automate modeling language evolution.

**Cross-Space Survey on Coupled Evolution.** The literature already provides a number of approaches to ease migration in response to modeling language evolution. Additionally, there are related syntax specification formalisms—like database schemas or grammars—which are subject to the same problem. We performed a systematic literature review to identify approaches that are related to model migration in the different technical spaces. We analyze these existing approaches with respect to the requirements derived from the case study. From the issues of existing approaches, we motivate our approach to automate model migration.

**Method for Evolutionary Metamodeling.** Based on the requirements, we developed the method COPE to support the evolution of modeling languages based on metamodels [Herrmannsdoerfer et al., 2009a]. This method models the evolution as a sequence of coupled operations which have been performed on the metamodel. To provide automation, the method provides different kinds of coupled operations for different levels of automation. To ensure semantics preservation, the method advocates to record the coupled operations already when they are performed.

**Language to Encode Coupled Operations.** We have developed a metaprogramming language to ease the implementation of coupled operations [Herrmannsdoerfer et al., 2008b]. The language softens the conformance of models to the metamodel during migration to ease the specification of model migrations. To ensure conformance after performing an operation, the language includes a transaction mechanism. In addition, the language provides an abstraction mechanism to reuse recurring coupled operations across metamodels.

**Library of Reusable Coupled Operations.** Most automation is provided by reusable coupled operations which encapsulate the adaptation and migration in a metamodel-independent way. To benefit from this automation, we need an extensive set of reusable coupled operations which covers a wide variety of evolution scenarios. Using the language, we have therefore built a library of reusable coupled operations [Herrmannsdoerfer et al., 2010b]. We classify these operations according to a number of properties known from the literature.

**Limitations of Automating Model Migration.** However, the evolution of modeling languages occasionally leads to metamodel changes for which the migration of models inherently cannot be fully automated. In these cases, the model migration requires information which is not available in the model. We formally characterize metamodel adaptations that prevent the automatic migration of models and outline different possibilities to cope with these kinds of metamodel changes [Herrmannsdoerfer and Ratiu, 2009, Herrmannsdoerfer and Ratiu, 2010].

**Metamodel to Record Coupled Operations.** To express the history model for a metamodel, we have developed a versioning metamodel [Herrmannsdoerfer, 2009]. This metamodel is expressive enough to record both the reusable and custom coupled operations to the history model, when they occur. A migrator for the batch migration of models that is specified in the migration language can be automatically generated from the history model.

**Evaluation through Case Studies.** To be able to apply the method in practice, we have implemented a tool for COPE based on the widely used Eclipse Modeling Framework (EMF) [Herrmannsdoerfer, 2011]. Using this tool, we have evaluated the method by means of six case studies. The case studies can be classified into the following categories: First, we have applied COPE to reverse engineer the history model for the metamodels of the Graphical Modeling Framework (GMF) [Herrmannsdoerfer et al., 2009c] and the Palladio Component Model (PCM) [Herrmannsdoerfer et al., 2009a]. Second, we have applied COPE to for-

ward engineer the history model for the metamodels of the Quamoco Quality Model and Unicase. Third, we have compared COPE to other model transformation and migration tools by participating in the Transformation Tool Contest (TTC) [Herrmannsdoerfer, 2010] and by performing a case study with the authors of other tools [Rose et al., 2010a].

**Metamodel Usage Analysis for Identifying Metamodel Improvements.** While model migration propagates metamodel changes to models, metamodel usage analysis identifies metamodel changes by analyzing models built with the metamodel. For instance, if certain metamodel elements are not used in models, we might be able to remove these elements from the metamodel. We have developed a method that derives usage expectations from the metamodel and compares them to the actual usage in models [Herrmannsdoerfer et al., 2010a]. Coupled operations can be proposed to remove deviations between actual and expected usage. To analyze whether we are really able to identify metamodel improvements, we have performed an empirical study on a large corpus of metamodels and models.

**Semantics-Preserving Model Migration.** When the metamodel of a modeling language is adapted, we may also need to migrate the semantics definition of the language, as it depends on the metamodel. If both the models and the semantics definition are migrated consistently, we can ensure semantics preservation in a constructive manner. To demonstrate the viability of the approach, we performed a simple case study [Herrmannsdoerfer and Koegel, 2010b]. Moreover, we discuss how the reusable coupled operations in the library can be extended with an adaptation of the semantics definition.

## 1.6 Outline of this Thesis

Figure 1.4 displays the contributions and chapters of this thesis, indicating which contributions are explained in which chapter.

Chapter 2 (*Background: Engineering of Modeling Languages*) introduces the terms and concepts needed to understand the following chapters. It gives an overview of the metamodel-based definition, implementation and evolution of modeling languages.

Chapter 3 (*State of the Practice: Automatability of Model Migration*) examines the state of the practice through an empirical study. The goal of the empirical study is to analyze the automatability of model migration in practice and to derive requirements for an approach.

Chapter 4 (*State of the Art: A Cross-Space Survey on Coupled Evolution*) presents the features of existing approaches from different technical spaces. It motivates the need for our approach by analyzing the existing approaches with respect to the requirements.

Chapter 5 (*COPE – Coupled Evolution of Metamodels and Models*) introduces our method COPE to automate the coupled evolution of metamodels and models. The method is based on a language to encode coupled operations and on a library of reusable coupled operations. The chapter also discusses the limitations of automating model migration.

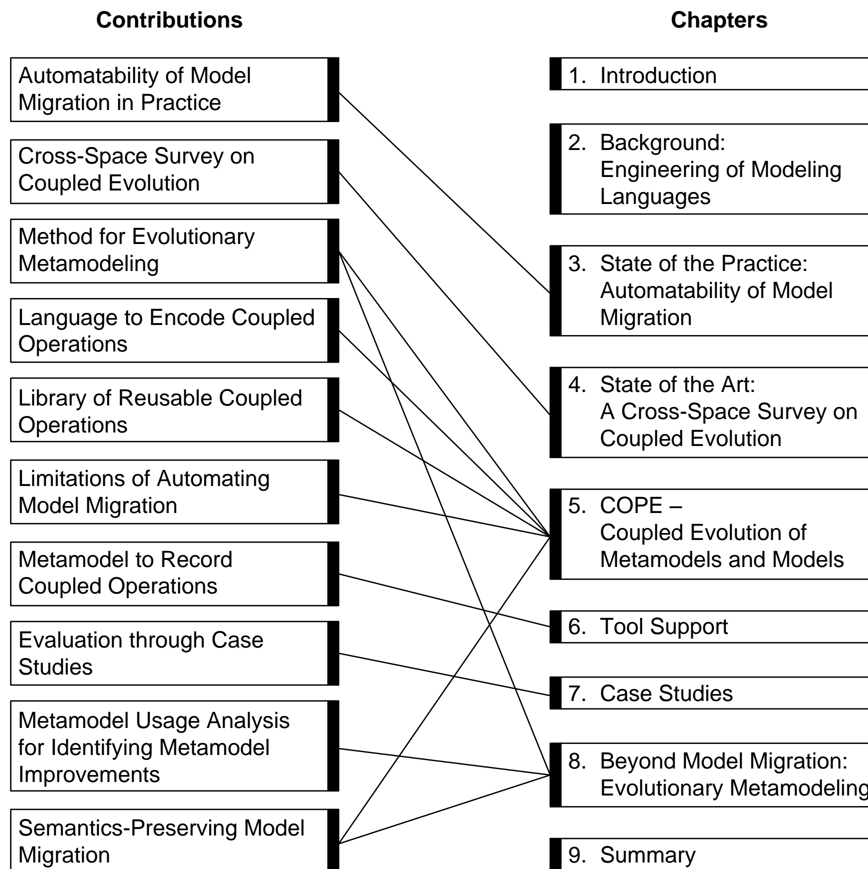



**Figure 1.4:** Structure of this thesis

Chapter 6 (*Tool Support*) presents the tool support to record the coupled evolution in a history model. The history model is based on a metamodel to record the operations performed on metamodels.

Chapter 7 (*Case Studies*) summarizes the results of the six case studies that evaluate the method and tool. It presents each case study using the typical structure of empirical studies: goal, object, execution, result, discussion and threats to validity.

Chapter 8 (*Beyond Model Migration: Evolutionary Metamodeling*) extends COPE to an evolutionary method for developing modeling languages. It presents a method to identify metamodel improvements by analyzing models as well as a method to automatically adapt the semantics definition when adapting the metamodel.

Chapter 9 (*Summary*) summarizes this thesis by presenting its contributions, their limitations and directions for future work.

Chapter 2

# Background: Engineering of Modeling Languages

This chapter provides a summary of the terms and concepts from existing literature that are needed to understand the following chapters: the definition, implementation and evolution of modeling languages based on metamodels. We use the running example of a dialect of state machines to illustrate the terms and concepts.

**Contents**


**2.1 Model-based Development** . . . . . . . . . . . . . . . . . . . . . . . . 25
**2.2 Metamodeling – Modeling the Abstract Syntax of Modeling Languages** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 29
**2.3 Concrete Syntax of Modeling Languages** . . . . . . . . . . . . . . . 45
**2.4 Semantics of Modeling Languages** . . . . . . . . . . . . . . . . . . . 48
**2.5 Evolution of Modeling Languages** . . . . . . . . . . . . . . . . . . . . 53
**2.6 Summary** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 64


Section 2.1 (*Model-based Development*) discusses model-based development of software systems. Section 2.2 (*Metamodeling – Modeling the Abstract Syntax of Modeling Languages*) introduces metamodels as means to define the abstract syntax of modeling languages. Section 2.3 (*Concrete Syntax of Modeling Languages*) explains how the concrete syntax can be defined based on the abstract syntax. Section 2.4 (*Semantics of Modeling Languages*) introduces the semantics of modeling languages and how it is defined in practice. Section 2.5 (*Evolution of Modeling Languages*) analyzes the causes and implications of the evolution of modeling languages. We summarize this chapter in Section 2.6 (*Summary*).

## 2.1 Model-based Development

Due to advances in technology, software systems are becoming more and more complex. Traditional code-based development fails to keep up with the increasing complexity, leading to more and more errors. To master the increasing complexity, model-

based development raises the abstraction level by creating models as abstractions of the software system.

### 2.1.1 Models and Modeling Languages

Models reduce the complexity of software development by abstracting away recurring implementation details. Implementation code—including these details—can then be automatically generated from these models. According to [Stachowiak, 1973], a model is an abstract description of an original—e.g. a software system—for a certain purpose.

We can think of different purposes for which models of a software system are created. First, models can be used only for documenting a software system so that it is easier to understand its architecture. Second, models can be used for testing by generating test cases from the models that provide an abstract description of the system derived from the requirements. Third, models can be used for simulation and verification to identify errors or validate the models against the system requirements. Fourth, models can be used for the generation of code implementing the system. This enumeration is not complete, i.e. we can think of various other purposes for which models are created.

**Example 2.1** (Model)**.** *Throughout this chapter, we use simple state machine models that describe the behavior of a software system as a running example. As an example model, we specify the simplified behavior of a controller for a pedestrian traffic light. Figure 2.1 depicts this model as a state transition diagram. When the traffic light is* red *and a pedestrian* request*s a green phase, the controller transitions to* wait *and activates a timer (*startTimer*). When the* timeOut *of the timer occurs, the controller transitions to* green *and activates the timer again. When this* timeOut *occurs, the controller returns to state* red*.*

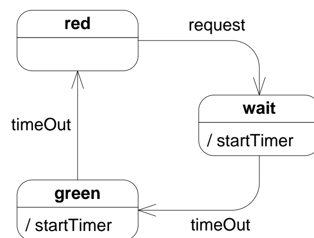



**Figure 2.1:** Example state machine model of the behavior of a pedestrian traffic light

A language is required to be able to specify state machine models of a software system like the pedestrian traffic light. Such a language is called a modeling language [Kleppe et al., 2003]. A modeling language is a well-defined language to specify certain kinds of models. A well-defined language is a language with well-defined form (syntax) and meaning (semantics) which is suitable for automated interpretation by a computer. Consequently, a modeling language defines how to write down a model, and how to assign the model a meaning. In Section 2.2 (*Metamodeling – Modeling the Abstract Syntax of Modeling Languages*), Section 2.3 (*Concrete Syntax of Modeling Languages*) and Section 2.4 (*Semantics of Modeling Languages*), we define in more detail the constituents of which a modeling language consists.

**Example 2.2** (Modeling Language)**.** *To describe the behavior of the pedestrian traffic light, we use a simple modeling language to specify state machines. The syntax of this modeling language allows language users to define states of the software system and transitions between the states. The semantics of this modeling language is that of Moore machines [Moore, 1956], i.e. a transition is activated by a trigger, and a state produces an effect.*

### 2.1.2 Benefits and Risks

Compared to code-based development of software systems, model-based development has certain benefits. The benefits of model-based development mainly stem from increasing the abstraction level at which software systems are developed. However, the higher abstraction level also leads to a number of risks.

**Benefits.** First, the higher abstraction level increases the productivity with which software systems are developed [Weiss and Lai, 1999]. The productivity increases, because the same functionality can be expressed with fewer constructs using abstractions. One goal is to automatically generate the code implementing the software system from the models. Moreover, abstractions of a software system are much easier to understand and hence to maintain than its code.

Second, the higher abstraction level increases the quality of the software systems which are developed as well as their specifications [Kieburtz et al., 1996]. The quality increases, because smaller, more abstract models are easier to analyze by both language users and tools. Certain validation rules can even be directly integrated into the modeling language, thereby identifying errors early on. Moreover, automatic generation of code from the models ensures that the implementation of the software system conforms to its models.

**Risks.** First, the higher abstraction level restricts the number of software systems to which model-based development can be applied [Kelly and Tolvanen, 2007]. The applicability is restricted, because a modeling language usually provides abstractions that fit a certain kind of software system. Consequently, these modeling languages can only be applied to specify models for this kind of system. The more restricted the number of software systems, the higher the abstractions that can be provided by the modeling language.

Second, effort is required to build tools that support the application of the modeling language [Spinellis, 2001]. To be able to develop models in the modeling language, an editor needs to be built including the validation rules. Effort is also required to build a code generator that is verified to preserve the meaning of the models. But not only editing and code generation are important, but also other aspects like distributed modeling which calls for tool support specialized to the modeling language. The effort required for building tool support for a modeling language should not exceed the productivity and quality gain provided by the modeling language.

### 2.1.3 The Quest for Abstraction

The level of abstraction that a modeling language provides is crucial for the advantages of model-based development [France and Rumpe, 2007]. Consequently, it is important to find the abstractions that are appropriate for the kinds of software system that are targeted by the modeling language as well as for the purpose of the modeling language.

**If the Abstraction Level is Too Low,** the modeling language may not lead to a significant gain in productivity and quality. Consequently, it may not be worth to build tool support for such a modeling language. The abstraction level of general-purpose modeling languages which are applicable to a wide variety of software systems is usually too low. However, their applicability to a wide variety of software systems allows their language engineers to amortize the effort for building tool support. In a nutshell, a modeling language should leave out the details that are not required for the purpose of the modeling language.

**If the Abstraction Level is Too High,** the modeling language may not be expressive enough to specify all required aspects of the software system. Consequently, the modeling language might only be applicable to build a very restricted kind of software system. The abstraction level of domain-specific modeling languages which are applicable to a restricted set of software systems is usually too high. In a nutshell, a modeling language should not leave out too many details that make it incomplete for developing different kinds of software systems.

### 2.1.4 Major Initiatives

There are various initiatives that refine the general idea of model-based development. The most prominent initiatives are Domain-Specific Modeling (DSM), Model-Driven Architecture (MDA) and Software Factories.

**Domain-Specific Modeling** (DSM) [Kelly and Tolvanen, 2007] advocates to build a modeling language tailored to the needs of an organization. Furthermore, a code generator needs to be built to fully generate the implementation code from the models built with the modeling language. DSM promises to significantly increase productivity due to the higher level of abstraction at which models are built compared to the code. Domain-specific modeling languages are easier to learn than general-purpose modeling languages like UML [Object Management Group, 2009], since the language engineers can choose the abstractions with which the language users are familiar. As only the requirements of one organization need to be taken into account, domain-specific modeling languages are easier to develop and maintain than general-purpose modeling languages. However, appropriate tool support is required to ease the development and maintenance of domain-specific modeling languages.

**Model-Driven Architecture** (MDA) [Kleppe et al., 2003] is a standard from the Object Management Group (OMG) [Object Management Group, 2003] which proposes to build models for software systems along different levels of abstraction. A

Platform-Independent Model (PIM) defines the system independently of a specific software or hardware platform. The PIM can be transformed into a Platform-Specific Model (PSM) which takes the software or hardware platform into account. The separation between PIM and PSM allows a PIM to be transformed to PSMs for different platforms. Finally, a PSM is transformed to implementation code for the corresponding platform. The OMG provides related standards like Query View Transformations (QVT) [Object Management Group, 2008b] to define model transformations or Meta Object Facility (MOF) [Object Management Group, 2006a] to define the modeling languages for PIM and PSM.

**Software Factories** [Greenfield et al., 2004] are product lines that allow the developers to quickly assemble software systems from a certain domain. The product line defines the commonalities and differences between these software systems as a modeling language. A software factory consists of three constituents: The software factory schema defines the components for building a product as well as how they can be composed. The software factory template provides the implementation of the components, which may be patterns, templates, frameworks, and so on. The extensible development environment can be configured with the software factory schema and template to provide tool support for building products. A product is thus a model that is defined with the software factory schema as modeling language and from which the implementation can be derived using the software factory template.

## 2.2 Metamodeling – Modeling the Abstract Syntax of Modeling Languages

According to [Harel and Rumpe, 2004] and [Chen et al., 2005], a modeling language—like any formal language—is completely defined through its abstract syntax, concrete syntax and semantics. The abstract syntax defines the set of valid models, the concrete syntax the textual, diagrammatic or tabular representation of a model, and the semantics the meaning of a model defined with the modeling language.

We put the abstract syntax in the center of the definition of a modeling language [Kleppe, 2008]. This is different from the traditional design of programming languages, where the concrete syntax is in the center of language definition [Aho et al., 1986]. Consequently, both concrete syntax and semantics are defined based on the abstract syntax. This enables the language engineer to define several concrete syntaxes as well as several semantics for a single abstract syntax. Furthermore, different related modeling languages are best integrated in terms of their abstract syntax [Braun, 2003, Braun, 2004].

In the following, we first illustrate the abstract syntax of a modeling language, before illustrating the concrete syntax and semantics in Section 2.3 (*Concrete Syntax of Modeling Languages*) and Section 2.4 (*Semantics of Modeling Languages*).

### 2.2.1 Meta Object Facility

There are a number of existing languages for defining abstract syntax like the Meta Object Facility (MOF) [Object Management Group, 2006a] from OMG, Kernel Metametamodel (KM3) [Jouault and Bézivin, 2006] from INRIA, Graph Object Property Relationship Role (GOPRR) [Kelly and Tolvanen, 2007] from Meta Case, MetaGME [Ledeczi et al., 2001] from the Generic Modeling Environment (GME), or the one provided by Microsoft DSL tools [Cook et al., 2007].

In this thesis, we use the Essential Meta Object Facility (E-MOF)—which is standardized by the OMG [Object Management Group, 2006a]—as a language to define the abstract syntax of a modeling language. We have chosen MOF for several reasons. First, MOF is standardized and thus not proprietary to a certain metamodeling tool like GOPPR, MetaGME and DSL Tools. Second, all the metamodeling languages are quite similar, representing models as graphs, only with differences in the metamodeling constructs they provide. Third, MOF is probably the most widely applied metamodeling language. Besides E-MOF which provides basic metamodeling constructs, Complete MOF (C-MOF) provides more advanced metamodeling constructs. However, C-MOF is hardly applied in practice due to its complexity.

E-MOF is based on the object-oriented paradigm and defines a hierarchy of models which are grouped into layers. This hierarchy is called meta hierarchy [Bézivin, 2005]. Figure 2.2 illustrates the layers of the meta hierarchy for a simplified version of the abstract syntax of the state machine modeling language. The figure shows both the abstract and concrete syntax of the different models. The abstract syntax on all layers is represented by UML object diagrams [Object Management Group, 2009] in which nodes are called objects and edges are called links. The concrete syntax are state transition diagrams on the model layer and UML class diagrams [Object Management Group, 2009] on the other layers in which nodes are called classes and edges are called references.

**Model Layer.** The lowest layer contains the models that are specified using a modeling language. A model in this layer conforms to a metamodel in the next upper layer. There is an instance-type relationship between the objects in the model and the classes in the metamodel as well as between the links in the model and the references in the metamodel. In Figure 2.2, the objects and links refer to the classes and references they instantiate by name. For example, the pedestrian traffic light model instantiates the classes State and Transition as well as the references source and target from the metamodel.

**Metamodel Layer.** The middle layer contains the metamodels that define the abstract syntax of modeling languages. A metamodel in this layer conforms to the metametamodel in the next upper layer. There is an instance-type relationship between the objects in the metamodel and the classes in the metametamodel as well as between the links in the model and the references in the metamodel. For example, the state machine metamodel instantiates the classes Class and Reference as well as the references features and type from the metametamodel.

**Metametamodel Layer.** The upmost layer contains the metametamodel that defines

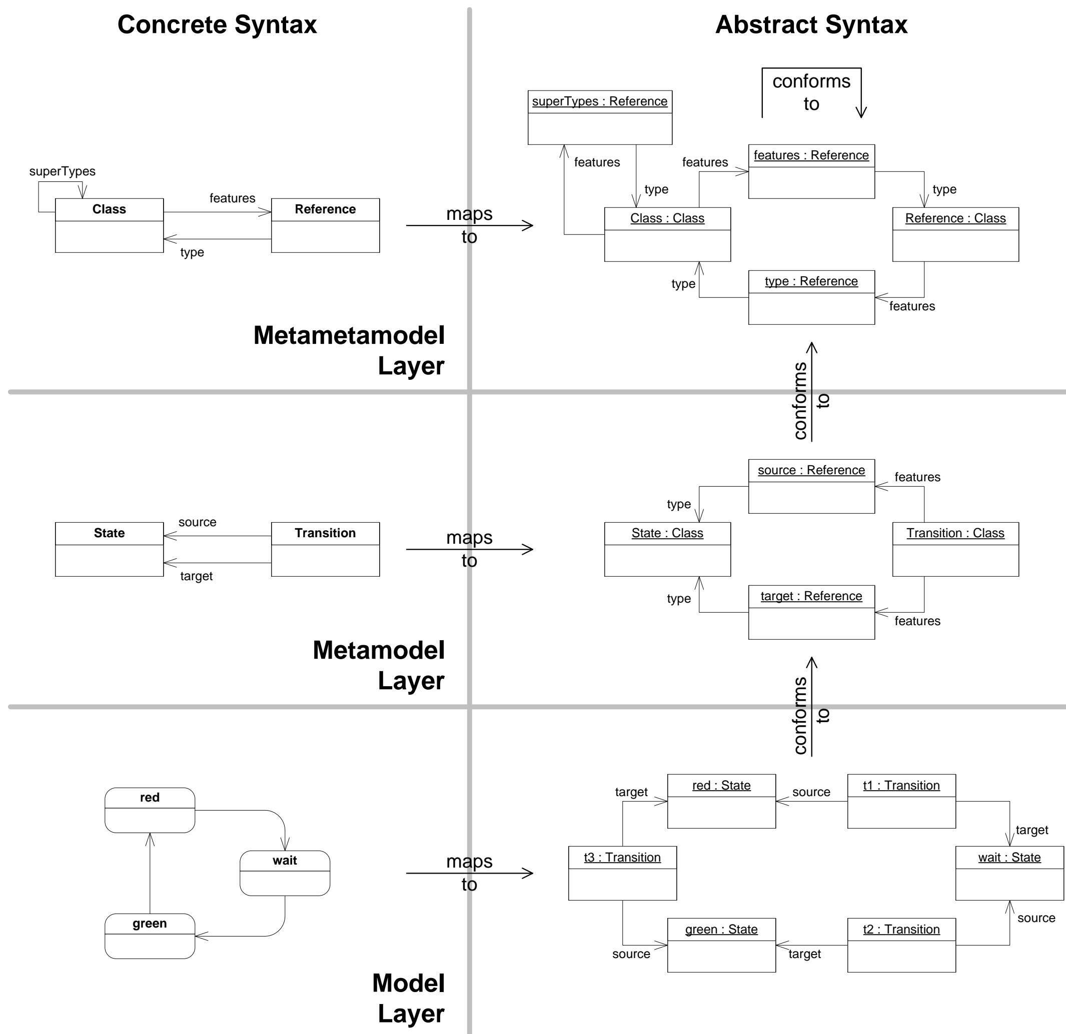



**Figure 2.2:** Meta hierarchy

the abstract syntax of the metamodeling language. The metametamodel in this layer conforms to itself, thus finishing the meta hierarchy. As a consequence, there is an instance-type relationship between the objects and links in the metametamodel and the classes and references in the metametamodel. For example, the metametamodel instantiates the classes Class and Reference as well as the references features, type and superTypes from the metametamodel.

### 2.2.2 Abstract Syntax of a Modeling Language

The abstract syntax defines the abstract, internal representation of a model. In that sense, the abstract syntax abstracts away all information which is not required to interpret a model with respect to the semantics, e.g. the concrete layouting of models, the fonts used to display text, etc. In the following, we present a formalization of the abstract syntax using graphs which is based on [Jouault and Bézivin, 2006] and [Kleppe and Rensink, 2008]. More specifically, we use directed multigraphs like [Jouault and Bézivin, 2006] and [Kleppe and Rensink, 2008]:

**Definition 2.1** (Directed Multigraph). *A directed multigraph is a tuple $G = (N, E, src, tgt)$ where*

- *$N$ is a finite set of nodes,*
- *$E$ is a finite set of edges,*
- *$src : E \to N$ is a total function that maps an edge $e \in E$ to its source node $src(e) \in N$, and*
- *$tgt : E \to N$ is a total function that maps an edge $e \in E$ to its target node $tgt(e) \in N$.*

Due to our definition, there may be multiple edges between two nodes. Note that we only consider models that are of finite size, since both the sets of nodes and edges are finite. To be able to label nodes and edges, we need identifiers: Let $\mathbb{ID}$ be an infinite set of identifiers. Like [Kleppe and Rensink, 2008], we define a model as a graph whose nodes are identifiers and whose nodes and edges are labeled by identifiers:

**Definition 2.2** (Model). *A model is a tuple $m = (N, E, src, tgt, lab)$ where*

- *$(N, E, src, tgt)$ is a directed multigraph with $N \subset \mathbb{ID}$ and*
- *$lab : N \cup E \to \mathbb{ID}$ is a function that labels nodes and edges with identifiers.*

*Let $\mathcal{M}$ be the set of all such models.*

**Example 2.3** (Model). *Figure 2.3 depicts a model as a UML object diagram [Object Management Group, 2009]. Nodes are represented by boxes and edges by arrows between boxes. The labels are shown on both nodes and edges—on nodes behind the node identifier followed by a colon.*

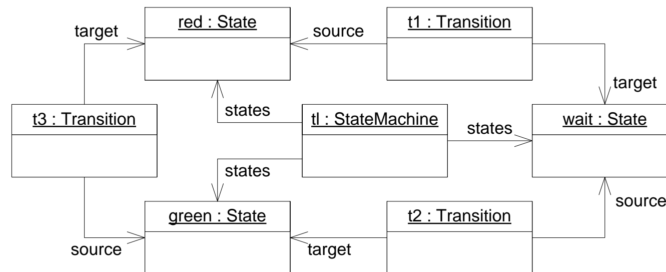



**Figure 2.3:** Example model represented as UML object diagram

To impose structural constraints on a model, the literature provides the notion of type graphs [Ehrig et al., 2008, Kleppe and Rensink, 2008]. A type graph defines types for nodes and edges in a graph as well as an inheritance relation between node types:

**Definition 2.3** (Type Graph). *A type graph is a tuple $TG = (N, E, src, tgt, inh)$ where:*

- *$N \subset \mathbb{ID}$ is a set of node types,*
- *$E \subset \mathbb{ID}$ is a set of edge types,*
- *$(N, E, src, tgt)$ is a directed multigraph, and*
- *$inh : N \times N$ is an acyclic relation expressing that some node types inherit from others.*

**Example 2.4** (Type Graph)**.** *Figure 2.4 depicts a type graph as UML class diagram. Node types are represented as boxes and edge types as arrows between boxes. The identifiers are shown for both node and edge types. Inheritance between node types is depicted as arrows with a triangle as arrow head.*

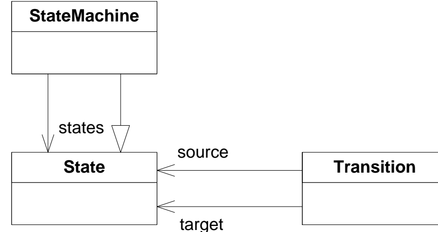



**Figure 2.4:** Example type graph represented as UML class diagram

By binding the labels of a model to the node type identifiers of a type graph, we can define an instance-type relationship between the model and the type graph. This binding can be expressed by the following helper predicates on a model $m = (N_m, E_m, src_m, tgt_m, lab)$ and a type graph $TG = (N_{TG}, E_{TG}, src_{TG}, tgt_{TG}, inh)$:

- $isInstanceOf(x, y)$. A node $x \in N_m$ or edge $x \in E_m$ is instance of node type $y \in N_{TG}$ or edge type $y \in E_{TG}$ if the label of the node or edge is the node or edge type, respectively:

$$isInstanceOf(x, y) :\Leftrightarrow lab(x) = y$$

- $isKindOf(n, t)$. A node $n \in N_m$ is called kind of node type $t \in N_{TG}$ if it is instance of a node type that inherits from node type $t$:

$$isKindOf(n, t) :\Leftrightarrow \exists s \in N_{TG} : isInstanceOf(n, s) \wedge (s, t) \in inh^*$$

  where $inh^*$ denotes the transitive closure of $inh$.

A model is correctly typed by a type graph if its labels are chosen from the types defined by the type graph, and its structure is consistent to the structure defined by the type graph modulo inheritance [Kleppe and Rensink, 2008]:

**Definition 2.4** (Typing)**.** *A model $m = (N_m, E_m, src_m, tgt_m, lab)$ is typed by type graph $TG = (N_{TG}, E_{TG}, src_{TG}, tgt_{TG}, inh)$ if the following two functions exist:*

- *a total function $nm : N_m \to N_{TG}$ respecting the instance-type relationship:*

$$\forall n \in N_m \quad : \quad isInstanceOf(n, nm(n))$$

- *a total function $em : E_m \to E_{TG}$ respecting the instance-type relationship modulo inheritance:*

$$\begin{aligned} \forall e \in E_m \quad : \quad & isInstanceOf(e, em(e)) \wedge isKindOf(src_m(e), src_{TG}(em(e))) \\ & \wedge isKindOf(src_m(e), src_{TG}(em(e))) \end{aligned}$$

*Let $\mathcal{M}_{TG}$ be the models that are typed by type graph $TG$.*

By the typing, the type graph imposes constraints on how the edges in a model can connect nodes. However, additional constraints may need to be necessary to further restrict the models. To capture these constraints, we may extend our definition of a type graph. To keep the definition as simple as possible, we allow instead to define additional graph constraints like proposed by [Kleppe and Rensink, 2008]:

**Definition 2.5** (Graph Constraint)**.** *Let $TG$ be a type graph. A graph constraint over $TG$ is a total function $c : \mathcal{M}_{TG} \to \mathbb{B}$ that decides whether the constraint is satisfied by a model $m \in \mathcal{M}_{TG}$ typed by $TG$.*

Kleppe and Rensink formalize a number of constraint templates that are often required [Kleppe and Rensink, 2008]. To show how constraints can be formalized, we mention three constraint templates as example. The multiplicity constraint on an edge type requires that certain cardinality restrictions have to hold for outgoing edges of that type:

**Example 2.5** (Multiplicity Constraint)**.** *Let $TG = (N_{TG}, E_{TG}, src_{TG}, tgt_{TG}, inh)$ be a type graph. A multiplicity constraint over $TG$ is a function $mult(r, l, u) : \mathcal{M}_{TG} \to \mathbb{B}$ where*

- *$r \in E_{TG}$ is an edge type defined by $TG$,*
- *$l, u \in \mathbb{N} \cup \{\infty\}$ are natural numbers representing lower and upper bound with $l \leq u$, and*
- *satisfaction is defined for all $m = (N_m, E_m, src_m, tgt_m, lab) \in \mathcal{M}_{TG}$ by*

$$\begin{aligned} mult(r,l,u) \quad :\Leftrightarrow \quad & \forall s \in N_m, isKindOf(s, src_{TG}(r)) : \\ & l \leq \|\{e \in E \mid src_m(e) = s \wedge isInstanceOf(e,r)\}\| \leq u \end{aligned}$$

The opposite constraint on two edge types requires that for edges of a first type, there are opposite edges of a second type:

**Example 2.6** (Opposite Constraint)**.** *Let $TG = (N_{TG}, E_{TG}, src_{TG}, tgt_{TG}, inh)$ be a type graph. An opposite constraint over $TG$ is a function $opposite(r, o) : \mathcal{M}_{TG} \to \mathbb{B}$ where*

- *$r, o \in E_{TG}$ are edge types defined by $TG$ with $src_{TG}(r) = tgt_{TG}(o)$ and $tgt_{TG}(r) = src_{TG}(o)$, and*
- *satisfaction is defined for all $m = (N_m, E_m, src_m, tgt_m, lab) \in \mathcal{M}_{TG}$ by*

$$\begin{aligned} opposite(r,o) \quad :\Leftrightarrow \quad & \forall s,t \in N_m, isKindOf(s, src_{TG}(r)), isKindOf(t, tgt_{TG}(r)) : \\ & \|\{e \in E \mid src_m(e) = s \wedge tgt_m(e) = t \wedge isInstanceOf(e,r)\}\| = \\ & \|\{e \in E \mid src_m(e) = t \wedge tgt_m(e) = s \wedge isInstanceOf(e,o)\}\| \end{aligned}$$

The acyclicity constraint on an edge type requires that there are no cycles involving edges of that type. To be able to define the acyclicity constraint, we first need to define when a set of edges is cycle free:

- A path through a graph $G = (N, E, src, tgt)$ is a sequence $\langle e_1, \ldots, e_n \rangle \in E^+$, such that consecutive edges in the path are connected in the graph:

$$\forall i \in \mathbb{N}, 1 \leq i < n : tgt(e_i) = src(e_{i+1})$$

- A set of edges $E' \subseteq E$ is cycle free if there is no path $\langle e_1, \ldots, e_n \rangle \in E^+$ in $G$, such that:
$$tgt(e_n) = src(e_1) \wedge \forall i \in \mathbb{N}, 1 \leq i < n : e_i \in E'$$

Based on these helper definitions, we can define the acyclicity constraint:

**Example 2.7** (Acyclicity Constraint)**.** *Let $TG = (N_{TG}, E_{TG}, src_{TG}, tgt_{TG}, inh)$ be a type graph. A acyclicity constraint over $TG$ is a function $acyclic(r) : \mathcal{M}_{TG} \rightarrow \mathbb{B}$ where*

- *$r \in E_{TG}$ is an edge type defined by $TG$,*
- *satisfaction is defined for all $m = (N_m, E_m, src_m, tgt_m, lab) \in \mathcal{M}_{TG}$ by*

$$acyclic(r) \quad :\Leftrightarrow \quad \{e \in E_m \mid isInstanceOf(e, r)\} \text{ is cycle free}$$

Note that the two edges types have to be opposite of each other in the type graph so that this constraint is defined. These constraint templates can be used to extend a type graph [Kleppe and Rensink, 2008]:

**Definition 2.6** (Type Graph with Constraints)**.** *A type graph with constraints is a tuple $TGC = (TG, C)$ where*

- *$TG = (N, E, src, tgt, inh)$ is a type graph, and*
- *$C \in \mathcal{P}(\mathcal{M}_{TG} \rightarrow \mathbb{B})$ is a set of graph constraints over $TG$.*

**Example 2.8** (Type Graph with Constraints)**.** *For the type graph shown in Figure 2.4, we need to impose additional constraints on statemachine models. We can use the multiplicity constraint template to ensure that a transition has exactly one source and target state:*

$$mult(\textsf{source}, 1, 1), mult(\textsf{target}, 1, 1)$$

*We can use the acyclicity constraint template to ensure that the state hierarchy contains no cycle:*

$$acyclic(\textsf{states})$$

For a type graph with constraints, we can determine when a model is syntactically correct with respect to it [Kleppe and Rensink, 2008]:

**Definition 2.7** (Syntactic Correctness)**.** *A model $m \in \mathcal{M}$ is syntactically correct with respect to a type graph with constraints $TGC = (TG, C)$ if*

- *$m \in \mathcal{M}_{TG}$ is typed by $TG$, and*
- *$m$ satisfies all constraints $C$: $\forall c \in C : c(m)$.*

Type graphs provide an elegant way to formalize metamodels. However, type graphs are not models and thus cannot be directly used as metamodels.

### 2.2.3 Simplified E-MOF Metametamodel

In this section, we formalize a simplified version of E-MOF based on type graphs. However, this formalization can be easily extended to completely cover E-MOF by adding additional constraints. We define the following helper predicate for a model $m = (N, E, src, tgt, lab)$:

- $Edge(x, y, z)$. This predicate states that there is an edge from node $x \in N$ to node $y \in N$ in the model that is an instance of the type $z \in \mathbb{ID}$:

$$Edge(x, y, z) :\Leftrightarrow \exists e \in E : src(e) = x \land tgt(e) = y \land isInstanceOf(e, z)$$

Similar to a type graph, a metamodel defines whether a model is syntactically correct with respect to it. However, according to our definition, a type graph is not a model. For instance, our definition of model does not directly provide an inheritance relation between nodes. However, a type graph can be represented as a model which we call a metamodel:

**Definition 2.8** (E-MOF Metamodel)**.** *An E-MOF metamodel is a model* $mm = (N_{mm}, E_{mm}, src_{mm}, tgt_{mm}, lab)$ *that is syntactically correct with respect to the following type graph with constraints* $TGC_{mm} = (TG_{mm}, C_{mm})$ *with* $TG_{mm} = (N_{TG_{mm}}, E_{TG_{mm}}, src_{TG_{mm}}, tgt_{TG_{mm}}, inh_{mm})$*:*

- $N_{TG_{mm}} := \{$*Class*, *Reference*$\}$
- $E_{TG_{mm}} := \{$*superTypes*, *features*, *type*, *class*, *opposite*$\}$
- $src_{TG_{mm}} := \{$*superTypes* $\mapsto$ *Class*, *features* $\mapsto$ *Class*, *type* $\mapsto$ *Reference*, *class* $\mapsto$ *Reference*, *opposite* $\mapsto$ *Reference*$\}$
- $tgt_{TG_{mm}} := \{$*superTypes* $\mapsto$ *Class*, *features* $\mapsto$ *Reference*, *type* $\mapsto$ *Class*, *class* $\mapsto$ *Class*, *opposite* $\mapsto$ *Reference*$\}$
- $inh_{mm} := \emptyset$
- $C_{mm} := \{opposite($*class*, *features*$), opposite($*opposite*, *opposite*$), mult($*type*$, 1, 1), mult($*class*$, 1, 1), mult($*opposite*$, 0, 1), acyclic($*superTypes*$)\}$

*Let* $\mathcal{MM}$ *be the set of all models that are syntactically correct with respect to* $TGC_{mm}$*. A metamodel* $mm \in \mathcal{MM}$ *defines a type graph with constraints* $TGC = (TG, C)$ *with* $TG = (N_{TG}, E_{TG}, src_{TG}, tgt_{TG}, inh)$ *in the following way:*

- $N_{TG} := \{n \in N_{mm} \mid isInstanceOf(n,$ *Class*$)\}$
- $E_{TG} := \{n \in N_{mm} \mid isInstanceOf(n,$ *Reference*$)\}$
- $src_{TG}(e) = n \in E_{TG} :\Leftrightarrow Edge(n, e,$ *features*$)$
- $tgt_{TG}(e) = n \in E_{TG} :\Leftrightarrow Edge(e, n,$ *type*$)$
- $inh := \{(n_1, n_2) \in N_{TG} \times N_{TG} \mid Edge(n_1, n_2,$ *superTypes*$)\}$
- $C := \{opposite(r, o) \mid r, o \in E_{TG} \land Edge(r, o,$ *opposite*$)\}$

Due to the constraints defined by $TGC_{mm}$, the type graph defined by the metamodel always exists and is well-defined.

**Example 2.9** (E-MOF Metamodel)**.** *Figure 2.5 depicts a metamodel as UML object diagram. The metamodel defines the type graph shown in Figure 2.4. Node types are represented by nodes of type* Class*, and edge types by nodes of type* Reference *that are connected to their source and target node type by edges of type* features*,* class *and* type*. For the sake of simplicity, edges that are opposite of each other are shown as a single line with the name of the edge types on the different line ends.*

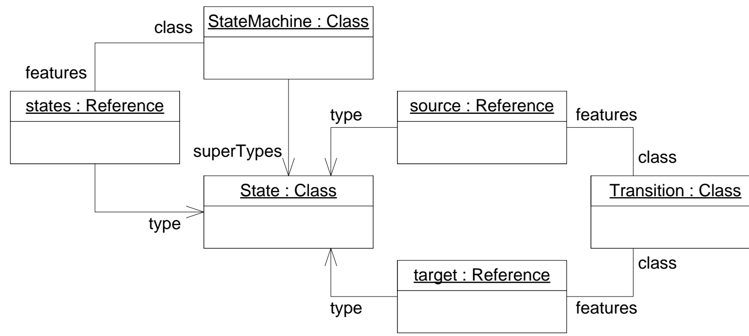



**Figure 2.5:** Example metamodel represented as UML object diagram

Using the type graph with constraints associated to a metamodel, we can define when a model is syntactically correct with respect to the metamodel. If the model is syntactically correct, we say that it conforms to the metamodel [Kleppe and Rensink, 2008]:

**Definition 2.9** (Conformance)**.** *Let $mm \in \mathcal{MM}$ be a metamodel and $TGC$ the type graph with constraints that is defined by $mm$. A model $m \in \mathcal{M}$ conforms to $mm$ if $m$ is syntactically correct with respect to $TGC$. Then we write $m \vDash mm$.*

All models that conform to a metamodel make up the modeling language that is defined by the metamodel [Kleppe, 2008]:

**Definition 2.10** (Modeling Language)**.** *Let $mm \in \mathcal{MM}$ be a metamodel. A modeling language $\mathcal{L}_{mm}$ defined by the metamodel $mm$ is the set of models that conform to that metamodel:*

$$\mathcal{L}_{mm} := \{m \in \mathcal{M} \mid m \vDash mm\}$$

In our understanding, a modeling language thus only covers the abstract syntax. However, based on a modeling language, we may also define a concrete syntax and semantics which are explained in Section 2.3 (*Concrete Syntax of Modeling Languages*) and Section 2.4 (*Semantics of Modeling Languages*), respectively.

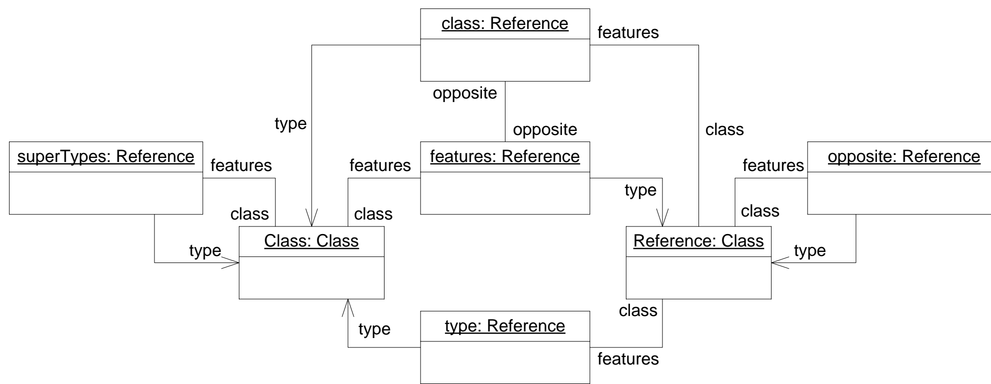
**Definition 2.11** (E-MOF Metametamodel)**.** *The E-MOF metametamodel $mmm \in \mathcal{MM}$ is the metamodel that defines the type graph with constraints $TGC_{mm}$ by which metamodels are typed.*

Figure 2.6(a) shows the simplified metametamodel as a UML object diagram, and Figure 2.6(b) shows the underlying type graph as a UML class diagram. Similar to a modeling language, we can define a metamodeling language:

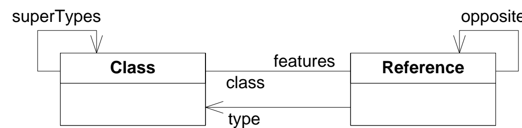
**Definition 2.12** (E-MOF Metamodeling Language)**.** *Let $mmm$ be the E-MOF metametamodel. The E-MOF metamodeling language $\mathcal{L}_{mmm}$ defined by the metametamodel $mmm$ is the set of metamodels that conform to that metametamodel:*

$$\mathcal{MM} := \mathcal{L}_{mmm} = \{mm \in \mathcal{M} \mid mm \vDash mmm\}$$

Since a metamodel is also a model and the metametamodel is a metamodel, the conformance relation can also be used to check whether a metamodel conforms to a




(a) Model representation as UML object diagram




(b) Type graph represented as UML class diagram

**Figure 2.6:** Simplified E-MOF metametamodel

metametamodel. The semantics of a metamodeling language maps a metamodel to the modeling language defined by the metamodel. Since the metametamodel is a metamodel and all metamodels have to conform to the metametamodel, the metametamodel needs to conform to itself.

### 2.2.4 Complete E-MOF Metametamodel

In the last section, we presented a formalization of a simplified version of the E-MOF metametamodel. However, the E-MOF metametamodel provides additional constructs for defining metamodels that impose additional constraints on models. Figure 2.7 displays the complete E-MOF metametamodel as a UML class diagram. Most of these additional constructs are formalized by the graph constraint templates published in [Kleppe and Rensink, 2008].

**Abstract Classes.** A Class can be made abstract. If a class is abstract, there must be no nodes in the model that represent instances of the class. Therefore, abstract classes provide a means to extract common features into a super class that is not required to have instances.

**Primitive Types.** In the graph, instances of a PrimitiveType are also represented as nodes, similar to instances of Classes. However, compared to Classes, PrimitiveTypes do not define references. As a consequence, nodes representing instances of a PrimitiveType do not have outgoing edges. They are thus the terminal nodes in graphs.

**Data Types.** DataTypes represent predefined primitive types like Boolean, Integer

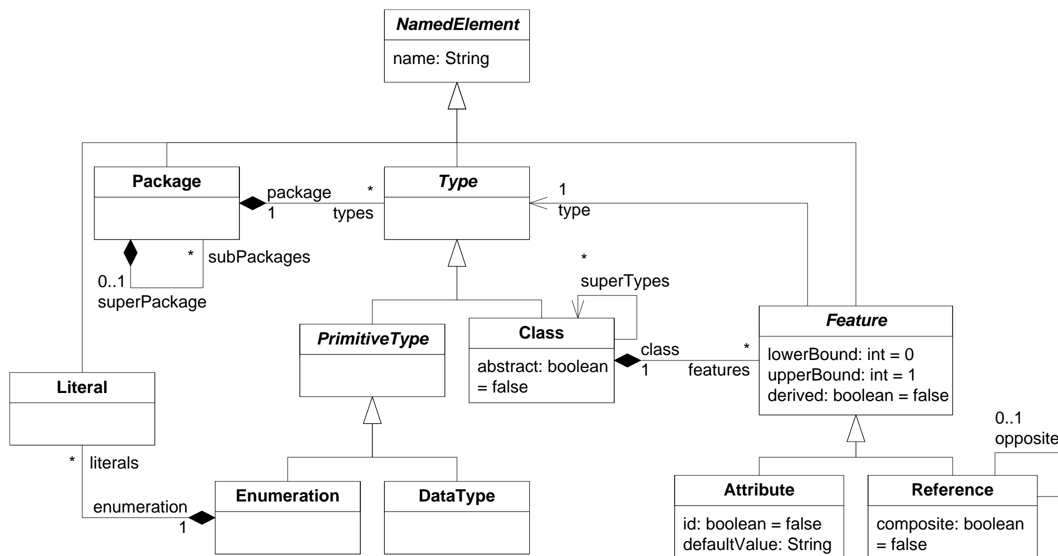



**Figure 2.7:** E-MOF metametamodel represented as UML class diagram

and String. For each literal of these data types, there is at most one node in the model. Edges in the model targeting the same literal refer to the same node. Data types may define an infinite number of possible literals, e.g. String. However, a model can only use a finite number of literals of a data type, since the set of nodes is finite.

**Enumerations and Literals.** An Enumeration can define a finite set of literals. Each literal that is used in the model is represented by exactly one node in the model. A Literal has a name to be able to distinguish it from the other literals.

**Types.** Type is a common abstract super class of Class and PrimitiveType. In the complete E-MOF metametamodel, all nodes have to be instances of Types. To be able to distinguish types, a Type has a name. The name of a type has to be unique among all types that are associated to the same package.

**Attributes.** In the graph, instances of an Attribute are also represented as edges, similar to instances of Reference. However, compared to References, Attributes have a PrimitiveType as type. As a consequence, edges representing instances of Attributes target nodes representing instances of primitive types.

**Features.** Feature is a common abstract super class of Reference and Attribute. In the complete E-MOF metametamodel, all edges have to be instances of features. To be able to distinguish features from each other, a Feature has a name. The name of the feature needs to be unique within all features of the class—including the ones inherited from the super types.

**Multiplicity.** There may be several edges outgoing from the same node that are instances of the same feature. To be able to restrict the number of outgoing edges of the same type, a Feature has a multiplicity—denoted through lower bound and upper bound. To conform to the metamodel, the number of edges—that are outgoing

from the same nodes and that are instances of the same feature—has to be at least the feature's lower bound and at most the feature's upper bound. The lower bound of a feature should be smaller or equal to the upper bound of the feature, but greater or equal to 0. The upper bound of a feature may be infinite, i.e. not restricted. However, there may be no infinite number of outgoing edges, since the overall number of edges in a model is finite.

**Identifying Attributes.** An attribute can serve as an identifier for nodes that are instances of the class in which the attribute is defined. The values of the identifier attribute have to be unique among all instances of the attribute's parent class or all its subclasses. A class should define at most one identifying attribute—including the inherited ones.

**Attribute Default Values.** An attribute may have a default value to reduce the effort for setting values that are often used. When an instance of a class is created, the value of each attribute defining a default value is automatically set to the default value. The default value is specified by a String which refers to the identifier of the node representing the value.

**Composite References.** Composite references restrict the edges that are instances of these references to a forest structure. An edge that is an instance of a composite reference is called a composite edge. Each node in the model may be the target of at most one composite edge. Moreover, all composite edges need to form an acyclic graph, i.e. there must not be a path of composite edges from a node to itself. The opposite reference of a composite needs to have an upper bound of 1.

**Ordered Features.** A feature is multi-valued if its upper bound is more than 1. In this case, there may be multiple edges outgoing of a node that are instances of the same reference. Currently, there is no order between the edges in a model. However, by default, all multi-valued features in E-MOF are ordered. To support this in our formalization, we need to introduce indices for edges, like proposed in [Kleppe and Rensink, 2008].

**Derived Features.** Values of features may be derived from the values of other features. Since the values of these features can be derived, they are not represented in the model graph. The metamodel needs to specify the function to determine the values of the derived features. E-MOF envisions to use the Object Constraint Language (OCL) [Object Management Group, 2006b] to specify these functions.

**Packages.** To group related types, they can be organized into packages. A Package consists of a number of types and can have sub packages. To be able to distinguish packages from each other, a Package has a name. The name of a sub package needs to be unique among the packages which belong to the same super package. The name of a root package needs to be unique among the packages which do not have a super package.

**Labels of Nodes and Edges.** To distinguish types from different packages, nodes

refer to them by means of their fully qualified name in E-MOF. Thus the identifier of a type node in the metamodel must be the fully qualified name of the type. The fully qualified name of a type is its name prefixed by the fully qualified name of the package. Similarly, the fully qualified name of the package is its name prefixed by the fully qualified name of the super package. For feature nodes, the identifier is just the name of the feature which is enough to distinguish the different features of a certain class. Since there might be features with the same name in different classes in E-MOF, we have to soften our restriction that node identifiers have to be unique.

**Additional Constraints.** There might be constraints which cannot be expressed using the constructs provided by Figure 2.7. More advanced constraints can be expressed using the Object Constraint Language (OCL) [Object Management Group, 2006b] as invariants on instances of a certain class. All nodes that are instances of this class need to fulfill the constraints defined for its class or any of its super classes.

### 2.2.5 UML Object and Class Diagrams

E-MOF models can be represented by UML object diagrams, and E-MOF metamodels can be represented by UML class diagrams [Object Management Group, 2009].

**UML Object Diagrams.** E-MOF models can be generically represented as UML object diagrams [Object Management Group, 2009]. A rectangle represents an object—a node which is instance of a class—and is divided into two sections. The first displays and underlines the identifier and the label of the node which are separated by a colon. The second lists the edges that have the node as source and that are instances of Attributes. Each edge is represented as a text line showing the label of the attribute and the value of the attribute—the identifier of the target node which are separated by an equality sign. A line from one rectangle to another represents a link—an edge from an object to another that is an instance of a Reference. The line has an arrow head to indicate the direction of the edge from the source to the target node and is labeled by the name of the reference. Edges that are instances of references opposite of each other are shown as one single line without an arrow head. Composite edges—that are instances of composite references—are represented by a filled diamond at the side, indicating the node from which the edges start, and thus do not require an arrow head to indicate the direction of the edge.

**Example 2.10** (Metamodel represented as UML Object Diagram)**.** *Figure 2.8 shows the metamodel of the state machine modeling language as a UML object diagram. This metamodel refines the metamodel shown in Figure 2.2 with additional classes and references. StateMachine, State and Transition are classes, states, outgoing, source and target are references, and name, trigger and effect are attributes. Figure 2.8 also illustrates the predefined primitive type String as a separate node which is used by all the attributes.*

**Example 2.11** (Model represented as UML Object Diagram)**.** *Figure 2.9 depicts the example state machine model using UML object diagrams. In the figure, for example, nodes denote either states or transitions, and reference edges denote either the source and target state of a transition. Figure 2.9 depicts attribute edges for the name of a state as well as for the trigger and effect of a transition.*

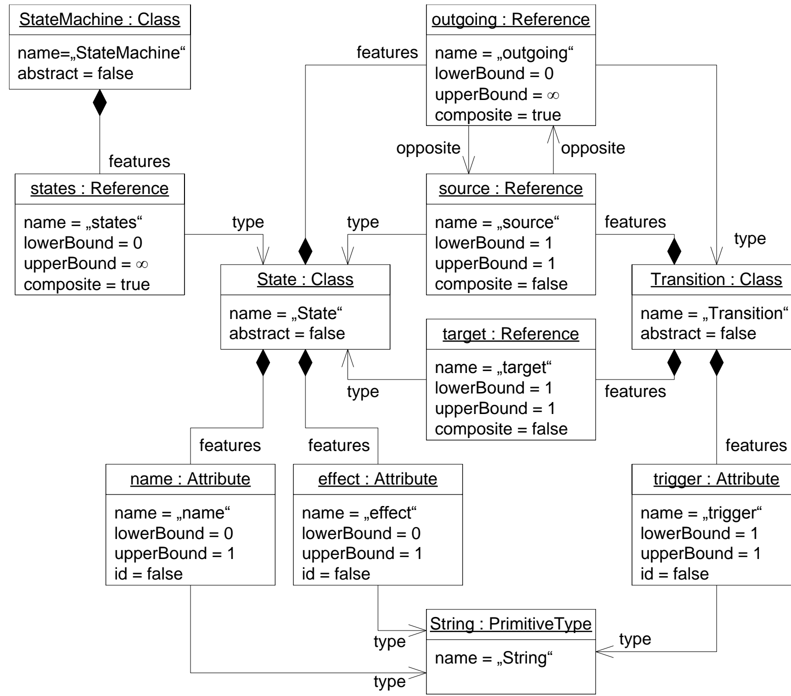



**Figure 2.8:** Metamodel represented as UML object diagram

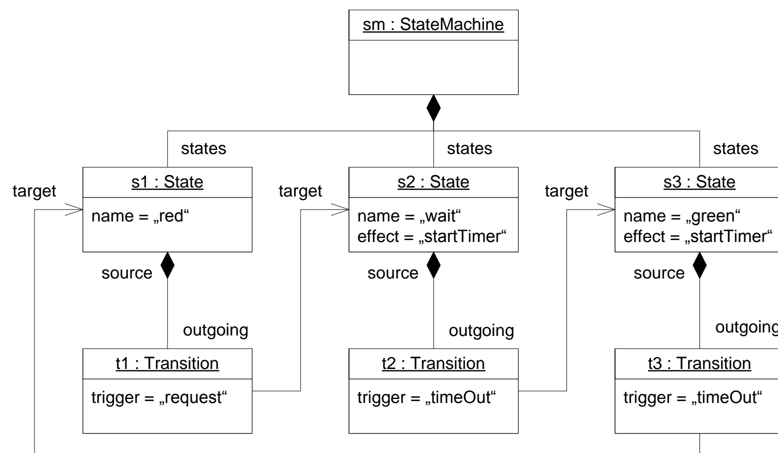



**Figure 2.9:** Model represented as UML object diagram

**UML Class Diagrams.** E-MOF metamodels can be represented as UML class diagrams [Object Management Group, 2009]. UML class diagrams are a more compact representation than UML object diagrams, and thus we use this notation throughout the thesis. Classes are represented by rectangles which are divided into sections: the first displaying the name of the class, and the second listing the attributes defined by the class. If a class is abstract, its name is formatted in italic style. Attributes are represented by text lines in the rectangles of the classes in which they are defined. A text line defines the name of an attribute followed by a colon, the name of the attribute's type, and optionally the default value prefixed by an equality sign. References are represented by lines that connect the class in which the reference is defined to the class which is the type of the reference. If the reference is unidirectional—i.e. has no opposite reference—an arrow head indicates the direction of the reference. Otherwise, if the reference is bidirectional—i.e. has an opposite reference—both references are shown only as one line without an arrow head. The name of the reference as well as its multiplicity are shown at the side indicating the type of the reference. Composite references are represented by a filled diamond at the side, indicating the class in which the reference is defined, and thus do not require an arrow head to indicate the direction of the reference.

**Example 2.12** (Metamodel represented as UML Class Diagram)**.** *Figure 2.10 shows the metamodel of the state machine modeling language in concrete syntax. The example metamodel defines classes to specify StateMachines, States and Transitions. It defines the composite references states and outgoing, the bidirectional reference outgoing/source and the unidirectional references states and target. The example metamodel defines attributes to specify the name and effect of a state as well as the trigger of a transition. All the attributes in the example metamodel are of type String.*

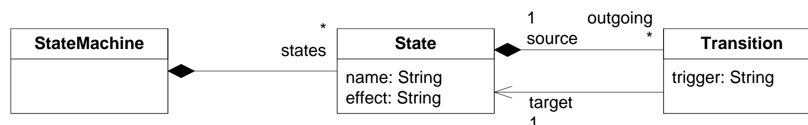



**Figure 2.10:** Metamodel represented as UML class diagram

### 2.2.6 Eclipse Modeling Framework

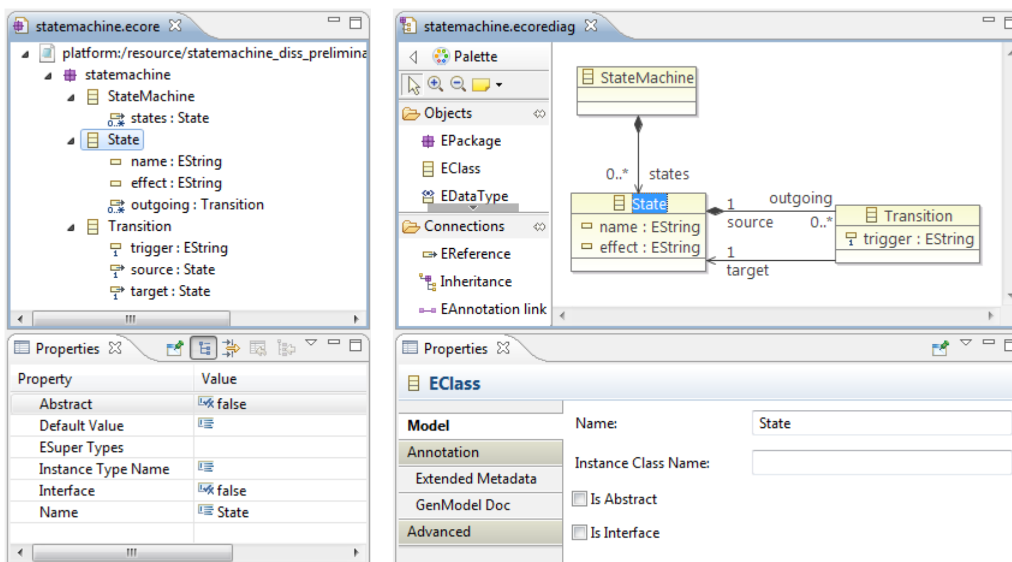
To be able to build models with a modeling language, tool support is required to persist, edit and interpret models. Language workbenches [Fowler, 2005] such as the Eclipse Modeling Framework (EMF)[1], MetaCase MetaEdit+[2], the Generic Modeling Environment (GME)[3] or Microsoft DSL Tools[4] significantly reduce the effort to build tool support for modeling languages around the metamodels. Language workbenches provide languages to build the abstract and concrete syntax as well as the semantics of modeling languages. A generic modeling framework interprets the defini-

[1] see EMF web site: `http://www.eclipse.org/modeling/emf/`
[2] see MetaCase MetaEdit+ web site: `http://www.metacase.com/`
[3] see GME web site: `http://www.isis.vanderbilt.edu/Projects/gme/`
[4] see Microsoft DSL Tools web site: `http://msdn.microsoft.com/de-de/library/bb126235(VS.90).aspx`

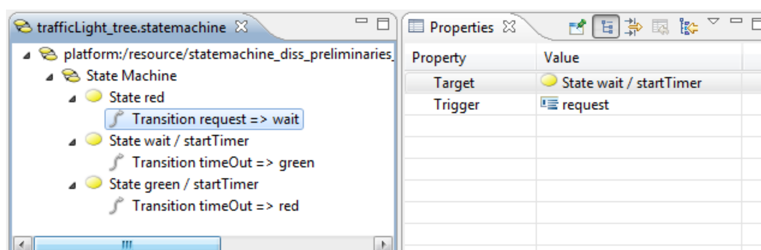
tions written in these languages and provides the appropriate tool support to persist, edit and interpret models conforming to the modeling language [Broy et al., 2010].

In this thesis, we use EMF for the implementation of modeling languages, as it is probably the most widely used modeling framework implementing E-MOF. EMF provides an implementation of E-MOF and various other OMG standards like e.g. the XML Metadata Interchange (XMI) [Object Management Group, 2007], the Object Constraint Language (OCL) [Object Management Group, 2006b], Query View Transformation (QVT) [Object Management Group, 2008b], and the MOF Model To Text Transformation Language (MOF2Text) [Object Management Group, 2008a]. In this section, we showcase the implementation of the abstract syntax of the state machine modeling language using EMF.

A metamodel is specified in Ecore—the metametamodel provided by EMF—which is similar to the metametamodel presented in Section 2.2.4 (*Complete E-MOF Metamodel*). Ecore is an implementation of E-MOF in Java for the Eclipse[5] platform. From an Ecore metamodel, a Java API for model access as well as a structural editor can be generated. The structural editor represents a model as a tree which corresponds to the abstract syntax.




(a) Definition (abstract syntax) (b) Definition (concrete syntax)




(c) Structural editor generated by EMF

**Figure 2.11:** Abstract syntax in EMF

[5]see Eclipse web site: `http://www.eclipse.org/`

**Example 2.13** (Implementation of Abstract Syntax)**.** *Figure 2.11 provides screenshots of the tools to build an Ecore metamodel that are provided by EMF.*

*Figure 2.11(a) shows the structural editor to build the Ecore metamodel using the abstract syntax. The structural editor displays the state machine metamodel as a hierarchy of packages, types and features. The properties of a metamodel element can be accessed through the properties view in the structural editor.*

*Figure 2.11(b) shows the graphical editor to build the Ecore metamodel using the concrete syntax. The graphical editor displays the state machine metamodel as a UML class diagram and is thus particularly suited for language engineers already familiar with UML class diagrams. Besides accessing properties through the properties view, new metamodel elements can be created using the palette of the graphical editor.*

*Figure 2.11(c) displays a screenshot of the structural editor generated from the state machine Ecore metamodel. The code generation can be customized by a generator model which decorates the Ecore metamodel or by overwriting the generated code. For the structural editor shown in the figure, the icons and labels of the model elements have been customized.*

## 2.3 Concrete Syntax of Modeling Languages

We first define the purpose of the concrete syntax of a modeling language, before we explain how it can be implemented in practice.

### 2.3.1 Concrete Syntax of a Modeling Language

The concrete syntax defines the concrete, external representation of a model. The concrete representation of a model is supposed to be read and understood by humans, but can also be used to persist models. We define the concrete syntax of a modeling language similar to [Chen et al., 2005] based on the abstract syntax:

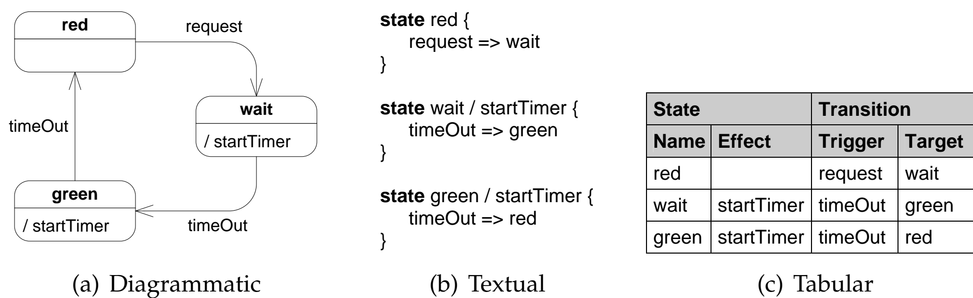
**Definition 2.13** (Concrete Syntax)**.** *The concrete syntax $CS$ of a modeling language $\mathcal{L}$ is defined by*

- *a concrete domain $\mathcal{CD}$ that specifies the set of possible concrete representations (graphical, textual or tabular) for models, and*
- *a concrete mapping $CS : \mathcal{CD} \to \mathcal{L}$ that maps a concrete representation $c \in \mathcal{CD}$ in the concrete domain to a model $CS(c) \in \mathcal{L}$.*

There are different syntactic domains for the definition of a concrete syntax: textual, diagrammatic and tabular. The textual syntax visualizes the model as linear texts, the diagrammatic syntax shows the model in diagrams with layout information, and the tabular syntax visualizes the model in two-dimensional tables. The mapping can be employed to implement tool support for authoring models in concrete syntax [Goldschmidt et al., 2008]. Note that due to the definition of the mapping different concrete representations can be mapped to the same abstract representation.

**Example 2.14** (Concrete Syntax)**.** *Figure 2.12 represents the example state machine model using several kinds of concrete syntax. Figure 2.12(a) represents the example model in a*

*diagrammatic concrete syntax. A state is represented as a rounded rectangle that is divided into two sections: the first section shows the name of the state, and the second a slash followed by the effect of the state. A transition is represented as an arrow that is labeled with the transition's trigger. Figure 2.12(b) represents the example model in a textual concrete syntax. A state is represented as a text block which is prefixed by the state's name and optionally a slash together with the transition's effect, and which contains all the transitions that have the state as source. A transition is represented as a line consisting of its trigger, an arrow, and its target state. Figure 2.12(c) represents the example model in a tabular concrete syntax [Herrmannsdoerfer et al., 2008c]. A transition is represented as a row, and there are columns to specify source state of the transition and its effect as well as the trigger of the transition and its target state.*




| State | | Transition | |
|---|---|---|---|
| Name | Effect | Trigger | Target |
| red | | request | wait |
| wait | startTimer | timeOut | green |
| green | startTimer | timeOut | red |

(a) Diagrammatic (b) Textual (c) Tabular

**Figure 2.12:** Model in concrete syntax

### 2.3.2 Implementing the Concrete Syntax

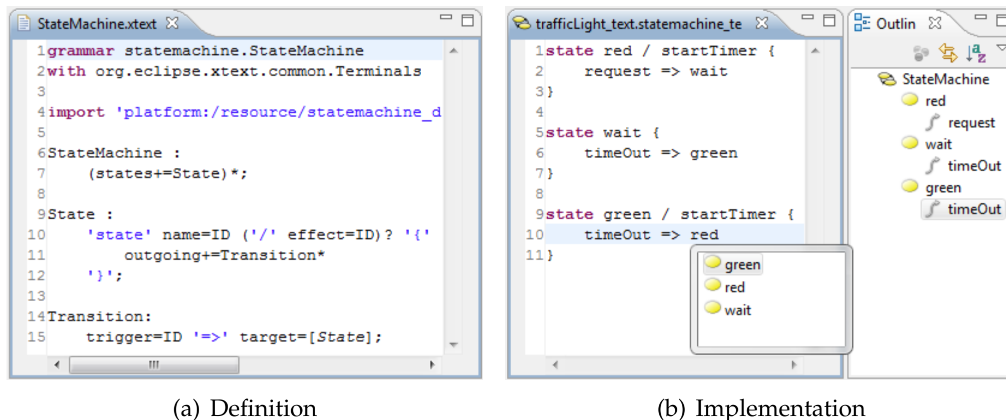
A language for defining concrete syntax needs to provide constructs that define both the concrete domain $\mathcal{CD}$ and the mapping $CS : \mathcal{CD} \rightarrow \mathcal{L}$ from the concrete domain to models. In MOF, the abstract syntax definition is independent of the concrete syntax definition. There are other OMG standards like XML Metadata Interchange (XMI) [Object Management Group, 2007] or Human-Usable Textual Notation (HUTN) [Object Management Group, 2004] that define a concrete syntax based on the abstract syntax. In contrast, GOPPR, GME and Microsoft DSL tools advocate the integrated definition of abstract and concrete syntax of a modeling language.

Implementations of the concrete syntax do not only allow the language users to browse the models, but also to edit them. Moreover, concrete syntax can also be used to persist models. For example, the OMG standard XML Metadata Interchange (XMI) [Object Management Group, 2007] provides a bidirectional mapping of models to XML that is canonically derived from the metamodel. In the terminology introduced in Section 2.3.1 (*Concrete Syntax of a Modeling Language*), XML is the concrete domain and the canonical mapping is the function defining the concrete syntax. Currently, EMF provides metamodeling languages to specify textual and diagrammatic concrete syntax.

**Textual Concrete Syntax.** In EMF, the textual concrete syntax is specified by a grammar using Xtext[6]. The metamodel can either be derived from the grammar, or the

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

grammar can be based on an existing metamodel. The grammar is used to specify the concrete syntax which has text as concrete domain. Xtext already comes with a number of standard terminals that can be used by the grammar. The grammar itself is specified using a sequence of productions. From the concrete syntax definition, a parser and a textual editor can be generated. The textual editor already provides a number of functions, e.g. on-the-fly parsing, syntax highlighting and auto completion. Xtext provides a number of extension points to customize the generated textual editor.




(a) Definition (b) Implementation

**Figure 2.13:** Textual concrete syntax in EMF

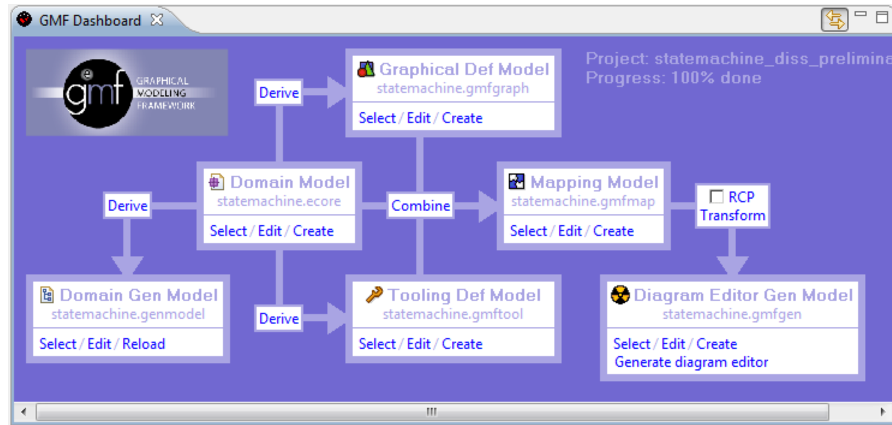
**Example 2.15** (Implementation of Textual Concrete Syntax)**.** *Figure 2.13(a) shows an Xtext grammar that defines the textual concrete syntax of the state machine modeling language. We base the grammar on the existing metamodel by importing the Ecore metamodel. For the state machine modeling language, we define a production for each class defined in the metamodel.*

*Figure 2.13(b) displays the textual editor that was generated from the state machine grammar. No customization was necessary to generate the textual editor shown in Figure 2.13(b). Note that the inner box illustrates auto completion that lists the possible target states of the transition.*

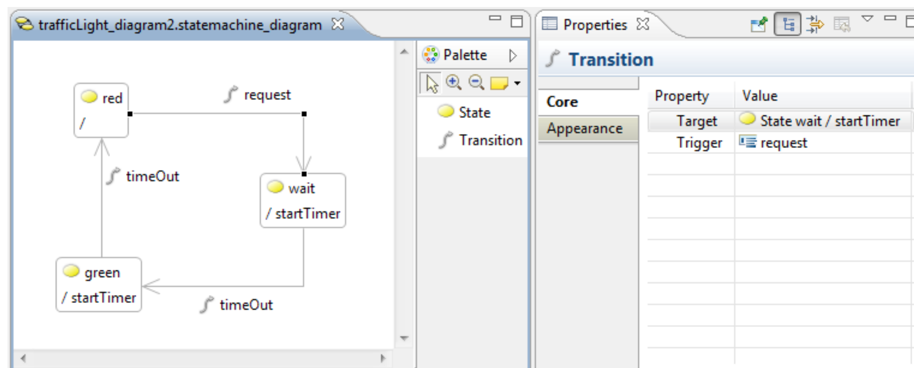
**Diagrammatic Concrete Syntax.** In EMF, the diagrammatic concrete syntax is specified by a set of models using the Graphical Modeling Framework (GMF)[7]. Figure 2.14(a) shows the GMF dashboard which supports the process of creating the GMF models. The GMF models are based on a domain model expressed in Ecore from which an appropriate generator model can be derived. Note that domain model is the GMF term for a metamodel. GMF provides wizards to derive the following models to define the concrete domain from the domain model: the graphical definition model defines the graphical elements like nodes and edges in the diagram, and the tooling definition model defines the tools available to author a diagram. The concrete mapping is defined by the mapping model which combines the domain model, the graphical definition model and the tooling definition model: Graphical elements from the graphical definition model and the tools from the tooling definition model are

[7]see GMF web site: `http://www.eclipse.org/modeling/gmp/`

mapped onto the constructs from the domain model. The mapping model is transformed into a diagram generator model from which a diagram editor can be generated. The diagram generator model can be altered to customize the code generation. The process follows an MDA approach as explained in Section 2.1.4 (*Major Initiatives*): The generator models are platform-specific (PSM), whereas all other models are platform-independent (PIM).




(a) Definition




(b) Implementation

**Figure 2.14:** Diagrammatic concrete syntax in EMF

**Example 2.16** (Implementation of Diagrammatic Concrete Syntax)**.** *Figure 2.14 shows the diagram editor generated from GMF models defined for the state machine modeling language. The diagram editor shows the graphical elements in the diagram and the tools in the palette. GMF also provides more advanced features like e.g. annotating, zooming, printing and layouting for the generated editor. The properties of a graphical element can be accessed through the properties view. The generated editor needed to be only customized to correctly show the labels of the transitions.*

## 2.4 Semantics of Modeling Languages

We first formally define the semantics of a modeling language, before we explain how it can be implemented in practice.

### 2.4.1 Semantics of a Modeling Language

The semantics of a modeling language associates a meaning to each model that is syntactically correct. As a consequence, syntactic correctness ideally should be chosen in a way that at least all syntactically correct models have a semantics. We define the semantics of a modeling language similar to [Harel and Rumpe, 2004] and [Chen et al., 2005] based on the abstract syntax:

**Definition 2.14** (Semantics)**.** *The semantics $S$ of a modeling language $\mathcal{L}$ is defined by*

- *a semantic domain $\mathcal{SD}$ that defines the set of meanings and contains a relation $\equiv$: $\mathcal{SD} \times \mathcal{SD}$ to check equivalence between meanings, and*
- *a semantic mapping $S : \mathcal{L} \to \mathcal{SD}$ that maps a model $m \in \mathcal{L}$ to its meaning $S(m) \in \mathcal{SD}$ in the semantic domain.*

Note that the meaning of a model is independent of its concrete representation, as we base the semantics solely on the abstract syntax. Generally, there are three ways to define semantics [Winskel, 1993]. The first one is to define the translation to another formalism (denotational semantics), the second one is to specify how a model is interpreted as sequence of computational steps (operational semantics), and the third one is to describe the semantics of the modeling language by an algebra (algebraic semantics).

**Example 2.17** (Semantics)**.** *In this example, we define a denotational semantics for the state machine modeling language. We first define helper functions required to navigate a state machine model, before we define the semantics.*

Model navigation. *We use the metamodel shown in Figure 2.10 as basis for the semantics definition. To define the semantics, we need a few helper functions to navigate the model $m = (N_m, E_m, src_m, tgt_m, lab)$. For each class defined in the metamodel, we can define a set of its instances in the model. For example, the set $State$ denotes the set of instances of the class* **State** *and its subclasses:*

$$State := \{x \in N_m \mid isKindOf(x, \textsf{State})\}$$

*For each feature defined in the metamodel, we can define a function to access its values in the model. Functions for multi-valued features return a set of nodes. For example, $outgoing : State \to \mathcal{P}(Transition)$ returns the set of* **outgoing** *transitions of a state $s \in State$:*

$$s.outgoing := \{t \in Transition \mid Edge(s, t, \textsf{outgoing})\}$$

*Functions for single-valued features return a single node. For example, $target : Transition \to State$ returns the* **target** *state of a transition $t \in Transition$:*

$$t.target = s \in State :\Leftrightarrow Edge(t, s, \textsf{target})$$

*Triggers and effects are only defined as Strings in the metamodel. To be able to use them for defining the semantics, we need to define the sets of events and actions used as triggers and effects in a model:*

$$\begin{aligned} Event &:= \{e \in N_m \mid \exists t \in Transition : t.trigger = e\} \\ Action &:= \{a \in N_m \mid \exists s \in State : s.effect = a\} \end{aligned}$$

*The semantics is defined based on the instance of the class* **StateMachine** *in the model. Therefore, we also need the function* $statemachine : \mathcal{L}_{mm} \rightarrow StateMachine$ *to obtain the single instance of* **StateMachine** *in the model:*

$$statemachine(m) = sm :\Leftrightarrow sm \in StateMachine \tag{2.1}$$

Semantics. *As semantic domain, we choose the input/output (I/O) behavior of the state machine [Rumpe, 1998]. One possibility to define the I/O behavior of state machines is through a stream-based function [Broy and Stølen, 2001] that maps a stream of events onto a stream of actions. The semantic domain is thus the set of these stream-based functions:*

$$\mathcal{SD} := \{Event^* \rightarrow \mathcal{P}(Action^*)\}$$

*Two stream-based functions* $s_1, s_2 \in \mathcal{SD}$ *are semantically equivalent if they produce the same stream of actions for the same stream of events:*

$$s_1 \equiv s_2 :\Leftrightarrow \forall es \in Event^* : s_1(es) = s_2(es)$$

*We define the semantics of the state machine modeling language by the helper function* $T : State \rightarrow \mathcal{SD}$ *which maps a state* $s \in State$ *to a function* $T(s) : Event^* \rightarrow \mathcal{P}(Action^*) \in \mathcal{SD}$*. The function* $T$ *is defined by induction over the stream of input events. In the base case, there are no more events left to be processed for a state* $s \in State$*:*

$$T(s)(\langle\rangle) := \{\langle\rangle\}$$

*The inductive step for a state* $s \in State$ *consumes the next event* $e \in Event$*. Based on the transitions activated in the state, we can decide whether the event leads to a state change. The set of transitions activated by an event* $e \in Event$ *in a state* $s \in State$ *is the set of outgoing transitions having the event as trigger:*

$$activated(s, e) := \{t \in s.outgoing \mid t.trigger = e\} \tag{2.2}$$

*In case there is at least one transition activated by event* $e \in Event$*, i.e.* $activated(s, e) \neq \emptyset$*, the semantics of the state* $s \in State$ *is defined as the union of the semantics of the states to which control can transition:*

$$T(s)(\langle e\rangle \circ es) := \bigcup_{t \in activated(s,e)} \langle t.target.effect\rangle \circ T(t.target)(es) \tag{2.3}$$

*The operator* $\circ$ *concatenates streams and is lifted to sets of streams if required. The operator* $\langle.\rangle$ *puts a single event or action into a stream or produces the empty stream* $\langle\rangle$ *in case of* $\perp$*. Otherwise, in case there are no activated transitions, the state machine remains in the same state:*

$$T(s)(\langle e\rangle \circ es) := T(s)(es)$$

*Since currently the metamodel does not define initial states, we define the semantics of the state machine* $sm \in StateMachine$ *to be the union of the semantics of all its states:*

$$T(sm)(es) := \bigcup_{s \in sm.state} T(s)(es) \tag{2.4}$$

*We define the semantics* $S(m) : Event^* \rightarrow \mathcal{P}(Action^*) \in \mathcal{SD}$ *of a model* $m \in \mathcal{L}_{mm}$ *based on the single instance of* **StateMachine** *in the model:*

$$S(m)(es) := T(statemachine(m))(es)$$

*However, as we have seen above, the semantics is defined based on the structure defined by the metamodel.*

### 2.4.2 Implementing the Semantics

A language for defining semantics needs to provide constructs that define both the semantic domain $\mathcal{SD}$ and the mapping $S : \mathcal{L} \rightarrow \mathcal{SD}$ from models to the semantic domain. For the metamodeling languages MOF, GOPPR, MetaGME and DSL Tools mentioned above, there is no standard means to explicitly define the semantics of a modeling language. However, they provide languages for model-to-model or model-to-text transformation to implicitly define the semantics of a modeling language. For example, there are the OMG standards MOF Query View Transformation (QVT) [Object Management Group, 2008b] or MOF Model To Text Transformation Language (MOF2Text) [Object Management Group, 2008a] that allow language engineers to build transformations for MOF-based modeling languages.

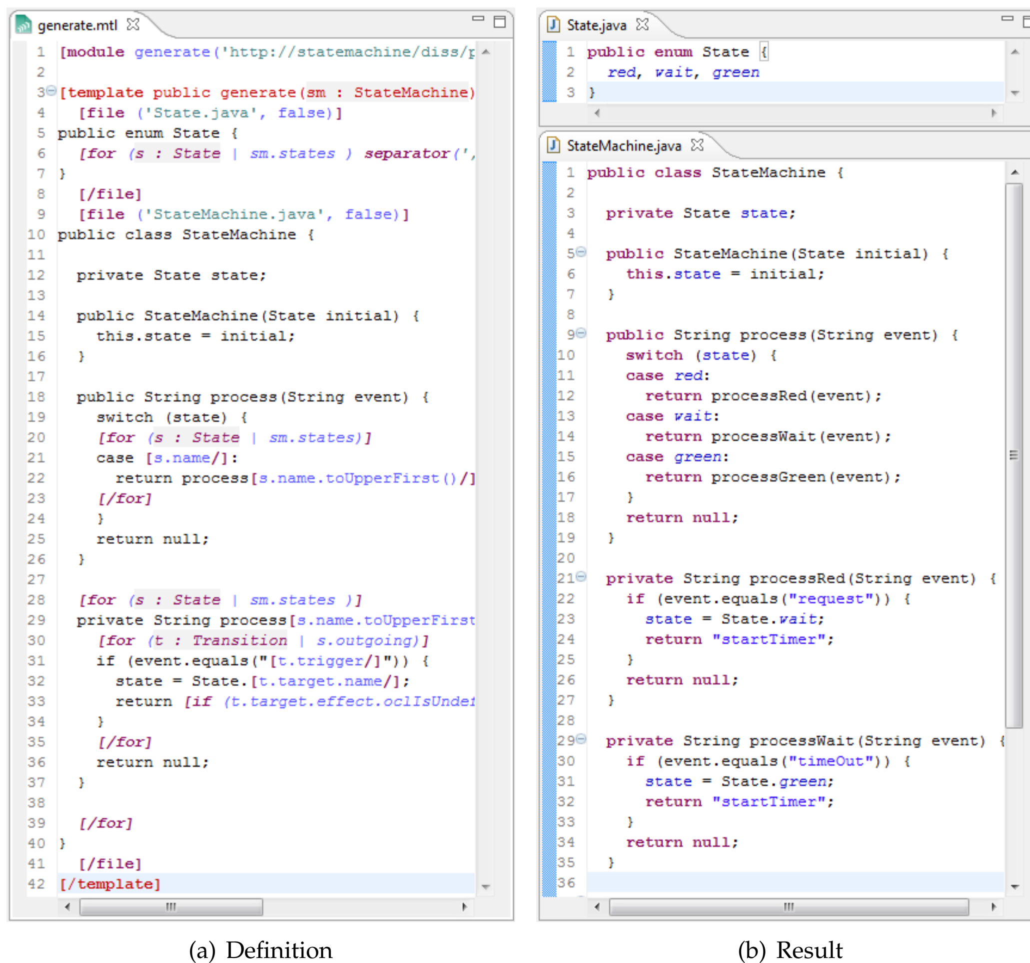
In practice, the semantics of a modeling language is usually not defined explicitly, but rather implicitly by building a code generator or simulator for the modeling language. A *code generator* generates executable code for the models created with the modeling language. A *simulator* allows the language users to step through the execution of the model built with the modeling language. EMF provides means to implement both a code generator and simulator for a modeling language.

**Code Generator.** In EMF, a code generator is specified using a model-to-text transformation language that maps a model specified with a modeling language to code. In the terminology introduced in Section 2.4.1 (*Semantics of a Modeling Language*), the code is the semantic domain and the model-to-text transformation is the semantic mapping. There are several languages for model-to-text transformation in EMF. Among these, the most widely known model-to-text transformation languages probably are Xpand[8] and Acceleo[9]. Whereas these languages provide different advanced features, they are both based on the same idea: a code generator is specified by means of templates which define how to assemble the code. In contrast to Xpand, Acceleo is an implementation of the OMG standard MOF Model to Text Transformation Language (MOF2Text) [Object Management Group, 2008a].

**Example 2.18** (Implementation of Code Generator)**.** *Figure 2.15(a) shows an implementation of a Java code generator for the state machine modeling language with Acceleo. A template enriches fragments of the target code with meta tags that express how to assemble the code. For example, there are meta tags to create files, to iterate over parts of the model, and to paste text from the model to the target code. For the state machine modeling language, we create a Java file for an enumeration of the state machine's states and a Java file to execute the state machine. In the second Java file, the template generates a constructor to set the* initial *state, and a method to* process *an event to change the current state of the state machine that delegates the processing to a method for the current state. The meta tags use expressions to navigate the model which are expressed in the Object Constraint Language (OCL) [Object Management Group, 2006b].*

*Figure 2.15(b) shows the result of applying the code generator to the model of the pedestrian traffic light. As specified by the template, two files are generated: one containing the enumeration of the states, and the other providing a class that implements the behavior of the state*

[8] see Xpand web site: `http://wiki.eclipse.org/Xpand`
[9] see Acceleo web site: `http://www.eclipse.org/acceleo/`




(a) Definition (b) Result

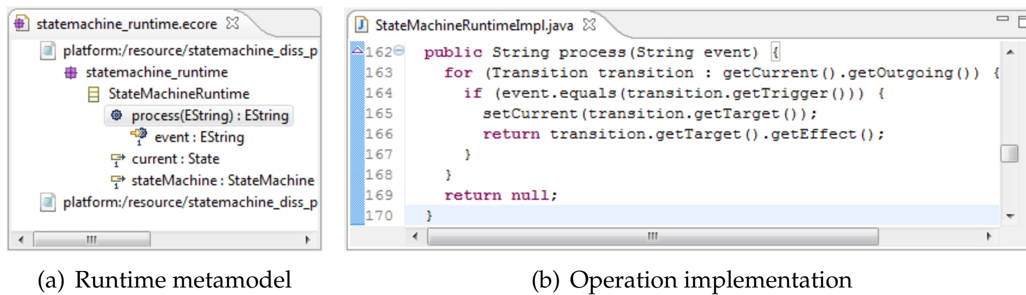
**Figure 2.15:** Semantics by code generation in EMF

*machine. To execute the state machine, the class needs to be instantiated providing the* initial *state and the* process *method needs to be invoked with the occurring events.*

**Simulator.** In EMF, a simulator is specified using a model-to-model transformation language that implements the execution steps as transformations applied to the runtime state of the model [Herrmannsdoerfer et al., 2009b]. The traces of state changes are thus the semantic domain, and the semantic mapping is defined by the model-to-model transformation. The metamodel needs to be extended to also provide constructs to maintain the runtime state of a model. There are several frameworks for implementing a simulator based on a language for model-to-model-transformation, e.g. Kermeta[10] [Muller et al., 2005], EProvide[11] [Sadilek and Wachsmuth, 2008], and the Model Execution Framework (MXF)[12] [Eichler et al., 2006]. Even though MXF is about to be contributed to EMF, there is not yet a standardized solution for building a simulator in EMF. In the following, we thus model the transformations as operations

[10] see Kermeta web site: `http://www.kermeta.org/`
[11] see EProvide web site: `http://eprovide.sourceforge.net/`
[12] see MXF web site: `http://wiki.eclipse.org/Model_Execution_Framework_(MXF)`

modifying the runtime state of the model, similar to Kermeta.




(a) Runtime metamodel (b) Operation implementation

**Figure 2.16:** Semantics by simulation in EMF

**Example 2.19** (Implementation of Simulator)**.** *Figure 2.16(a) displays the extension of the state machine metamodel to maintain the runtime state. A new class* StateMachineRuntime *is introduced that maintains the* current *state of a* stateMachine*. As the lower bound of* current *is 1, an initial state needs to be set, before the* stateMachine *can be executed. Additionally, an operation is introduced to* process *an* event*, change the* current *state and output an* action *based on the provided state machine model.*

*Figure 2.16(b) shows the implementation of the operation in Java. Besides the shown implementation of the simulator, frameworks like Kermeta, EProvide and MXF also provide more advanced features to control the simulation by pausing and setting breakpoints or to visualize the runtime state in the concrete syntax of the model during simulation.*

## 2.5 Evolution of Modeling Languages

Even though often neglected, software languages are subject to evolution due to changing requirements or technologies [Favre, 2005]. This holds for both general-purpose and domain-specific languages.

General-purpose languages evolve less frequently, since they are usually widely applied, and thus evolution has to be triggered by heavy-weighted committees. Nearly every well-known general-purpose programming language—e.g. Java, C#—evolves during its lifetime. UML [Object Management Group, 2009] is a well-known example for an evolving general-purpose modeling language.

Domain-specific languages can evolve more often, since they are usually only applied within an organization, and thus evolution can be performed in a more agile manner. Examples for evolving domain-specific modeling languages that have been built in our group are AutoFOCUS [Huber et al., 1996] for the domain of embedded systems, COLA (COmponent LAnguage) [Kugele et al., 2007] for the domain of automotive software, Service-ADL (Service Architecture Description Language) [Krüger et al., 2006] for the domain of service-oriented architectures, and QMM (Quality MetaModel) [Deissenboeck et al., 2007] for the domain of software quality.

Modeling languages are even more prone to change than programming languages, since it is often difficult to determine the right level of abstraction, as explained in

Section 2.1.3 (*The Quest for Abstraction*). This thesis focuses more on the evolution of domain-specific modeling languages, since they evolve more often and thus can be developed in a more evolutionary manner.

### 2.5.1 Reasons for Language Evolution

A modeling language may evolve due to a multitude of reasons. Like conventional software evolution, the reasons can be grouped according to the maintenance categories into perfective, corrective, preventive and adaptive maintenance [Lientz and Swanson, 1980].

**Perfective Maintenance** adds new features to enhance a software system. Looking at a modeling language, this means that new constructs are added to the modeling language. We expect that most of the changes fall into this category due to the typical way of developing a modeling language [Meyer, 1996]. At the beginning, a modeling language usually starts with a set of core constructs, expressive enough to specify certain software systems. Later, new constructs are added to the modeling language so that it can be used to specify more software systems or that certain properties of software systems can be specified more easily.

**Corrective Maintenance** corrects faults discovered in a software system. Like software, a modeling language may also be prone to errors and thus subject to corrective maintenance. The fault can be found in any of the constituents of a modeling language: the abstract syntax may not be restrictive enough to identify certain models as syntactically incorrect [Sadilek and Weißleder, 2008], the concrete syntax may not be able to recognize certain concrete representations, and the semantics may interpret certain models in an unexpected manner. These errors might be only found after building models for which the constituents are error-prone.

**Preventive Maintenance** refactors a software system to prevent faults in the future. Looking at a modeling language, this means that its constructs are simplified to make the modeling language easier to use and maintain. If more and more constructs are added to a modeling language due to perfective maintenance, it may turn out that the constructs are non-orthogonal and that combining them may lead to faults. In this case, preventive maintenance is necessary to integrate the constructs in a way that they are more orthogonal to each other, thus preventing such errors.

**Adaptive Maintenance** adapts a software system to a changing environment. The environment of a modeling language are the languages by means of which the modeling language is specified. These languages may also be subject to evolution due to the similar reasons. However, we expect that languages evolve less frequently than modeling languages, since their domain is much more stable [Favre, 2003]. Adaptive maintenance is also necessary, in case language engineers change one of the languages used for specifying the modeling language.

### 2.5.2 Metamodel and Semantics Evolution

To change a modeling language, its metamodel must be evolved. The metamodel is evolved by removing and adding nodes and edges in the metamodel. As a consequence, a metamodel change is witnessed by a source and a target metamodel. However, both metamodels share some commonality, since otherwise the metamodel change would not be an evolution, but rather a revolution. The degree of commonality can be expressed by the greatest common metamodel for both metamodels. We need the following definitions, before we can define a metamodel change:

**Definition 2.15** (Submodel)**.** *Let* $m_1 = (N_1, E_1, src_1, tgt_1, lab_1) \in \mathcal{M}$ *and* $m_2 = (N_2, E_2, src_2, tgt_2, lab_2) \in \mathcal{M}$ *be two models. A model* $m_1$ *is a submodel of another model* $m_2$ *if the following two functions exist:*

- *a total, injective function* $nm : N_1 \to N_2$ *which preserves identifiers and labels of nodes:*
$$\forall n_1 \in N_1 : n_1 = nm(n_1) \wedge lab_1(n_1) = lab_2(nm(n_1))$$
- *a total, injective function* $em : E_1 \to E_2$ *which preserves labels as well as source and target nodes of edges:*
$$\forall e_1 \in E_1 : lab_1(e_1) = lab_2(em(e_1)) \wedge src_2(em(e_1)) = nm(src_1(e_1)) \wedge tgt_2(em(e_1)) = nm(tgt_1(e_1))$$

*Then we write* $m_1 \subseteq m_2$*.*

Note that this definition does not require that the submodel of a model is part of the model. We can derive a submodel from a model by deleting nodes and edges. If we delete all nodes and edges, we obtain the empty model which is a submodel of all models:

**Definition 2.16** (Empty Model)**.** *The empty model is the model* $m_\emptyset = (N, E, src, tgt, lab)$ *that has neither nodes nor edges:*
$$N = \emptyset \wedge E = \emptyset$$

Since a metamodel is also a model, these definitions can be also applied to metamodels. We can thus determine whether a metamodel is a submetamodel of another metamodel as well as whether a metamodel is not-empty. Based on the submodel relation, we can define the greatest common metamodels for two metamodels:

**Definition 2.17** (Greatest Common Metamodel)**.** *Let* $mm_1, mm_2 \in \mathcal{MM}$ *be two metamodels. The greatest common metamodel* $mm \in \mathcal{MM}$ *for metamodels* $mm_1$ *and* $mm_2$ *is the metamodel that is the greatest submodel of both metamodels:*

$$mm \subseteq mm_1 \wedge mm \subseteq mm_2 \wedge \\ \forall mm' \in \mathcal{MM}, mm' \subseteq mm_1 \wedge mm' \subseteq mm_2 : mm' \subseteq mm$$

*Let* $gcmm(mm_1, mm_2) \in \mathcal{MM}$ *be the greatest common metamodel for two metamodels* $mm_1$ *and* $mm_2$*.*

We require the greatest common metamodel to conform to the metametamodel, since otherwise isolated nodes may be in the greatest common metamodel and thus it may only be empty in rare cases. The commonality between the source and target metamodel of a metamodel change can then be expressed by the greatest common metamodel:

**Definition 2.18** (Metamodel Change)**.** *A metamodel change is a mapping $mm_1 \mapsto mm_2$ from one metamodel $mm_1 \in \mathcal{MM}$ to another metamodel $mm_2 \in \mathcal{MM}$ in which both metamodels share a non-empty greatest common submetamodel $mm$:*

$$gcmm(mm_1, mm_2) \neq m_\emptyset$$

The metamodel change can also be expressed by removing nodes and edges from the source metamodel to the greatest common metamodel and by inverting the removals of nodes and edges from the greatest common metamodel to the target metamodel.

**Example 2.20** (Metamodel Change)**.** *Figure 2.17 illustrates a metamodel change that changes state machines from Moore to Mealy machines. In Moore machines, the* effect *of the state machine only depends on the current state [Moore, 1956]. In contrast, the* effect *of the state machine depends also on the* trigger *in Mealy machines [Mealy, 1976]. In the metamodel, we thus move the attribute* effect *from* State *to* Transition*. The figure also shows the greatest common metamodel between both metamodels, indicating that most of the metamodel stays the same in response to this metamodel change.*

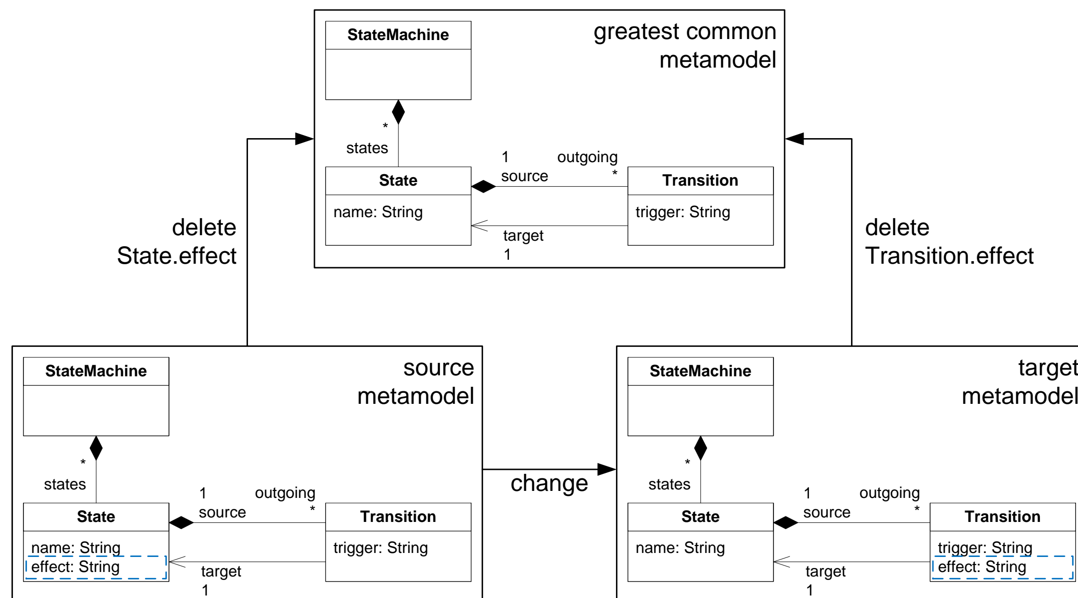



**Figure 2.17:** Moore to Mealy machines

The greatest common metamodel provides an indicator for the commonality between the source and target metamodel after a metamodel change. In case of an evolution, the greatest common metamodel is rather big compared to source and target metamodel; in case of a revolution, it is rather small.

When the metamodel is changed, also the semantics of the modeling language may need to be changed. Similar to a metamodel change, both versions of the semantics

have to share some commonality. This commonality can be expressed by a relation on elements of the two versions of the semantic domain:

**Definition 2.19** (Semantics Change)**.** *Let $mm_1 \mapsto mm_2$ be a metamodel change. A semantics change is a mapping $S_1 \mapsto S_2$ from semantics $S_1 : \mathcal{L}_{mm_1} \rightarrow \mathcal{SD}_1$ to semantics $S_2 : \mathcal{L}_{mm_2} \rightarrow \mathcal{SD}_2$ for which there is a left total relation $\cong \subseteq \mathcal{SD}_1 \times \mathcal{SD}_2$ which preserves the equivalence relation for both semantic domains:*

$$\forall s_1, s_2 \in \mathcal{SD}_1, s_3, s_4 \in \mathcal{SD}_2 : s_1 \equiv s_2 \wedge s_1 \cong s_3 \wedge s_3 \equiv s_4 \Rightarrow s_2 \cong s_4$$

In the most simple case, the semantic domain remains the same due to the semantics change, i.e. $\mathcal{SD}_1 = \mathcal{SD}_2$. In this case, the relation between the semantic domains is the equivalence relation on the semantic domain. In a more general case, a semantic domain can be embedded into a different semantic domain.

**Example 2.21** (Semantics Change)**.** *In response to the metamodel change from Moore to Mealy machines shown in Figure 2.17, we need to change the semantics. In the following, we use indices to distinguish the functions for metamodel version 2 from those of metamodel version 1. Since effects can no longer be accessed on states, we need to change formula* (2.3)*:*

$$T_2(s)(\langle e \rangle \circ es) \quad := \bigcup_{t \in activated_2(s,e)} \langle t.effect_2 \rangle \circ T_2(t.target_2)(es)$$

*The other formulas do not need to be changed. Moreover, we can use the same semantic domain for both language versions and thus we can use the equivalence relation to relate both semantics.*

### 2.5.3 Breaking Metamodel Changes

We can characterize changes to the metamodel according to their impact on existing models. In the literature, metamodel changes which destroy the conformance of a model to its metamodel are called breaking changes [Gruschko et al., 2007]:

**Definition 2.20** (Breaking Metamodel Change)**.** *A metamodel change $mm_1 \mapsto mm_2$ from $mm_1 \in \mathcal{MM}$ to $mm_2 \in \mathcal{MM}$ is called breaking if there is a model conforming to the original metamodel that no longer conforms to the adapted metamodel:*

$$\exists m \in \mathcal{M} : m \vDash mm_1 \wedge m \nvDash mm_2$$

*Otherwise, the metamodel change is called non-breaking.*

Changes between two metamodels can be expressed in a difference model which consists of a set of primitive changes [Cicchetti et al., 2008]. Similar to Sprinkle [Sprinkle, 2003] and Becker et al. [Becker et al., 2007], we characterize the primitive changes that can be derived from the metametamodel introduced in Section 2.2.4 (*Complete E-MOF Metametamodel*).

Table 2.1 classifies primitive metamodel changes into non-breaking (NB) and breaking (B) changes. We group the changes first according to the classes and second according to the features defined by the metametamodel. The kinds of changes that

can be effected on a feature depend on its multiplicity: Single-valued features support to modify a value, whereas multi-valued features support to add or remove a value. For the table, we assume that the changes really change the metamodel, e.g. the new value of a single-valued feature must be different from the old value. Sometimes the classification depends on the old or new value of the feature. In these cases, the condition for the classification is stated in the last column.

**Table 2.1:** Breaking and non-breaking metamodel changes

| Class | Feature | Change | NB | B | Condition |
|---|---|---|---|---|---|
| Metamodel | packages | add | • | | |
| | | remove | • | | package is empty |
| | | | | • | package is not empty |
| Package | name | modify | | • | |
| | subPackages | add | • | | |
| | | remove | • | | package is empty |
| | | | | • | package is not empty |
| | types | add | • | | |
| | | remove | • | | type is a primitive type |
| | | | | • | type is a class |
| Type | name | modify | | • | |
| Enumeration | literals | add | • | | |
| | | remove | • | | enumeration is not used |
| | | | | • | enumeration is used |
| Literal | name | modify | • | | enumeration is not used |
| | | | | • | enumeration is used |
| Class | abstract | modify | • | | new value is false |
| | | | | • | new value is true |
| | features | add | • | | new feature is optional |
| | | | | • | new feature is mandatory |
| | | remove | | • | |
| | superTypes | add | • | | new superclass has no mandatory feature |
| | | | | • | new superclass has mandatory features |
| | | remove | • | | superclass defines no features |
| | | | | • | superclass defines features |
| Feature | name | modify | | • | |
| | lowerBound | modify | • | | lower bound is decreased |
| | | | | • | lower bound is increased |
| | upperBound | modify | • | | upper bound is increased |
| | | | | • | upper bound is decreased |
| Attribute | id | modify | • | | new value is false |
| | | | | • | new value is true |
| | type | modify | | • | |
| Reference | composite | modify | | • | |
| | type | modify | • | | new value is superclass of old value |
| | | | | • | new value is not superclass of old value |
| | opposite | modify | • | | new value is null |
| | | | | • | new value is not null |

The classification is independent of the models that are actually built with the metamodel. Moreover, it is rather too strong than too weak: for example, changes to a class can only break models, in case the class or any of its subclasses can be instantiated. For composite references defined by the metametamodel, there would also be a change to move a metamodel element to another parent. However, a move can be modeled as a remove and add in the classification, where the remove is usually

breaking. Changes of names are usually breaking, since the models refer by name to the instantiated metamodel elements. The change of a package name also affects objects of the classes contained in the package, since the objects refer to the class by its fully qualified name. Note that a feature is optional if its lower bound is 0, and mandatory if the lower bound is at least 1.

### 2.5.4 Model Migration

If both the syntax and semantics of existing models are preserved after a metamodel change, we call the change a backwards-compatible change:

**Definition 2.21** (Backwards-Compatible Metamodel Change). *Let $mm_1 \mapsto mm_2$ be a metamodel change from metamodel $mm_1 \in \mathcal{MM}$ to $mm_2 \in \mathcal{MM}$, and $S_1 \mapsto S_2$ be the appropriate semantics change from semantics $S_1 : \mathcal{L}_{mm_1} \to \mathcal{SD}_1$ to semantics $S_2 : \mathcal{L}_{mm_2} \to \mathcal{SD}_2$. The change is called backwards-compatible if*

- *all models conforming to metamodel $mm_1$ also conform to metamodel $mm_2$:*

$$\mathcal{L}_{mm_1} \subseteq \mathcal{L}_{mm_2}$$

- *the meaning of all models is preserved:*

$$\forall m \in \mathcal{L}_{mm_1} : S_1(m) \cong S_2(m)$$

As a consequence, in response to a backwards-compatible metamodel change, the existing models can still be used with the evolved modeling language.

**Example 2.22** (Backwards-Compatible Metamodel Change). *Figure 2.18 illustrates a backwards-compatible metamodel change of the state machine modeling language that introduces hierarchical states to be able to structure complex state machines.*

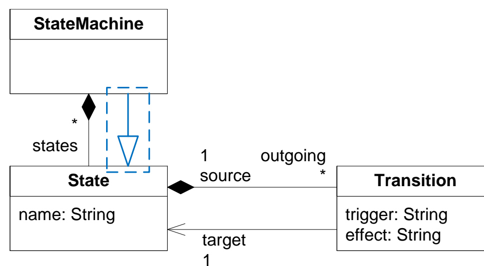



**Figure 2.18:** Introduction of hierarchical states

Metamodel Change. *In the metamodel, we thus make* State *a super type of* StateMachine*. Now, a* State *can also be a* StateMachine *which can again be decomposed into a number of* states*, and so on. The changes are highlighted by colored and dashed boxes in Figure 2.18. The existing flat state machine models still conform to the evolved metamodel, since the metamodel change only adds the possibility to build hierarchical state machines, but does not affect the existing models. To be able to extend the semantics, we need the helper function $parent_2 : State_2 \to StateMachine_2 \cup \{\bot\}$ which returns the parent of a state $s \in State_2$:*

$$s.parent_2 = sm \in StateMachine_2 :\Leftrightarrow Edge(sm, s, \textsf{states})$$

*Since* **states** *is a composite reference, there might be at most one parent state machine for each state.*

Semantics Change. *In response to the metamodel change, we also need to update the semantics. Since there might be several instances of* **StateMachine** *now, we have to update formula* (2.1) *to return the instance of* **StateMachine** *which does not have a parent state machine:*

$$statemachine_2(m) = sm \in StateMachine_2 :\Leftrightarrow sm.parent_2 = \bot$$

*Due to hierarchical states, we have to update formula* (2.2) *to also take the transitions of parent states into account for transitions activated in a state* $s \in State_2$*:*

$$activated_2(s, e) \;:=\; \bigcup_{p \in s.parent_2^*} \{t \in p.outgoing_2 \mid t.trigger_2 = e\}$$

*where* $^*$ *is the transitive closure. The other formulas do not need to be adapted and thus stay the same. Note that the formulas are polymorphic, i.e.* $T_2(s, es)$ *in formula* (2.3) *is automatically redirected to formula* (2.4)*, in case* $s \in StateMachine_2$*. We can show that the change preserves the meaning of existing flat state machines:*

$$\begin{aligned} activated_2(s, e) \;&:=\; \bigcup_{p \in s.parent_2^*} \{t \in p.outgoing_2 \mid t.trigger_2 = e\} \\ &= \bigcup_{p \in s.parent_2^*} activated_1(p, e) \\ &= \bigcup_{p \in \{s\}} activated_1(p, e) = activated_1(s, e) \end{aligned}$$

*Since all other formulas remain the same, they return the same stream of actions for the same stream of events.*

In case a metamodel change is not downwards-compatible, existing models need to be migrated so that they can be used with the evolved modeling language. A model migration needs to fulfill a number of properties so that the migrated models can be used with the evolved modeling language. Sprinkle distinguishes syntax-and semantics-preserving model migration [Sprinkle, 2003], according to the two constituents of a model that might be invalidated due to language evolution:

**Definition 2.22** (Syntax-Preserving Model Migration)**.** *Let* $mm_1 \mapsto mm_2$ *be a metamodel change from metamodel* $mm_1 \in \mathcal{MM}$ *to* $mm_2 \in \mathcal{MM}$*. Let* $\mathcal{L}_{mm_1}$ *and* $\mathcal{L}_{mm_2}$ *be the modeling languages that are defined by these metamodels. A syntax-preserving model migration is a function that transforms models that conform to metamodel* $mm_1$ *to models that conform to metamodel* $mm_2$*:*

$$mig : \mathcal{L}_{mm_1} \rightarrow \mathcal{L}_{mm_2}$$

A syntax-preserving model migration thus guarantees that the migrated models are syntactically correct in the new language version, if the original models were syntactically correct in the old language version. Note that syntax preservation does not mean that the syntax of a model stays the same, but that the model stays syntactically correct through the migration.

**Example 2.23** (Syntax-Preserving Model Migration)**.** *To be syntax-preserving, a model migration for the change from Moore to Mealy machines (see Figure 2.17) just needs to remove the effects from the states. Since effects are not required for states, models migrated in that way are syntactically correct. However, in general, their meaning is not preserved, as the interpretation of these models does no longer produce outputs:*

$$\begin{aligned} T_1(s)(\langle e \rangle \circ es) &= \bigcup_{t \in activated_1(s,e)} \langle t.target_1.effect_1 \rangle \circ T_1(t.target_1)(es) \\ &= \bigcup_{t \in activated_1(s,e)} \langle \rangle \circ T_1(t.target_1)(es) = \{\langle \rangle\} \end{aligned}$$

*The last equality can be shown by induction over the stream of input events.*

To be able to guarantee that the meaning of a model is not altered during migration, a model migration needs to be semantics-preserving:

**Definition 2.23** (Semantics-Preserving Model Migration)**.** *Let $mm_1 \mapsto mm_2$ be a metamodel change from metamodel $mm_1 \in \mathcal{MM}$ to $mm_2 \in \mathcal{MM}$, and $S_1 \mapsto S_2$ be the appropriate semantics change from semantics $S_1 : \mathcal{L}_{mm_1} \to \mathcal{SD}_1$ to semantics $S_2 : \mathcal{L}_{mm_2} \to \mathcal{SD}_2$. A syntax-preserving model migration $mig : \mathcal{L}_{mm_1} \to \mathcal{L}_{mm_2}$ is called semantics-preserving if*

$$\forall m \in \mathcal{L}_{mm_1} : S_1(m) \cong S_2(mig(m))$$

A semantics-preserving model migration thus guarantees that the migrated models have the same semantics in the new language version as the original models in the old language version.

**Example 2.24** (Semantics-Preserving Model Migration)**.** *A semantics-preserving model migration for the transition from Moore to Mealy machines (see Figure 2.17) needs to preserve the effects. To be semantics-preserving, a model migration for the evolution from Moore to Mealy machines needs to move the effects appropriately from states to transitions. However, there is a well-known algorithm to convert Moore to Mealy machines that can be used to implement a semantics-preserving model migration.*

The commonality between the source and target metamodel is relevant for model migration. Usually, the part of the model that conforms to the greatest common metamodel does not need to be modified during migration. The part of the model that is deleted to derive the greatest common metamodel from the source metamodel no longer conforms to the target metamodel and thus needs to be removed. However, the removed information is usually migrated to the part that is created to obtain the target metamodel from the greatest common metamodel. In summary, the effort for model migration should depend on the size of the differences between the source and target metamodel, and not on the size of the metamodel versions [Sprinkle, 2003].

### 2.5.5 Model Transformation for Model Migration

The model migration can be manually specified as a model transformation which transforms the model conforming to the old metamodel to a model conforming to the evolved metamodel [Cicchetti et al., 2008]. Figure 2.19 shows the typical architecture

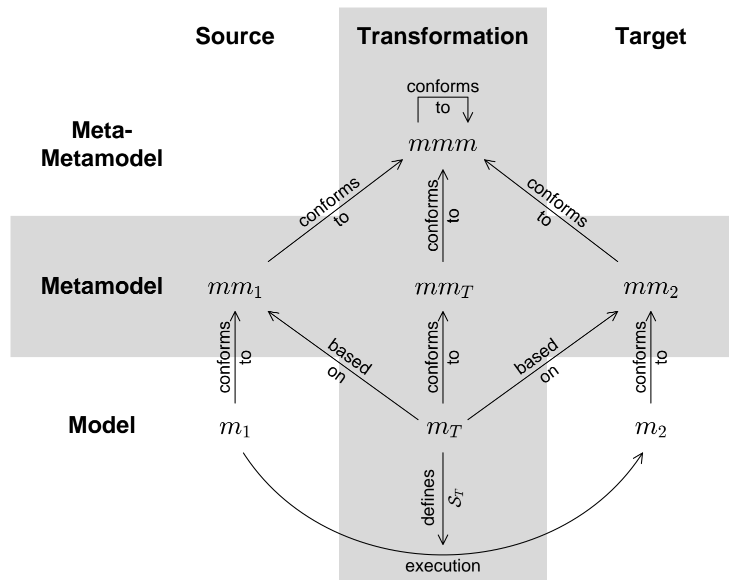



**Figure 2.19:** Model transformation

to define and execute model transformations. We define a model transformation similar to [Mens and Van Gorp, 2006]:

**Definition 2.24** (Model Transformation)**.** *A model transformation is a function* $t : \mathcal{M} \rightarrow \mathcal{M}$ *that maps a source model* $m_1 \in \mathcal{M}$ *to a target model* $m_2 \in \mathcal{M}$*.*

A model transformation is usually defined using a transformation language:

**Definition 2.25** (Model Transformation Language)**.** *A model transformation language is a modeling language* $\mathcal{L}_T = \mathcal{L}_{mm_T}$ *with metamodel* $mm_T \in \mathcal{MM}$ *and semantics* $S_T : \mathcal{L}_T \rightarrow \{\mathcal{M} \rightarrow \mathcal{M}\}$ *that maps transformation models* $m_T \in \mathcal{L}_T$ *to model transformations* $S_T(m_T) : \mathcal{M} \rightarrow \mathcal{M}$*.*

The transformation definition defines a model transformation in a transformation language based on the source and target metamodel of the transformation:

**Definition 2.26** (Transformation Definition)**.** *A transformation definition* $m_T \in \mathcal{L}_T$ *is a model that is defined using a transformation language* $\mathcal{L}_T$*. The transformation definition defines a model transformation* $t : \mathcal{M}_{mm_1} \rightarrow \mathcal{M}_{mm_2}$ *based on the source metamodel* $mm_1 \in \mathcal{MM}$ *and the target metamodel* $mm_2 \in \mathcal{MM}$*.*

We can also specify a model transformation that consumes or produces a transformation definition [Tisi et al., 2009]:

**Definition 2.27** (Higher-Order Model Transformation)**.** *A higher-order model transformation is a model transformation whose source and/or target models are themselves transformation definitions.*

Braun and Marschall formalize in more detail the syntax and semantics of the model transformation language BOTL (Bidirectional Object-oriented Transformation Language) [Braun and Marschall, 2003, Marschall, 2005]. Based on this formalization,

they can ensure that BOTL transformation definitions are syntax-preserving and bidirectional, i.e. can be executed in both directions. Bidirectional transformation languages are particularly suited to define transformations that exchange models between model-based development tools [Braun, 2003, Braun, 2004].

Besides BOTL, there are already quite a number of languages for specifying model transformations. The most prominent transformation languages are Query View Transformation (QVT) [Object Management Group, 2008b] for MOF, or the ATLAS transformation language (ATL) [Jouault and Kurtev, 2006] for EMF. Besides bidirectionality, model transformation languages can be classified according to a number of other categories [Mens and Van Gorp, 2006, Czarnecki and Helsen, 2006]. In the following, we are interested in the categories that are important for model migration.

**Metamodel Layer.** A model transformation is defined on the metamodel layer between the source and target metamodel. We distinguish exogenous and endogenous model transformation, depending on whether source and target metamodel of the transformation are different or not [Mens and Van Gorp, 2006, Czarnecki and Helsen, 2006]. *Exogenous* model transformation requires to specify the mapping of all elements from the source to the target metamodel. As typically only a subset of metamodel elements are modified by the metamodel evolution, a model migration specified as an exogenous transformation contains a high fraction of identity rules. The fraction of the identity rules is proportional to the size of the intersection between the two metamodel versions.

**Example 2.25** (Migrating Transformation Definition)**.** *To illustrate this, Figure 2.20 shows an ATL implementation of the migration for the metamodel change from Moore to Mealy machines as shown in Figure 2.17. An ATL transformation definition consists of a set of rules each of which maps elements from the source metamodel to elements from the target metamodel. With the exception of the highlighted line, the definition specifies an identity transformation which covers exactly the greatest common metamodel of the two metamodel versions as presented in Example 2.20.*

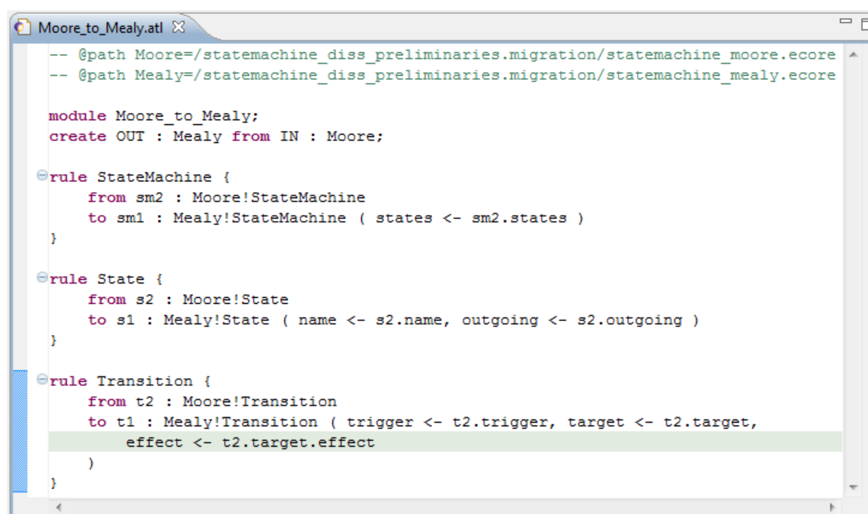



**Figure 2.20:** Model migration defined in ATL

Concerning this aspect, *endogenous* transformation is better suited to the nature of

model migration, as it only has to address those metamodel elements for which the model needs to be modified. However, endogenous transformation requires the source and the target metamodel to be the same which is not the case for metamodel evolution. Hence, conventional languages for model transformation are not well suited to specify a model migration [Sprinkle, 2003].

**Model Layer.** A model transformation is executed on the model layer, consuming a source model and producing a target model. We distinguish out-of-place and in-place model transformation, depending on whether the source and target model of the transformation are different or not [Mens and Van Gorp, 2006, Czarnecki and Helsen, 2006]. *Out-of-place* model transformation creates the target model from scratch by executing the transformation definition. Since a model migration usually only needs to transform a rather small part of the model, out-of-place transformation may be rather slow for transforming large models. Concerning this aspect, *in-place* model transformation is better suited for model migration, as it directly updates the source model so that it becomes the target model. However, in-place transformation is usually only possible for endogenous transformation, in order to assure that the target model conforms to the target metamodel. Hence, conventional languages for model transformation are not well suited to execute a model migration.

## 2.6 Summary

In this chapter, we gave an overview of the definition, implementation and evolution of modeling languages. A modeling language consists of abstract syntax, concrete syntax and semantics. The abstract syntax can be defined by a metamodel that is specified using an appropriate metamodeling language. The metamodel can be interpreted by a modeling framework to provide an implementation for the modeling language. Modeling languages are subject to evolution, requiring the migration of models already defined with the modeling language. Model migration is a special case of model transformation, in which only a small part of the metamodel changes.

Chapter 3

# State of the Practice: Automatability of Model Migration

Automating model migration in response to metamodel adaptation promises to substantially reduce effort. Unfortunately, little is known about the types of changes occurring during metamodel adaptation in practice and, consequently, to which degree reconciling model migration can be automated. In an empirical study, we analyzed the changes that occurred during the evolution history of two industrial modeling languages and classified them according to their level of potential automation. Based on the results, we present a list of requirements for tools effectively supporting model migration in practice. This chapter is partly based on [Herrmannsdoerfer et al., 2008a].

**Contents**


**3.1 State of the Art** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

**3.2 Classification of Metamodel Changes** . . . . . . . . . . . . . . . . . . 67

**3.3 Study Design** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72

**3.4 Study Implementation** . . . . . . . . . . . . . . . . . . . . . . . . . . 75

**3.5 Requirements for Automating Model Migration** . . . . . . . . . . 78

**3.6 Summary** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78


In Section 3.1 (*State of the Art*), we motivate the study by analyzing existing approaches and case studies. The proposed classification for the automatability of the model migration is presented in Section 3.2 (*Classification of Metamodel Changes*). In Section 3.3 (*Study Design*), we outline the setup, and in Section 3.4 (*Study Implementation*), the results of the study we performed on the histories of two industrial metamodels. In Section 3.5 (*Requirements for Automating Model Migration*), we discuss the results and derive requirements for efficient tool support. The study is concluded in Section 3.6 (*Summary*) with the implications for our approach.

## 3.1 State of the Art

Work related to this empirical study are either approaches trying to automate model migration or case studies conducted to evaluate the approaches.

**Approaches.** Different kinds of approaches [Rose et al., 2009] have been proposed to automate the migration of models in reponse to metamodel evolution.

*Manual specification* approaches extend model transformation languages with a means to automatically copy model elements that are not affected by the metamodel evolution. Examples are Sprinkle's language [Sprinkle and Karsai, 2004] and its successor, the Model Change Language (MCL) [Narayanan et al., 2009], as well as Flock [Rose et al., 2010d]. Manual specification approaches require most effort to specify model migrations, but allow their users to express complex migrations.

*Operation-based* approaches provide reusable coupled operations which are used to adapt the metamodel and which also encapsulate a model migration. An example is Wachsmuth's approach [Wachsmuth, 2007] which presents a set of reusable operations and classifies them according to instance preservation properties. Operation-based approaches reduce the effort by reusing recurring migrations, but require effort to choose the coupled operations.

*Matching* approaches try to automatically derive a model migration from the difference between two metamodel versions. Examples are Gruschko's proposal [Gruschko et al., 2007], Cicchetti's approach [Cicchetti et al., 2008] and the Atlas Matching Language (AML) [Garcés et al., 2009]. Matching approaches provide most automation, but are usually not able to derive complex migrations.

Existing approaches to support model migration mainly differ in the provided level of automation, expressiveness, reuse of migration knowledge and preservation properties. Little is known on the combination of capabilities that best supports the requirements faced during development and maintenance of metamodels and models in practice.

**Case Studies.** Besides modelware where metamodels evolve [Favre, 2003], language evolution affects other technical spaces [Kurtev et al., 2002]: database schemas evolve in dataware [Meyer, 1996], grammars evolve in grammarware [Klint et al., 2005], and APIs evolve [Dig and Johnson, 2006], too. Compared to the large number of approaches in these technical spaces [Rahm and Bernstein, 2006], there are only few case studies that evaluate the effectiveness of the presented approaches.

In *dataware*, Sjøberg measures the primitive changes and their impact on existing applications in the evolution of a health management system [Sjøberg, 1993]. Curino et al. reverse engineered the evolution of the Wikipedia schema as a sequence of coupled operations and propose it as a benchmark for other approaches [Curino et al., 2008a]. Lerner applies her matching approach TESS to real-life and artificial schema evolution examples to show that it is able to detect compound changes [Lerner, 2000].

In *grammarware*, Staudt et al. evaluated their matching approach TransformGen with

67 changes of ARL, a language for writing semantic routines [Garlan et al., 1994]: 18 changes could be detected automatically, 30 needed to be customized and 19 could not be detected. Lämmel applied operations to repair a COBOL grammar automatically recovered from an informal language specification [Lämmel and Verhoef, 2001]. He used the same operations to define correspondences between the different versions of the Java grammar [Lämmel and Zaytsev, 2009b]. Overbey and Johnson show that the whole evolution of Fortran and Java can be supported by refactorings for migrating programs [Overbey and Johnson, 2009].

In *APIware*, Dig and Johnson showed by studying the evolution of 5 APIs that over 80% of the evolution can be covered by reusable refactoring operations [Dig and Johnson, 2006].

In *modelware*, Geest et al. apply their matching approach to detect primitive changes to the WSSF metamodel [Geest et al., 2008]. Garcés et al. evaluate their customizable matching approach by detecting compound changes to the well-known Petri net example as well as an evolution of the Java metamodel [Garcés et al., 2009].

The mentioned case studies either do not measure automatibility of the model migration or are not independent of the approach that is applied. In contrast, our goal is to measure automatability in a way to be able to derive requirements for an approach to best support model migration in practice.

## 3.2 Classification of Metamodel Changes

In this section, we introduce a classification that allows us to determine how far model migration can be automated. Usually the metamodel is adapted manually, and models are migrated at different levels of automation. The first level of automation is to define a transformation that is able to automatically migrate a single model. A higher level of automation is achieved, if a single transformation definition can be used to migrate all existing models of a metamodel. When manually specifying such transformations, one discovers that they contain recurring patterns. Thus, the third level of automation corresponds to the application of higher-order transformations embodying such recurring patterns that automate both metamodel adaptation and model migration. In order to define the levels of automation, we introduce the notion of a coupled change:

**Definition 3.1** (Coupled Change)**.** *A coupled change is a combination of a metamodel change and the semantics-preserving migration of the models conforming to that metamodel.*

Coupled changes do not comprise metamodel changes that do not require a migration of models:

**Definition 3.2** (Metamodel-Only Change)**.** *A metamodel change is called metamodel-only if the metamodel change is backwards-compatible, i.e. preserves both abstract syntax and semantics of all models.*

In the following, we introduce the individual classes in combination with representative examples, working our way up from lower to higher levels of potential automation. Figure 3.1 depicts an overview of the classification. For each class of coupled

changes, the figure indicates to which level they are specific: The higher the level on which a coupled change depends, the more can be reused and therefore automated. A model-specific coupled change can only be used for a subset of the models of a metamodel. A metamodel-specific coupled change provides automation for all models of a metamodel. A metamodel-independent coupled change can be even applied to all metamodels and their models.

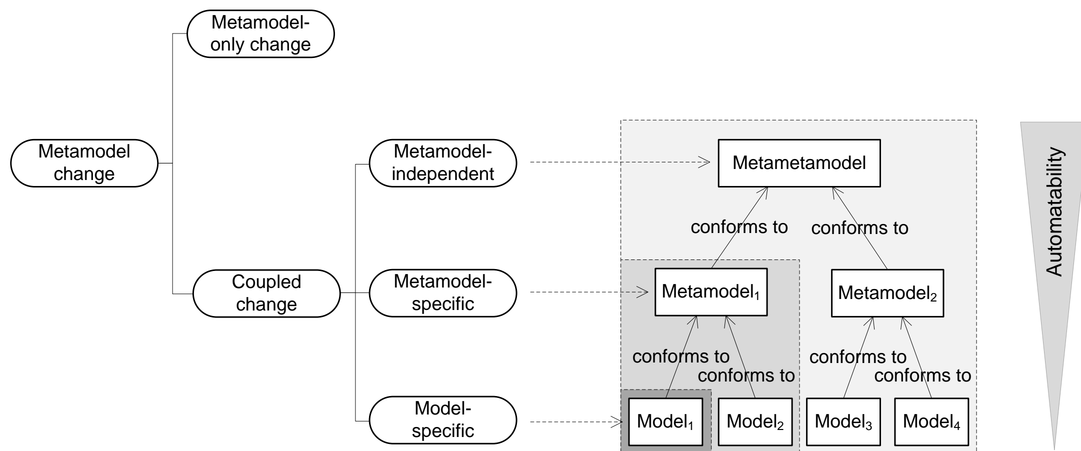


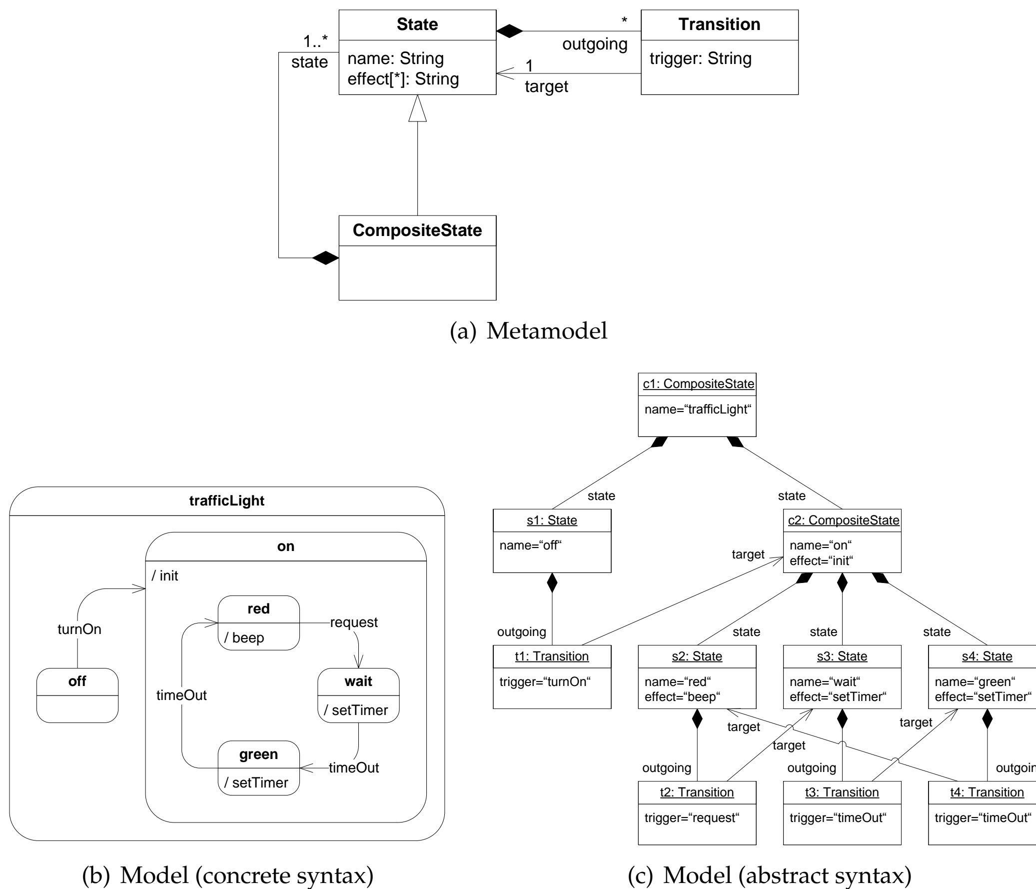

**Figure 3.1:** Automatibility of metamodel changes

### 3.2.1 Running Example

We use a simple modeling language for hierarchical state machines to illustrate our classification. Figure 3.2 depicts the metamodel and a corresponding model in both concrete and abstract syntax.

**Metamodel.** A State may be decomposed into sub states through its subclass CompositeState. A Transition has a target state and relates to its source state through the outgoing composition. A transition is activated by a trigger. When a state is entered, a sequence of actions is performed as effect. Strings are used for state names and to denote triggers and effects. The root element of a state machine model is of type CompositeState.

**Model.** The model describes the simplified behavior of a controller for a pedestrian traffic light and uses all the constructs defined by the metamodel. The authorities can turn on the traffic light, which gets initialized as a result. The transition to turn it off is omitted so that the model remains simple. When the traffic light is red and a pedestrian requests a green phase, the controller transitions to wait and activates a timer (setTimer). When its timeOut occurs, the controller transitions to green and activates the timer again. When its timeOut occurs, the controller returns to state red and notifies pedestrians with a beep.




(a) Metamodel

(b) Model (concrete syntax)

(c) Model (abstract syntax)

**Figure 3.2:** State machine example

### 3.2.2 Model-Specific Coupled Change

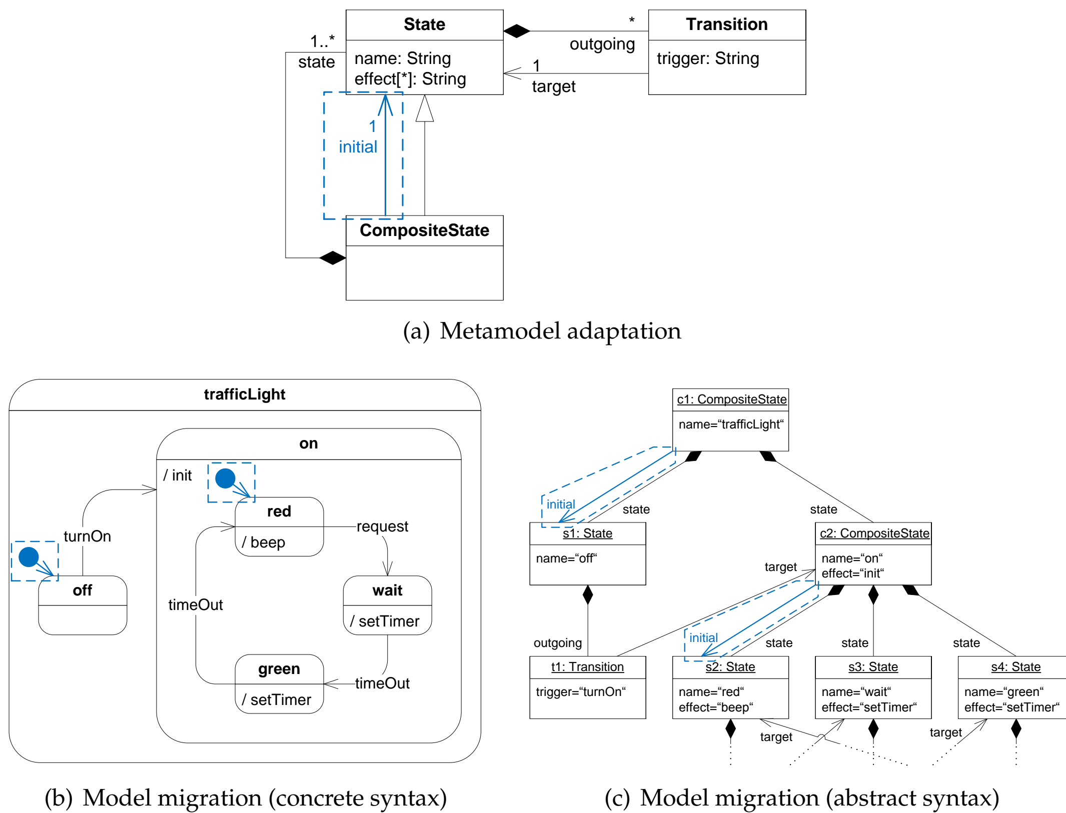
A coupled change is called model-specific if the migrating transformation is specific to a restricted set of models and thus cannot be reused to migrate different models of the same metamodel:

**Definition 3.3** (Model-Specific Coupled Change)**.** *A coupled change is called model-specific if its migration cannot be specified as a single transformation definition on the metamodel level that can be applied to migrate all models of a metamodel.*

This happens when the specification of a migration requires information which varies from model to model.

**Example 3.1** (Model-Specific Coupled Change)**.** *Figure 3.3[1] depicts both metamodel and model after an example of a model-specific coupled change. In the metamodel, the reference* initial *is introduced to denote the initial state within a composite state. As this reference is mandatory, the model needs to be migrated to add the missing information. However, the initial states cannot be inferred from the information already available in the model, and default values cannot be provided either. Therefore, model-specific information is required to be able to completely specify the migration.*

[1]For better overview, modified elements are highlighted by dashed boxes.




(a) Metamodel adaptation

(b) Model migration (concrete syntax)

(c) Model migration (abstract syntax)

**Figure 3.3:** Introduction of initial states

### 3.2.3 Model-Independent, Metamodel-Specific Coupled Change

When a coupled change is not model-specific and all models of a metamodel can be automatically migrated, it is called model-independent:

**Definition 3.4** (Model-Independent Coupled Change)**.** *A coupled change is called model-independent if its migration can be specified as a single transformation definition on the metamodel level that can be applied to migrate all models of a metamodel.*

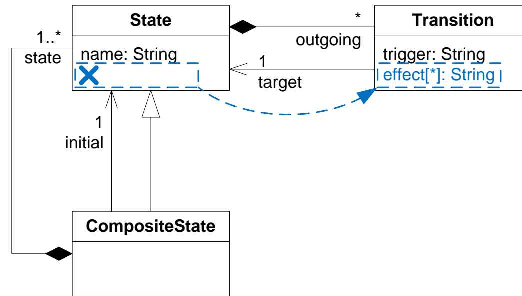
Model-independent coupled changes can be further subclassified. If the change is specific to a certain metamodel, it is called metamodel-specific and cannot be generated using a higher-order model transformation:

**Definition 3.5** (Metamodel-Specific Coupled Change)**.** *A coupled change is called metamodel-specific if its migration cannot be specified as a transformation definition that can be produced from the metamodel using a higher-order model transformation defined on the metametamodel level.*

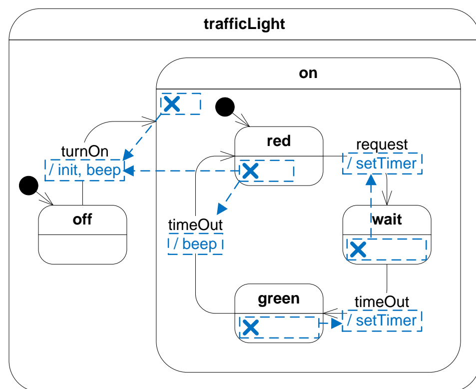
In that case, the reuse of the coupled change across different metamodels makes no sense.

**Example 3.2** (Metamodel-Specific Coupled Change)**.** *Figure 3.4 depicts the impact of a metamodel-specific coupled change, which changes the state machine from a Moore to a Mealy machine. In the metamodel, the attribute* effect *is moved from* State *to* Transition*. As*

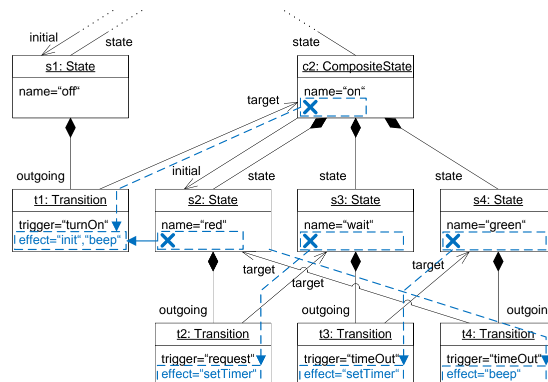
*states are no longer allowed to specify an* **effect***, the model no longer conforms to the modified metamodel. To reconcile the model with the metamodel, the effect has to be moved from all states to their incoming transitions. In the presence of hierarchical states, the migration is however more involved, as we also have to take the effect of the initial states into account for composite states. As the migration is rather specific and therefore not likely to recur very often, it makes no sense to reuse this coupled change across metamodels.*




(a) Metamodel adaptation




(b) Model migration (concrete syntax)




(c) Model migration (abstract syntax)

**Figure 3.4:** Moore to Mealy machine

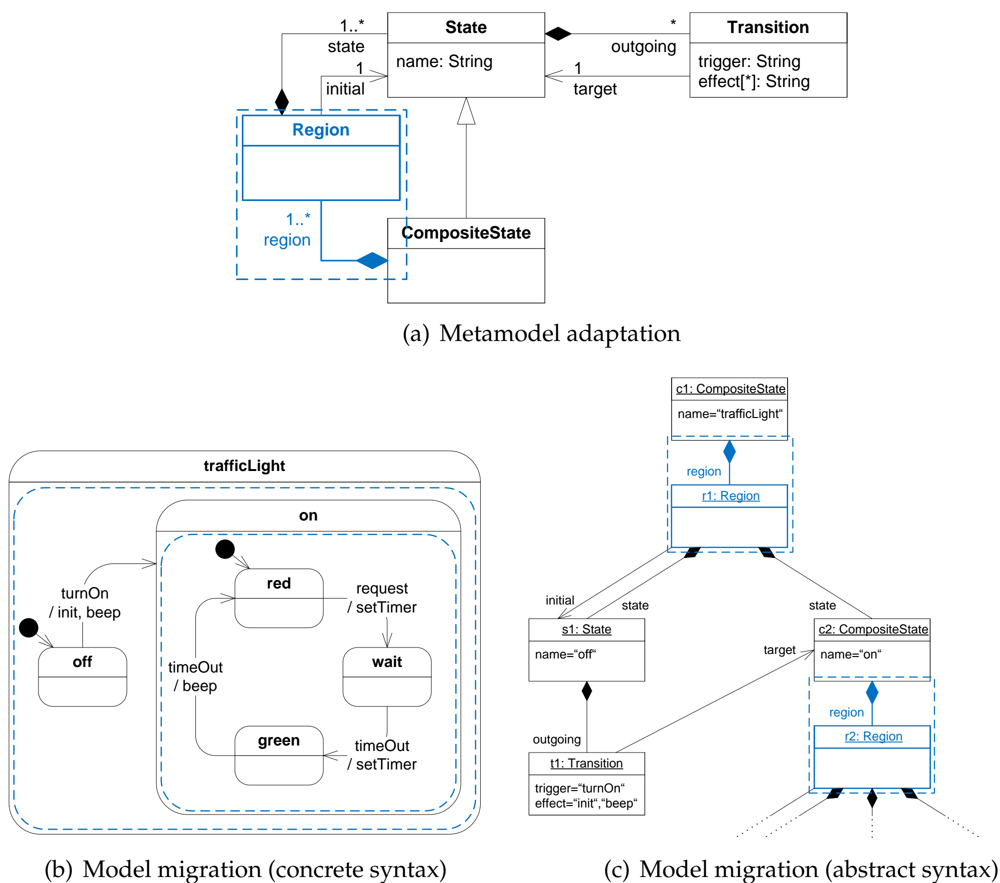
### 3.2.4 Metamodel-Independent Coupled Change

If both metamodel adaptation and model migration do not depend on the metamodel's domain, the coupled change is called metamodel-independent and can be expressed in a generic manner.

**Definition 3.6** (Metamodel-Independent Coupled Change)**.** *A coupled change is called metamodel-independent if its migration can be specified as a transformation definition that can be produced from the metamodel using a higher-order model transformation defined on the metametamodel level.*

If a metamodel-independent coupled change is likely to recur in the evolution of different metamodels, it makes sense to generalize it to a high-order model transformation that can be reused to evolve other metamodels.

**Example 3.3** (Metamodel-Independent Coupled Change)**.** *Figure 3.5 depicts the impact of a metamodel-independent coupled change, which is a first step to introduce the concept of concurrent regions to the modeling language. In the metamodel, the class* Region *is introduced as a container of the directly contained sub states within a* CompositeState*. To compensate the change in a model, the migration creates a* Region *as child of each* CompositeState *and moves all directly contained sub states to the newly created* Region*. A possible generalization of this coupled change—which is known as* Extract Class *in object-oriented refactoring [Fowler, 1999]—extracts a collection of features of one class into a new class which is accessible from the old class via a single-valued composition to the new class. For the introduction of concurrent regions, the single-valued composition needs to be generalized as a next step to allow for multiple regions.*



(a) Metamodel adaptation

(b) Model migration (concrete syntax)

(c) Model migration (abstract syntax)

**Figure 3.5:** Introduction of concurrent regions

## 3.3 Study Design

In order to asses the potential for automation in practice, we applied the classification to the histories of two industrial metamodels. In this section, we present the design of this study: the goals, the input and the method of the study.

### 3.3.1 Study Goal

The study was performed to answer which fraction of metamodel changes

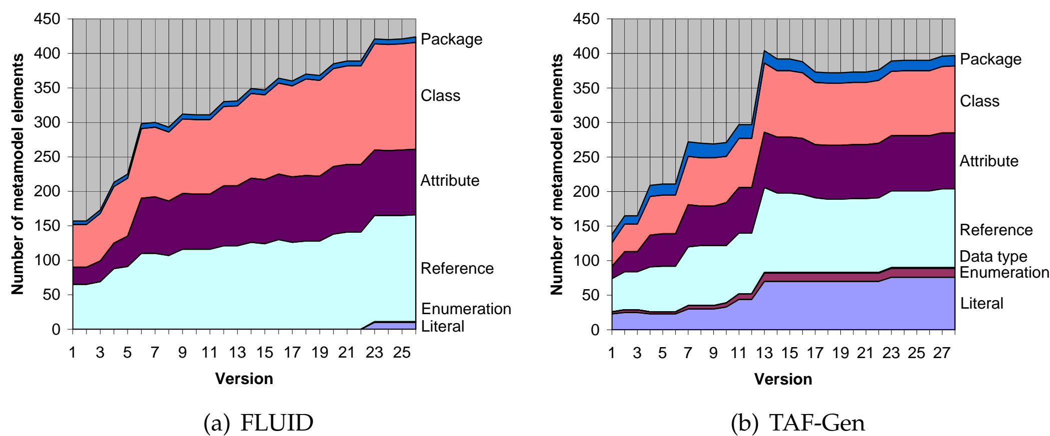
- **RQ1.** *is metamodel-only and thus requires no model migration?* Since metamodel-only changes do not require a model migration, they have to be regarded separately.
- **RQ2.** *is model-specific and thus defies automation of migration?* If there are model-specific coupled changes, the model migration cannot be fully automated.
- **RQ3.** *is metamodel-independent and thus generalizable across metamodels?* The higher the fraction of metamodel-independent coupled changes, the higher the degree of potential automation for the model migration.

### 3.3.2 Study Object

Two industrial metamodels from BMW Car IT[2] were chosen as input. Both metamodels were developed and maintained by several persons.

**FLUID** (FLexible User Interface Development) is a framework for rapid prototyping of human machine interfaces in the automotive domain [Hildisch et al., 2007]. A metamodel defines a modeling language that enables the abstract specification of a human machine interface. An executable prototype of the human machine interface can be generated from a model written in that language.

To get a better impression of the evolution, Figure 3.6(a) illustrates the number of metamodel elements for each considered version of the FLUID metamodel. The area is further partitioned into the different kinds of metamodel elements. The figure clearly shows the transition from an initial development phase to a maintenance phase around version 6, where the grow in number of metamodel elements slows down.




(a) FLUID (b) TAF-Gen

**Figure 3.6:** Metamodel evolution in numbers

[2]see BMW Car IT web site: `http://www.bmw-carit.de/`

**TAF-Gen** (Test Automation Framework - Generator) is a framework to automatically generate test cases for human machine interfaces in the automotive domain [Benz, 2007]. The metamodel defines a statechart variant, a structural screen model and a test case language.

Figure 3.6(b) depicts the number of metamodel elements for each considered version of the TAF-Gen metamodel. In this case, the transition from the initial development phase to the maintenance phase is more distinctive and happens around version 13.

### 3.3.3 Study Execution

The histories of the metamodels were only available in the form of snapshots. A snapshot depicts the state of a metamodel at a particular point in time. As a consequence, further information had to be inferred to obtain the coupled changes leading from one metamodel version to the next. In order to achieve this, we performed the following steps[3]:

1. *Extraction of metamodel versions* (1 person week): Each available version of the metamodel was obtained from the revision control system used in the development of the metamodel[4].
2. *Comparison of subsequent metamodel versions* (2.5 person weeks): Since both revision control systems used are snapshot-based, they provide no information about the sequence of changes which led from one version to the following. Therefore, successive metamodel versions had to be compared in order to obtain the changes in a difference model. The difference model consists of a collection of primitive changes from one metamodel version to the next and has been determined with the help of tool support[5].
3. *Detection of coupled changes* (3 person weeks): Some primitive changes only make sense when regarded in combination with others. When an attribute is for example removed from a class and an attribute with the same name and type is added to its super class, then the two changes have to be interpreted as a compound change in order to conserve the values of the attribute in a model. Therefore, primitive changes were combined based on the information how corresponding model elements were migrated. The coupled changes between metamodel versions were documented in a table.
4. *Classification of coupled changes* (1 person week): The classification was applied to each detected coupled change.

[3] In order to get an impression of the extent of the study, the approximate effort is mentioned in parenthesis for each step.

[4] The metamodels were specified by the Ecore metamodeling language from the Eclipse Modeling Framework (EMF, `http://www.eclipse.org/modeling/emf/`).

[5] Two prototypes now contributed to the EMF Compare tool (`http://www.eclipse.org/modeling/emft/?project=compare`) were applied.

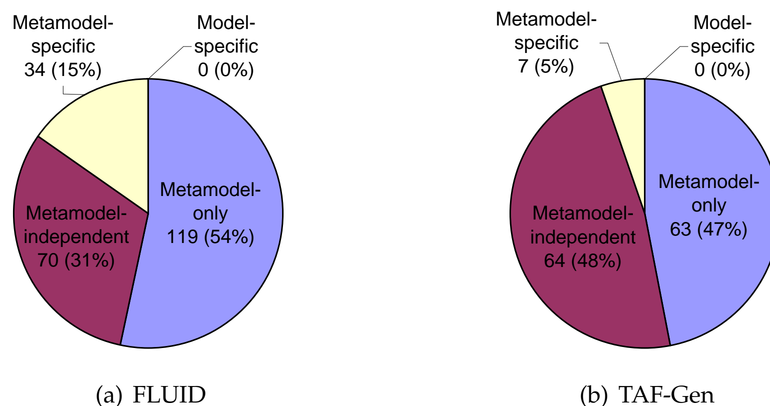
## 3.4 Study Implementation

This section presents the results of the study, discusses them and mentions threats to the results' validity.

### 3.4.1 Study Result

In this section, we present the results of the study in a compiled form. The full results are shown in Table 3.1, together with an informal description of the metamodel adaptation and model migration. Due to a non-disclosure agreement with BMW Car IT, we cannot provide more detailed information.

**FLUID.** Figure 3.7(a) shows which fraction of all metamodel changes falls into each class. 54% of the metamodel changes can be classified as metamodel-only. No coupled change can be classified as model-specific, 15% as metamodel-specific and 31% as metamodel-independent. Note that each metamodel change was counted as one, even though some changes were more complex than others.




(a) FLUID (b) TAF-Gen

**Figure 3.7:** Classification of metamodel changes

**TAF-Gen.** Figure 3.7(b) illustrates the fragmentation of all encountered metamodel changes into the four classes. 47% of the metamodel changes have not required a migration at all. As in the history of FLUID, no coupled change is model-specific. The fraction of metamodel-independent coupled changes is even higher than in case of FLUID, amounting to 48% compared to 5% classified as metamodel-specific.

### 3.4.2 Study Discussion

**RQ1.** *Which fraction of metamodel changes is metamodel-only and thus requires no model migration?* The study showed that in practice the history of a metamodel can be split into mostly small metamodel changes. We found out that most metamodel changes required a migration of existing models. Furthermore, snapshots from different metamodel versions are not sufficient to derive the model migration.

**Table 3.1:** Number of occurred metamodel changes and their classification

| Element | Metamodel adaptation | Model migration | Class | FLUID | TAF-Gen | Overall |
|---|---|---|---|---|---|---|
| Pack. | create package | no migration required | MMO | 0 | 10 | 10 |
| | move package | compensate move | MMI | 0 | 8 | 8 |
| | delete empty package | no migration required | MMO | 0 | 4 | 4 |
| Enum. | create enumeration | no migration required | MMO | 0 | 4 | 4 |
| | reorder enumeration literal | no migration required | MMO | 0 | 1 | 1 |
| | rename enumeration literal | compensate rename | MMI | 0 | 4 | 4 |
| Class | create class | no migration required | MMO | 0 | 3 | 3 |
| | create subclass | no migration required | MMO | 45 | 2 | 47 |
| | change superclass | remove data of lost features | MMI | 0 | 1 | 1 |
| | move class | compensate move | MMI | 5 | 18 | 23 |
| | delete class | remove objects | MMI | 1 | 0 | 1 |
| | delete subclass | remove objects | MMI | 1 | 0 | 1 |
| | rename class | compensate rename | MMI | 5 | 5 | 10 |
| | make class abstract | migrate objects to sub classes | MMS | 1 | 0 | 1 |
| | specialize superclass (optional features) | no migration required | MMO | 3 | 1 | 4 |
| Attribute | create optional attribute | no migration required | MMO | 15 | 5 | 20 |
| | create required attribute | set default value | MMI | 5 | 1 | 6 |
| | change attribute type | convert values | MMI | 2 | 1 | 3 |
| | decrease upper bound of reference | remove superfluous values | MMS | 1 | 0 | 1 |
| | drop attribute identifier | no migration required | MMO | 6 | 0 | 6 |
| | generalize attribute multiplicity | no migration required | MMO | 5 | 1 | 6 |
| | make attribute identifier | guarantee uniqueness | MMS | 3 | 0 | 3 |
| | pull up optional attribute | no migration required | MMO | 1 | 0 | 1 |
| | pull up required attribute | set default value | MMI | 1 | 0 | 1 |
| | delete attribute | delete attribute values | MMI | 11 | 1 | 12 |
| | rename attribute | compensate rename | MMI | 3 | 1 | 4 |
| Association | create optional reference | no migration required | MMO | 5 | 4 | 9 |
| | create optional composite reference | no migration required | MMO | 3 | 3 | 6 |
| | create required composite reference | create objects | MMI | 3 | 0 | 3 |
| | change reference type | delete links | MMI | 5 | 0 | 5 |
| | change composite reference type | migrate objects | MMS | 1 | 0 | 1 |
| | decrease upper bound of reference | delete superfluous links | MMS | 1 | 1 | 2 |
| | make reference transient | delete links | MMI | 0 | 1 | 1 |
| | drop reference transient | no migration required | MMO | 0 | 1 | 1 |
| | generalize reference multiplicity | no migration required | MMO | 14 | 2 | 16 |
| | generalize reference type | no migration required | MMO | 6 | 4 | 10 |
| | increase lower bound of reference | no migration required | MMO | 2 | 0 | 2 |
| | create opposite reference | no migration required | MMO | 0 | 3 | 3 |
| | move reference along reference | move links accordingly | MMI | 0 | 1 | 1 |
| | push down reference | no migration required | MMO | 0 | 2 | 2 |
| | pull up optional reference | no migration required | MMO | 1 | 1 | 2 |
| | delete composite reference | delete objects | MMI | 3 | 0 | 3 |
| | delete cross reference | delete links | MMI | 1 | 4 | 5 |
| | delete opposite reference | no migration required | MMO | 0 | 2 | 2 |
| | rename reference | compensate rename | MMI | 7 | 3 | 10 |
| | make reference composite | no migration required | MMO | 1 | 2 | 3 |
| | drop reference composite | assign otherwise | MMS | 22 | 3 | 25 |
| Composite | reference to class | replace link by reference object | MMI | 0 | 1 | 1 |
| | extract class | extract object for extracted class | MMI | 6 | 2 | 8 |
| | extract superclass | no migration required | MMO | 3 | 1 | 4 |
| | inline class | remove object of inlined class | MMI | 0 | 1 | 1 |
| | merge references | move data accordingly | MMI | 4 | 0 | 4 |
| | inline subclass | migrate objects to super class | MMI | 0 | 2 | 2 |
| | inline superclass | no migration required | MMI | 0 | 1 | 1 |
| | merge classes | migrate objects to common class | MMI | 1 | 0 | 1 |
| | class to reference | replace reference object by link | MMI | 1 | 0 | 1 |
| | reference to identifier | set identifier based on links and remove links | MMI | 2 | 0 | 2 |
| | identifier to reference | create links based on identifier | MMI | 2 | 0 | 2 |
| | inheritance to delegation | create delegation object | MMI | 0 | 1 | 1 |
| | fold class | extract object for folded class | MMI | 1 | 1 | 2 |
| | fold superclass | no migration required | MMO | 1 | 0 | 1 |
| Compl. | create optional composite structure | no migration required | MMO | 8 | 7 | 15 |
| | delete composite structure | remove data | MMI | 0 | 6 | 6 |
| | complex restructuring | complex migration | MMS | 5 | 3 | 8 |

| | | FLUID | TAF-Gen | Overall |
|---|---|---|---|---|
| | **Overall** | **223** | **134** | **357** |
| Metamodel-only | **MMO** | **119** | **63** | **182** |
| Metamodel-independent | **MMI** | **70** | **64** | **134** |
| Metamodel-specific | **MMS** | **34** | **7** | **41** |
| Model-specific | **MS** | **0** | **0** | **0** |

**RQ2.** *Which fraction of metamodel changes is model-specific and thus defies automation of migration?* As we have not found any model-specific coupled changes, we would have been able to specify transformations to automate migration of all models. However, model-specific coupled changes cannot be entirely excluded and we further investigate them in Section 5.4 (*Limitations of Automating Model Migration*).

**RQ3.** *Which fraction of metamodel changes is metamodel-independent and thus generalizable across metamodels?* More than two thirds of all coupled changes were classified as metamodel-independent which provides a large potential for further automation. We also found a small number of metamodel-specific coupled changes and thus the model migration was a combination of both metamodel-specific and -independent coupled changes.

### 3.4.3 Threats to Validity

The result of the study suggests a high degree of potential automation for model migration. However, threats need to be mentioned which can affect the validity of the result. They are presented according to the steps of the method to which they apply together with the measures taken to mitigate them:

1. *Extraction of metamodel versions*: It was assumed that committing the metamodel to the revision control system indicates a new version of the metamodel. Therefore, only the primitive changes from one commit to the next were considered to be combined. However, metamodels were sometimes committed in a premature version, and hence complex changes which span several commits of the metamodel threaten the validity. Even though enacted development guidelines at BMW Car IT forbid to commit artifacts in a premature version, primitive changes of other metamodel versions were also taken into account, when a migration could not be determined otherwise.
2. *Comparison of subsequent metamodel versions*: A prerequisite to determine the differences is the calculation of a matching between the elements of one metamodel version and those of the next. However, in the absence of unique and unchangeable element identifiers, the comparison cannot always be performed unambiguously [Robbes and Lanza, 2007]. Furthermore, the difference model leaves out changes which have been overwritten by others in the course of the evolution from one version to the next. In order to mitigate this threat, the correctness of the primitive changes was validated in close cooperation with the language engineers.
3. *Detection of coupled changes*: Unfortunately, models were not available for all versions of the corresponding metamodels. This poses a threat to the correct formation of coupled changes, since primitive changes were combined based on the associated migration. In order to mitigate this risk, the language engineers were exhaustively questioned about the correctness of the derived migration.
4. *Classification of coupled changes*: The differentiation between metamodel-specific or -independent coupled changes is not 100% sharp. Even though a generalization may be constructed for the most sophisticated changes, it is unlikely that it

can be reused on any other metamodel. In order to mitigate this risk, we chose a conservative strategy: When we were not sure whether reuse makes sense, we classified such a coupled change rather metamodel-specific than metamodel-independent.

## 3.5 Requirements for Automating Model Migration

Based on the results of the analysis, we discuss several requirements that an approach must fulfill in order to profit from the automation potential in practice.

*Reuse of migration knowledge.* To profit from the high number of metamodel-independent coupled changes found in the study, a practical approach needs to provide a mechanism to specify metamodel adaptation and corresponding model migration independent of a specific metamodel.

*Expressive, custom migrations.* As there was a non-negligible number of metamodel-specific coupled changes, the approach must be flexible enough to allow for the definition of custom metamodel adaptation and model migration. Since metamodel-specific changes can be arbitrarily complex, the formalism must be expressive enough to cover all evolution scenarios.

*Modularity.* In order to be able to specify the different kinds of coupled changes independently of each other, a practical approach must be modular. Modularity implies that the specification of a coupled change is not affected by the presence of any other coupled change.

*History.* Since snapshots of the metamodel versions are not sufficient to derive the model migration, a metamodel history is required in which the performed metamodel changes are recorded.

Existing approaches to automate model migration only satisfy the stated requirements to a certain degree: The manual specification approaches of Sprinkle [Sprinkle and Karsai, 2004], MCL [Narayanan et al., 2009] and Flock [Rose et al., 2010d] do not provide a construct for the reuse of migration knowledge across metamodels. The operation-based approach by Wachsmuth [Wachsmuth, 2007] is not expressive enough to capture all kinds of migration scenarios—due to the restricted set of high-level primitives. The matching approaches by Gruschko [Gruschko et al., 2007], by Cicchetti [Cicchetti et al., 2008] and AML [Garcés et al., 2009] leave open how they achieve modularity and how they deal with complex custom migrations. In order to fully profit from the automatablity of model migration in practice, an approach is needed that fulfills all the presented requirements.

## 3.6 Summary

We presented a study of the evolution of two real world metamodels. Our study confirmed that metamodels evolve in practice and that most metamodel changes require a migration of existing models. The study's main goal was to determine the poten-

tial for reduction of language evolution efforts through appropriate tool support. To this end, we categorized metamodel changes according to their degree of metamodel specificity. When a change is metamodel-specific, the corresponding model migration is as well. Otherwise, the model migration can be reused to migrate models that obey to different metamodels.

Our results show that there is a large potential for the reuse of coupled evolution operations, because more than two thirds of all coupled changes were not metamodel-specific. If metamodel adaptation and model migration are encapsulated into a coupled operation, it is possible to reuse the operation for the evolution of different metamodels and their models. Such reuse of already tested coupled evolution operations can reduce maintenance effort and error likelihood.

Nevertheless, a third of the coupled changes were specific to the metamodel's domain and therefore required a custom model migration. A metamodel hence evolves in a sequence of metamodel-specific and -independent changes. Therefore, an approach for automated model migration must support the reuse of coupled changes as well as the definition of new metamodel-specific changes.

# Chapter 4

# State of the Art: A Cross-Space Survey on Coupled Evolution

Like any software artifact, software languages are subject to evolution [Favre, 2005]. When a software language evolves, existing language utterances may no longer conform to the evolved language. To prevent loss of information, existing utterances need to be migrated. Coupled evolution automates the migration of existing utterances by attaching a migration to the evolution of a language definition. Software language evolution affects different technical spaces like dataware, grammarware, XMLware, and modelware [Kurtev et al., 2002]. In each technical space, different coupled evolution approaches have been proposed. However, it is largely unknown how approaches from different technical spaces relate to each other. To alleviate this, we performed a systematic literature review on coupled evolution approaches. We derived a feature model focused on determining commonalities and differences between approaches from different technical spaces. We motivate our approach from issues of the existing approaches in modelware and from the lessons learned from approaches in the other technical spaces.

**Contents**


**4.1 Cross-Space Terminology** . . . 82
**4.2 Review Systematics** . . . 84
**4.3 Classification of Approaches** . . . 86
**4.4 Dataware** . . . 89
**4.5 Grammarware** . . . 95
**4.6 XMLware** . . . 96
**4.7 Modelware** . . . 97
**4.8 Cross-Space Comparison** . . . 99
**4.9 Motivation of our Approach** . . . 101
**4.10 Summary** . . . 103


We start with the terminology that we define for our review in Section 4.1 (*Cross-Space Terminology*), and the systematics of our review in Section 4.2 (*Review System-*

*atics*). Next, we propose the classification scheme in Section 4.3 (*Classification of Approaches*). In Section 4.4 (*Dataware*) to Section 4.7 (*Modelware*), we present the various approaches for coupled evolution together with their classification. Each section deals with a particular technical space. Finally, we summarize the approaches across all technical spaces in Section 4.8 (*Cross-Space Comparison*) and motivate our approach in Section 4.9 (*Motivation of our Approach*), before we conclude in Section 4.10 (*Summary*).

## 4.1 Cross-Space Terminology

Figure 4.1 illustrates the terminology on which the survey is based. The terms are independent of the technical space to be able to compare the different technical spaces with each other. To be independent of the technical space, the terms are adopted from linguistics [van Sterkenburg, 2003].

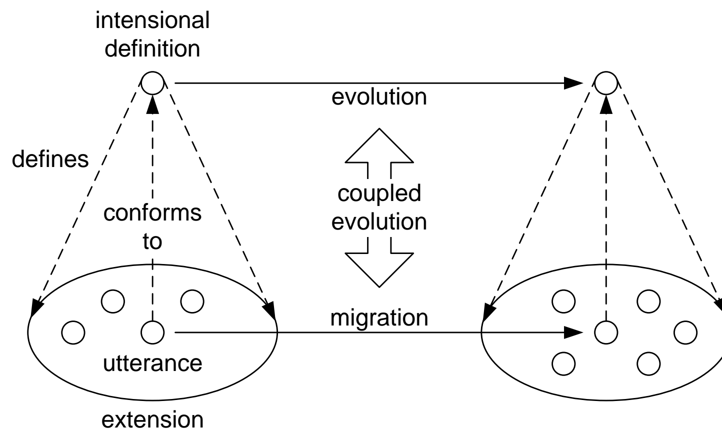



**Figure 4.1:** Terminology across technical spaces

**Language Definition.** In the survey, we do not only consider modeling languages, but all sorts of languages used to develop software:

**Definition 4.1** (Software Language)**.** *Software language is a general term for artificial languages that are used to develop software.*

Software languages exist in different technical spaces [Kurtev et al., 2002], e.g. programming languages, modeling languages, XML formats, and data models. All these technical spaces [Kurtev et al., 2002] deal with *intensional definition*s of possibly infinite sets of *utterances*:

**Definition 4.2** (Utterance)**.** *An utterance of a software language is an element that can be built in the syntactic domain of the software language.*

**Definition 4.3** (Intensional Definition)**.** *An intensional definition of a software language defines whether an utterance is syntactically correct with respect to the software language.*

In that sense, the term utterance generalizes the term model, and the term intensional definition generalizes the term metamodel, which are both defined in Section 2.2.2

(*Abstract Syntax of a Modeling Language*). Like a metamodel, an intensional definition determines the set of syntactically correct utterances which is called *extension*:

**Definition 4.4** (Extension)**.** *An extension of a software language is the set of utterances that are syntactically correct with respect to the software language.*

Depending on the technical space, different terms have been established for such an intensional definition, its extension, and the utterances in the extension. Grammarware and modelware specify *languages* by *grammars* respectively *metamodels*. In grammarware, the utterances of a language are either called *word* or *sentence*. In modelware, these utterances are called *model*. XMLware and dataware rely on *schemas* to define sets of *documents* respectively *databases*. While dataware provides no term for a set of databases, a set of documents is either called *format* or *language* in XMLware. Table 4.1 summarizes the various terms used in the different technical spaces.

**Table 4.1:** Terminology in different technical spaces

| technical space | dataware | grammarware | XMLware | modelware |
|---|---|---|---|---|
| intensional definition | schema | grammar | schema | metamodel |
| extension | — | language | format/language | language |
| utterance | database | word/sentence | document | model |

**Coupled Evolution.** Despite their different terminology, all these technical spaces face a common problem: intensional definitions are subject to *evolution* [Favre, 2005]:

**Definition 4.5** (Evolution)**.** *An evolution is a transformation that transforms the intensional definition of a software language from an old to a new version.*

As a consequence, utterances which conform to an original definition might no longer conform to an evolved definition and *migration* is needed:

**Definition 4.6** (Migration)**.** *A migration is a transformation that transforms an utterance that conforms to the old version of an intensional definition to an utterance that conforms to its new version.*

This migration is often called co-evolution, since it depends on the evolution of the definition. Manual migration of utterances is tedious and error-prone, hence migration needs to be automated. *Coupled evolution* addresses the automation of the migration based on its dependency on the evolution of intensional definitions [Lämmel, 2004]:

**Definition 4.7** (Coupled Evolution)**.** *Coupled evolution is a combination of the evolution of an intensional definition and the corresponding migration of the utterances of the intensional definition.*

**Terminological Conventions.** Throughout the survey, we use the terms according to the technical space addressed by a certain approach. When talking about approaches from different technical spaces, we stick to the general terms *intensional definition*, *extension*, and *utterance*. For an utterance of the extension of a definition, we say the utterance *conforms to* the definition.

## 4.2 Review Systematics

A systematic review requires a thorough publication search strategy to cover all research conducted within the scope of the survey. Clearly defined selection criteria are needed to refine the set of found publications on relevance. In this section, we discuss both the search strategy and selection criteria.

### 4.2.1 Search Strategy

The rigor of the search process is a distinguishing factor for systematic reviews versus ad-hoc reviews [Kitchenham and Charters, 2007]. Following an iterative process, we have set a search strategy and followed it throughout the survey. The search strategy comprises two stages: A selection of relevant papers from a large set of conferences and journals (the initial sources), and recursively and exhaustively following relevant references of all papers included in the survey.

**Initial Sources.** As a starting point of the survey, we chose a set of relevant conferences and journals shown in Table 4.2. By studying all editions of each of these conferences and journals, we selected potentially relevant publications using a liberal application of the selection criteria outlined below. We applied a more refined selection process later, when reviewing selected publications. The set of conferences and journals is not intended to be a complete set containing all relevant literature. It merely provides a set of initial sources.

**Reference Inclusion.** To complement the initial sources, for each publication, we included all referenced work relevant to the survey. By applying reference inclusion recursively, we expanded the survey outside the scope of the initial sources. By applying the recursive reference inclusion exhaustively, we completed the set of publications.

### 4.2.2 Selection Criteria

The survey covers published literature, with the exclusion of workshop papers. We have set out the scope of the survey by means of a set of inclusion and exclusion criteria, presented below. Papers falling within the relevant technical space, yet rejected based on the selection criteria, were recorded along with the reason for rejection. The list of excluded papers together with the reason for rejection can be found in Chapter A (*Papers Excluded from the Survey*) in the appendix.

**Inclusion Criteria.** This survey focuses on *coupled evolution* of an intensional definition and its utterances. We speak of *evolution*, when external factors cause the intensional definition to vary over time, yielding different versions of the same definition. Subsequent versions should show clear resemblance. *External factors* are influences not enforced by the surrounding system itself, examples are a changing domain, an increased knowledge or understanding of the system, or a changing user base. We speak of *coupled* evolution, when the evolution of the intensional definition primarily

**Table 4.2:** Initial article sources

| Acronym | Full Name | Years |
|---|---|---|
| **Conferences** | | |
| BNCOD | British National Conference on Databases | 1981 – 2009 |
| CAiSE | Int. Conference on Advanced Information Systems Engineering | 1989 – 2009 |
| CIKM | Int. Conference on Information and Knowledge Management | 1992 – 2009 |
| CSMR | European Conference on Software Maintenance and Reengineering | 1997 – 2009 |
| ECMFA | European Conference on Modeling Foundations and Applications | 2005 – 2009 |
| ECOOP | European Conference on Object-Oriented Programming | 1987 – 2009 |
| EDOC | Int. "Enterprise Computing Conference" | 2000 – 2009 |
| ER | Int. Conference on Conceptual Modeling | 1979 – 2009 |
| GTTSE | Generative and Transformational Techniques in Software Engineering | 2005 – 2007 |
| ICDE | Int. Conference on Data Engineering | 1988 – 2010 |
| ICMT | Int. Conference on Model Transformation | 2008 – 2009 |
| ICSE | Int. Conference on Software Engineering | 1976 – 2009 |
| ICSM | Int. Conference on Software Maintenance | 1993 – 2009 |
| MODELS | Int. Conference on Model Driven Engineering Languages and Systems | 1997 – 2009 |
| OOPSLA | Object-Oriented Programming, Systems, Languages & Applications | 1986 – 2009 |
| SLE | Int. Conference on Software Language Engineering | 2008 – 2009 |
| VLDB | Int. Conference on Very Large Databases | 1975 – 2009 |
| WCRE | Working Conference on Reverse Engineering | 1993 – 2009 |
| **Journals** | | |
| JSME | Journal of Software Maintenance and Evolution | 1989 – 2010 |
| JVLC | Journal of Visual Languages and Computing | 1993 – 2010 |
| KAIS | Knowledge and Information Systems | 1999 – 2010 |
| SIGMOD | ACM's Special Interest Group on Management of Data | 1977 – 2009 |
| SIGPLAN | ACM's Special Interest Group on Programming Languages | 1987 – 2010 |
| SoSyM | Software and Systems Modeling | 2002 – 2010 |
| TKDE | IEEE Transactions on Knowledge and Data Engineering | 1989 – 2010 |
| TOPLAS | ACM Transactions on Programming Languages and Systems | 1979 – 2010 |
| TOSEM | ACM Transactions on Software Engineering and Methodology | 1992 – 2010 |
| TSE | IEEE Transactions on Software Engineering | 1975 – 2010 |
| VLDBJ | Journal on Very Large Databases | 1992 – 2010 |

determines utterance migration. Manual migration of individual utterances falls outside the scope of the survey due to the lack of coupling to the evolution. Supported manual construction of an automatic migration falls within the scope of the survey.

**Exclusion Criteria.** We excluded work focused on comparison of intensional definitions, since these do not discuss a coupling. Such comparison includes work on change detection, model comparison, difference calculation and difference representation. We also excluded work on schema matching, schema integration, database integration and migration of legacy database systems, since in these works, subsequent versions of the intensional definition (if even existent) do not have to show clear resemblance. As such, there is no clear focus on evolution. Finally, we also excluded work on database views on utterances, when these are not explicitly called in to prevent or aid coupled evolution. The technical space of ontology evolution [Flouris et al., 2008] is considered out of scope of the survey, since it generally does not consider utterance migration. API evolution is considered out of scope, since the extension is not completely defined by the intensional definition.

## 4.3 Classification of Approaches

We were able to derive a classification scheme which is independent of technical spaces, that is we can classify approaches from the different technical spaces along the same criteria. Using the classification, we can thus compare approaches from different technical spaces. Figure 4.2 presents the topmost level of the classification scheme as a feature model [Kang et al., 1990]. An approach can be classified according to the technical space it addresses, the way it handles evolution and migration, and the evaluation it has undergone.

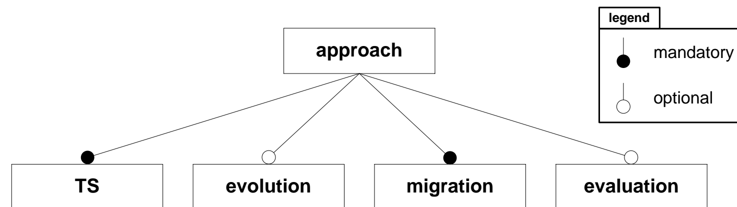



**Figure 4.2:** Classification of coupled evolution approaches

### 4.3.1 Technical Space

Figure 4.3 presents the classification of the approaches according to the technical space (TS) they address. We cover the technical spaces of *dataware*, *grammarware*, *XMLware*, and *modelware*. In dataware, we distinguish approaches which address *relational* and *object-oriented* database management systems.

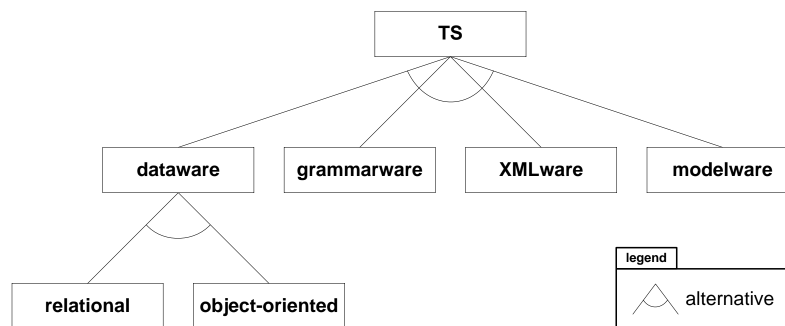



**Figure 4.3:** Classification of approaches according to technical spaces

### 4.3.2 Evolution

Figure 4.4 presents the classification of the approaches according to how they specify and obtain the evolution of the intensional definition.

**Specification.** The evolution of an intensional definition is implicitly specified by the original and the evolved version of the definition. However, many approaches are based on explicit evolution specifications. We distinguish two styles of such specifications: *Imperative* specifications describe the evolution by a sequence of applications

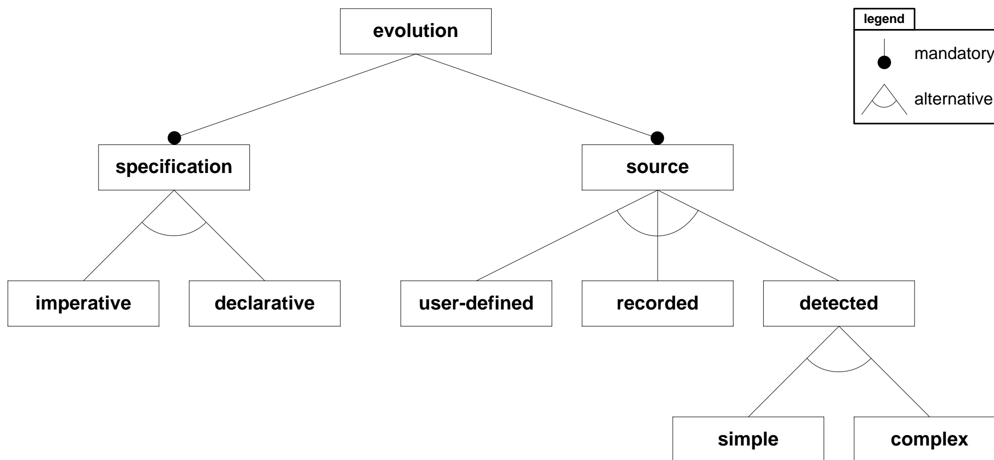



**Figure 4.4:** Classification of approaches according to evolution

of change operations. In contrast, *declarative* specifications model the evolution by a set of differences between the original and evolved version of a definition.

**Source.** Explicit evolution specifications can have different sources. One prominent source is the automated *detection* of the evolution based on the original and evolved version of a definition. We distinguish two kinds of detections: First, detections which are only able to detect *simple* changes like additions and deletions. For some approaches, this includes the detection of moves as well. Second, detections which can also detect more *complex* changes, for example extracting and inlining of constructs. As an alternative to detection, the evolution can be *recorded* while the user edits a definition, or *user-defined* where the user specifies the evolution manually.

### 4.3.3 Migration

Figure 4.5 presents the classification of the approaches according to how they specify and perform the migration.

**Coupling.** In contrast to evolution, migration is always specified explicitly. In coupled evolution, the dependency of migration on evolution is reflected by coupling evolution specifications with migration specifications. We distinguish three kinds of couplings: With a *fixed* coupling, the migration is completely defined by the evolution. Only the developer of a coupled evolution tool can add new couplings. With an *overwritable* coupling, the user can overwrite single applications of a coupling with custom migrations. With an *extendable* coupling, the user can add completely new couplings between elements of evolution and migration specifications.

**Language.** Migration specifications must be executable and therefore be expressed in executable languages. Such a language might be *customly defined* as a domain-specific migration language. Alternatively, an existing *transformation language* (TL) can be reused. Typically, this language comes from the technical space addressed by a coupled evolution approach. Another way is to add migration support to a *general-*

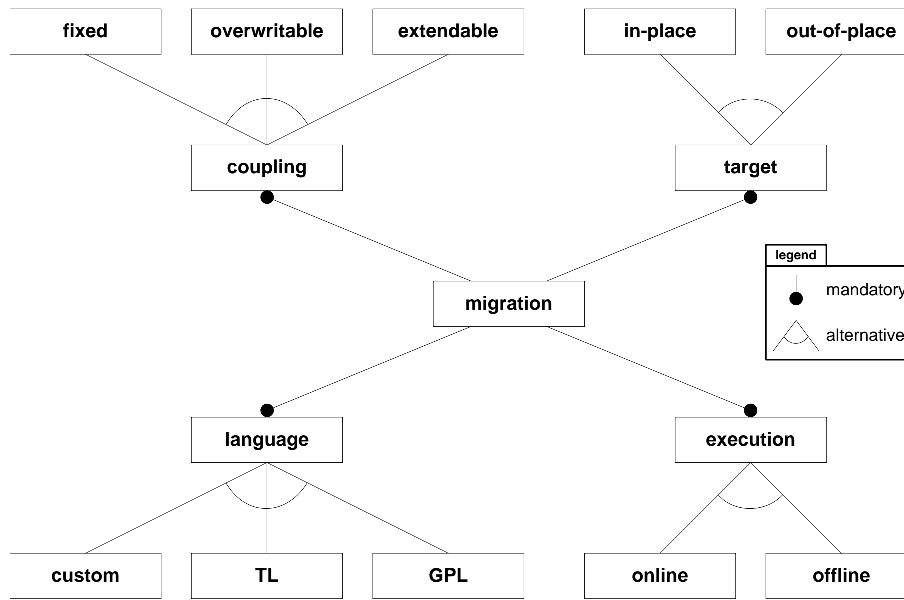



**Figure 4.5:** Classification of approaches according to migration

*purpose programming language* (GPL) in form of an API or an embedded domain-specific language.

**Target.** Migration might be performed either *in-place* or *out-of-place*. In the first case, the target of the migration is the original utterance itself which is modified during migration. In the second case, the target is a new migrated utterance which is created during migration. The original utterance is preserved.

**Execution.** Furthermore, the migration might be executed *offline* where applications can not use any of the utterances during the migration, or *online* where applications can still use the utterances and where the usage of an utterance by an application triggers lazy migration.

### 4.3.4 Evaluation

Figure 4.6 presents the classification of the approaches according to the degree of their evaluation. Evaluation is crucial for the validation of coupled evolution approaches. Approaches might provide no evaluation at all. They might provide only evaluation of *preliminary* nature, e.g. by toy examples. Often, evaluation is required explicitly in corresponding papers. Other approaches perform a *case study* on industrial or open-source systems of medium to large scale. Some authors provide a *comparison* of their approach with existing approaches.

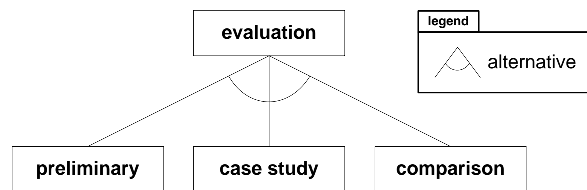



**Figure 4.6:** Classification of approaches according to evaluation

## 4.4 Dataware

In dataware, a database is the utterance that conforms to the schema which is the intensional definition. There is no term for the extension that is defined by the intensional definition. Schema evolution has been a field of study for several decades, yielding a substantial body of research [Rahm and Bernstein, 2006]. Due to the large number of approaches, dataware is further subdivided—according to the data modeling paradigm—into relational [Codd, 1970] and object-oriented [Kim, 1990] dataware.

On the technical space of database schema evolution, *Roddick* presents an annotated bibliography [Roddick, 1992]. The bibliography categorizes papers along the evolving formalism into evolution of relational data models and object-oriented data models as well as miscellaneous works. Relevant papers from the bibliography are included in this survey.

### 4.4.1 Relational Dataware

In relational dataware, schemas define tables and relations between tables. Tables consist of records, which are products of primitive values. Relations are modeled explicitly within records, yet their consistency is generally ensured by the database system. Table 4.3 lists the approaches from relational dataware together with their classification according to the feature model presented in Section 4.3 (*Classification of Approaches*). We distinguish manual specification and operation-based approaches. The typical features of these approaches are emphasized in the table.

**Table 4.3:** Classification of the relational dataware approaches

<table>
<tr><th colspan="2"></th><th colspan="2">Evolution</th><th colspan="5">Migration</th><th>Evaluation</th></tr>
<tr><th colspan="2">Approach</th><th>Specific.</th><th>Source</th><th>Coupling</th><th colspan="2">Language</th><th>Target</th><th>Exec.</th><th></th></tr>
<tr><td colspan="10">Manual specification</td></tr>
<tr><td colspan="2">Ronström</td><td>imperative</td><td>user-defined</td><td>overwritable</td><td colspan="2">GPL</td><td>out</td><td>online</td><td>—</td></tr>
<tr><td colspan="10">Operation-based</td></tr>
<tr><td colspan="2">Shneiderman</td><td rowspan="3">imperative</td><td rowspan="3">user-defined</td><td rowspan="2">fixed</td><td colspan="2">TL</td><td>out</td><td rowspan="2">offline</td><td rowspan="2">—</td></tr>
<tr><td>Ambler</td><td>SQL</td><td colspan="2">GPL</td><td rowspan="2">in</td></tr>
<tr><td>PRISM</td><td>SQL</td><td>overwritable</td><td>TL</td><td>SQL</td><td>online</td><td>case study</td></tr>
</table>

**Manual Specification** approaches require the user to manually specify the database migration. *Ronström* presents an approach for online schema evolution and migration of a telecom database [Ronström, 2000]. Schema evolution is performed by first

creating the new schema elements, copying old data and keeping the data in sync by appropriate triggers. Next, the new schema elements are tested and if successful, new transactions may be executed, and old data and schema elements are removed.

**Operation-based** approaches specify coupled evolution as a sequence of coupled operations, encapsulating both schema evolution and database migration.

*Shneiderman* et al. propose an architecture for the coupled evolution of relational schemas and databases as well as applications and programs [Shneiderman and Thomas, 1982]. They present 15 coupled operations for schema transformation and discuss their effect on databases in terms of a relational algebra. Though based on previous practical experiences with their own schema definition language, the approach is completely theoretical.

*Ambler* and Sadalage propose an agile and evolutionary design of a relational database [Ambler and Sadalage, 2006]. Their book discusses database refactoring, evolutionary data modeling, database regression testing, configuration management for database artifacts and sandboxes for developers.

*PRISM* is a schema evolution workbench providing schema modification operations, tools to evaluate schema change effects, translation of old queries, automatic data migration, and documentation of intervened changes [Curino et al., 2008b, Curino et al., 2009]. Migration predictability is achieved by characterizing the extent of information preservation in response to schema changes, and by automating data conversion. PRISM has been evaluated by reverse engineering the schema evolution of Wikipedia [Curino et al., 2008a].

**Other Papers.** Sockut and Goldberg introduce basic concepts of database reorganization [Sockut and Goldberg, 1979]. They classify database reorganizations into levels along the affected construct. The end-user level represents data views, the infological level defines attributes and relationships, the string level defines access paths, the encoding level defines physical representation, and the physical device level maps representation onto storage.

Ventrone et al. argue that, similar to database integration, domain evolution can create problems of semantic heterogeneity—i.e. clashes of implicit semantics [Ventrone and Heiler, 1991]. These are similar to those encountered in database integration and similar solutions apply.

Roddick discusses schema versioning issues [Roddick, 1995]. He concludes for any versioning solution: A database administrator should guide schema modifications; Schema modifications should be symmetric—i.e. existing data is viewable through the new schema and later recorded data is viewable through the previous schema; And schema modifications should be expressed in algebraic operations for formal verification.

Sjøberg measures evolution and its impact in a health management system, comprising 150k lines of code [Sjøberg, 1993]. Various types of names (such as class and relation names) and their usage are tracked, detecting additions and deletions over time. Renaming and more complex changes are left undetected. Over 18 months, while transitioning from development to production, relations increased by 139%

and fields by 274%. In one month, comprising 140 schema changes, one third of the names were deleted and one tenth were added, affecting nearly 6,000 code locations.

### 4.4.2 Object-Oriented Dataware

In object-oriented dataware, schemas are defined using classes which can inherit from other classes, define attributes or associations to other classes. A database consists of objects which are instances of these classes. Casais [Casais, 1995], Benatallah [Benatallah, 1999], as well as Rashid and Sawyer [Rashid and Sawyer, 2005] present categories of approaches for object-oriented dataware which we have combined. Operation-based and schema matching approaches modify the schema and database in-place and specify the schema evolution either imperatively or declaratively. Versioning and view-based approaches allow their users to have different schema versions present at the same time and transform the database between these versions out-of-place. Hybrid approaches combine complementary approaches. Table 4.4 enumerates the approaches from object-oriented dataware together with their classification.

**Table 4.4:** Classification of the object-oriented dataware approaches

| Approach | | Evolution: Specific. | Evolution: Source | Migration: Coupling | Migration: Language | | Migration: Target | Migration: Exec. | Evaluation |
|---|---|---|---|---|---|---|---|---|---|
| **Operation-based** | | | | | | | | | |
| Banerjee | ORION | *imperative* | user-defined | fixed | — | | *in* | online | — |
| Penney | GemStone | | | | | | | offline | |
| Nguyen | Sherpa | | | | | | | online | |
| Al-Jadir | F2 | | | | | | | offline | case study |
| Ferrandina | $O_2$ | | | overwritable | GPL | C++ | | online | — |
| SERF | PSE | | | extendable | | OQL | | offline | |
| **Schema matching** | | | | | | | | | |
| OTGen | | *declarative* | recorded | overwritable | TL | | *in* | offline | — |
| TESS | | | det. complex | | | | | | case study |
| **Class versioning** | | | | | | | | | |
| Skarra | ENCORE | declarative | user-defined | fixed | GPL | | *out* | *online* | — |
| Monk | CLOSQL | | | overwritable | TL | | | | |
| SADES | Jasmine | | | extendable | | | | | comparison |
| **Schema versioning** | | | | | | | | | |
| Kim | ORION | imperative | user-defined | fixed | — | | *out* | *online* | — |
| Andany | Farandole 2 | imperative | | | | | | | |
| Clamen | | declarative | | overwritable | TL | | | | |
| Lautemann | COAST | imperative | | | | | | | |
| Bouneffa | GORM | imperative | | | | | | | |
| **View-based** | | | | | | | | | |
| Tresch | COCOON | *declarative* | user-defined | overwritable | TL | COOL | *out* | *online* | — |
| EVER | | | | | | EVER | | | |
| Brèche | $O_2$ | | | | | VDL | | | |
| TSE | GemStone | | recorded | | | MV | | | preliminary |
| **Hybrid** | | | | | | | | | |
| Benatallah | | imperative | user-defined | overwritable | TL | OQL | *i/o* | *online* | — |

**Operation-based** approaches specify schema evolution imperatively as a sequence

of schema modification operations. An operation application not only adapts the schema, but also triggers in-place migration of the database to restore a consistent state.

*Banerjee* et al. present the semantics of a fixed set of primitive operations for ORION [Banerjee et al., 1987]. The operations are sound—i.e. preserve the schema invariants—and complete—i.e. are expressive enough to transform between any two schemas. They are implemented by hiding values from the database which can be performed online. Similar approaches are proposed for GemStone by *Penney* and Stone [Penney and Stein, 1987], for Sherpa by *Nguyen* and Rieu [Nguyen and Rieu, 1989], and for F2 by *Al-Jadir* [Al-Jadir and Léonard, 1998]. While GemStone supports only offline migration, Sherpa uses techniques known from artificial intelligence to automatically propagate changes of a class to its instances. To improve implementation and performance, F2 splits objects into multiple objects (multiobjects), distributing inherited attributes to objects specific to the class they were inherited from.

In addition to high-level operations [Brèche, 1996] that are composed of primitive operations [Zicari, 1991], *Ferrandina* et al. present operations to redefine the structure of a class as a whole in $O_2$ [Ferrandina et al., 1995]. To guarantee consistency between schema and data, a default migration function is associated to each class that has been modified. $O_2$ offers the possibility to overwrite the default migration functions by attaching custom migration functions encoded in a general-purpose language.

Besides a language to implement custom migrations, *SERF* (Schema evolution through an Extensible, Re-usable and Flexible framework) also provides a template mechanism to extend the predefined couplings with a new operation [Claypool et al., 1998]. However, the flexibility comes at the price that the migration can no longer be performed online. SERF has been applied to define reusable couplings for the evolution of uni- and bidirectional associations [Claypool et al., 2000]

**Schema Matching** approaches derive the in-place migration based on a matching between schema versions that is either recorded or detected. The matching is a declarative specification of the changes between the two schema versions.

*OTGen* (Object Transformer Generator) records changes performed on a schema by updating a transformation specification from which a migrator can be generated [Lerner and Habermann, 1990]. For each simple change applied to the schema, OTGen adds default rules to the transformation that preserve consistency between database and schema, and affect the database as little as possible. To support more complex migrations, the transformation can be manually modified using statements to initialize variables, perform context-dependent changes, move information, create objects and share information among objects.

*TESS* (Type Evolution Software System) derives migration rules by detecting changes between two schema versions [Lerner, 1997, Lerner, 2000]. The detection is based on a comparison algorithm with three stages that compare classes by their name, by their use sites or structurally. TESS allows its users to customize the comparison by selecting which stages are used, which classes are compared and which rules have to be acknowledged. TESS verifies whether the resulting migration rules are complete, i.e. cover the whole schema. TESS was evaluated by means of a case study and two

experiments.

**Class Versioning** approaches allow several versions of the same class to be present at the same time. They provide mechanisms to perform an out-of-place migration of instances from one class version to another.

*Skarra* and Zdonik propose to manage all versions of a class interface in a common version set interface [Skarra and Zdonik, 1986]. Additional error handling is added to existing classes, to prevent invalid (outside domain) and undefined properties. To support database migration, an object of a class can be transformed into an object of another as part of the interface.

*Monk* and Sommerville propose to use update and backdate functions on classes to allow for more flexible migration [Monk and Sommerville, 1993]. The update and backdate functions are user-defined with the query language CLOSQL, but applied automatically when needed. Combination of update and backdate functions allows objects of any class version to be transformed to any other version.

*SADES* (Semi-Autonomous Database Evolution System) employs aspect-orientation to make migration code independent of the evolved classes [Rashid and Sawyer, 2000]. Thereby, SADES can be easily adapted to different definitions of compliance of the objects to the class definitions. SADES was extensively evaluated by a qualitative and quantitative comparison to related approaches [Rashid and Sawyer, 2005].

**Schema Versioning** approaches version the schema as a whole in contrast to class versioning approaches.

*Kim* and Chou extend Bannerjee's approach for ORION to derive new schema versions instead of changing the schema [Kim and Chou, 1988]. Schema evolution is specified imperatively using the same operations, but new invariants and operations are necessary to manage schema versions. *Andany* et al. propose a similar approach for Farandole 2 which is also able to version sub schemas [Andany et al., 1991].

*Clamen* proposes to specify evolution declaratively by relating different schema versions [Clamen, 1994], which can also be used for schema integration. For each schema version, an interface is provided to objects and the interfaces compose the object's state into facets. Attributes can be shared between interfaces, independent of other interfaces, derived from other interfaces and dependent on other interfaces. Whenever an attribute value of a facet is modified, dependent attributes in other facets need to be updated.

*Lautemann* proposes an approach which specifies the evolution imperatively [Lautemann, 1996, Lautemann, 1997]. Migration between objects of different schema versions is specified by forward and backward migration functions. For certain schema changes, default migration functions are derived automatically, and can be overwritten by custom migration functions. *Bouneffa* presents a comparable approach, in which each object-schema version combination is represented by a facet [Bouneffa and Boudjlida, 1995]. User-defined mapping functions map objects from one facet into another.

**View-based** approaches use the view mechanism of database systems to simulate schema evolution. View definition languages provide a declarative way to specify schema evolution. The database is not modified, but is transformed out-of-place when calculating the view.

*Tresch* and Scholl are the first to propose database views as a means to manage schema evolution [Tresch and Scholl, 1993], as database migrations are expensive and break compatibility of existing applications. Views can be applied for capacity-preserving and capacity-reducing changes, but are not applicable for capacity-increasing transformations. They envision an implementation in COCOON using the view definition language COOL. *Brèche* et al. envision a similar approach for simulating schema changes in $O_2$ using VDL (View Definition Language) [Brèche et al., 1995].

*EVER* (EVolutionary ER diagrams) enhances the graphical constructs used in Entity Relational diagrams to be able to specify derivation relationships between schema versions [Liu et al., 1993, Liu et al., 1994]. EVER diagrams can be translated into relational or object-oriented database schemas. For each schema version, a consistent, updatable view is maintained. Therefor, the user has to specify derivation relationships between schema versions. For capacity-increasing changes, new attributes are added to the underlying database schema.

*TSE* (Transparent Schema Evolution) also supports capacity-increasing changes [Ra and Rundensteiner, 1995b, Ra and Rundensteiner, 1997]. Schema changes are recorded and mapped to views expressed in the view definition language MultiView. Thereby, each object instance can be accessed directly using different schema versions. Only for capacity-increasing changes, the actual database schema is changed. Besides the set of primitive changes known from other approaches, TSE was extended to handle more complex changes [Ra and Rundensteiner, 1995a]. To optimize the generated views after a lot of schema changes, obsolete views can be consistently removed [Crestana-Jensen et al., 2000].

**Hybrid** approaches combine several other approaches to unite their advantages.

*Benatallah* proposes a hybrid approach that combines schema versioning and schema modification [Benatallah, 1999]. When a schema change operation is applied, the user can decide whether the current schema version is modified or a new version is created. A language based on the standardized Object Query Language (OQL) is provided to define arbitrary migration semantics. Depending on whether the schema is modified or not, the migration specification is used to migrate the database in-place or out-of-place.

**Other Papers.** Casais [Casais, 1995] presents a survey of techniques to manage class evolution in object-oriented systems. On the class level, tailoring creates subclasses, surgery uses change primitives, versioning supports different versions of the same class, and reorganization performs more complex changes. On the object level, change avoidance prevents impact on objects, conversion modifies objects and filtering wraps objects.

Pons and Keller propose to organize operations in a multi-level catalog in which operations from higher levels are implemented using operations from lower lev-

els [Pons, 1997]. The catalog shows which modifications can be performed to the schema, starting from the primitives that a database system provides.

Li identifies the main issues in research on object-oriented schema evolution [Li, 1999]. The issues are semantic integrity consisting of referential integrity and consistency of constraints, schema evolvability encompassing structural and behavioral evolution, as well as application compatibility consisting of downward and upward compatibility.

Vermolen and Visser present a cross-space generalization of coupled evolution and propose a general solution based on a generated domain-specific transformation language for the meta level [Vermolen and Visser, 2008]. As application of the general solution, a coupled evolution tool set for object-oriented data models and relational databases is presented.

## 4.5 Grammarware

In the grammarware space, a *grammar* is an intensional definition of a *language*. An utterance of a language is called either *word* or *sentence*. Main programming languages, like e.g. Java, try to avoid the need for migration. New versions of such languages typically include older versions of the same language. However, there are a few approaches which address migration explicitly by coupled evolution. As is shown in Table 4.5, we group them into the two categories of grammar matching and operation-based approaches. Though originally proposed for modelware approaches [Rose et al., 2009], these categories fit here as well.

**Table 4.5:** Classification of the grammarware approaches

| | Evolution | | Migration | | | | | Evaluation |
|---|---|---|---|---|---|---|---|---|
| **Approach** | **Specific.** | **Source** | **Coupling** | **Language** | | **Target** | **Exec.** | |
| **Grammar matching** | | | | | | | | |
| TransformGen | *declarative* | *recorded* | overwritable | TL | | out | offline | case study |
| **Operation-based** | | | | | | | | |
| Lever | *imperative* | user-defined | extendable | GPL | Jython | out | offline | preliminary |

**Grammar Matching** approaches infer the migration from the matching between two grammar versions.

*TransformGen* infers the migration between two grammar versions from the recorded editing operations applied to a grammar [Staudt et al., 1987, Garlan et al., 1994]. The migration is specified as a transformation on the abstract syntax tree. Starting from an identity transformation, the transformation is altered, when editing operations are applied to the grammar. Additionally, the transformation can be customized by the user. Thereby, a static analysis helps to prevent errors. TransformGen was applied to evolve the tree-oriented programming language ARL.

**Operation-based** approaches approaches provide coupled operations that encapsulate grammar evolution and word migration.

*Lever* is an operation-based approach that provides a suite of operations coupling grammar evolution with word migration [Juergens and Pizka, 2006, Pizka and Juergens, 2007b, Pizka and Juergens, 2007a]. Furthermore, it supports the migration of compilers. Lever comes with three DSLs embedded in a scripting language: One for grammar evolution, one for word migration, and another one offering abstractions on top of the other two for defining coupled operations. The user specifies grammar evolution imperatively by a sequence of operation applications. It has been evaluated using a fictitious evolution of a catalog description language.

**Other Papers.** Lämmel presents an operation suite just for grammar evolution [Lämmel, 2001]. A similar operation suite is used in a lightweight verification method to maintain the correspondence between grammar versions [Lämmel and Zaytsev, 2009a]. The method is used to recover grammar relationships in different releases of the Java Language Standard [Lämmel and Zaytsev, 2009b]. Though these suites come without a coupling for migration, this coupling can be added by defining the effects of operations on the word level.

Overbey and Johnson discuss a side effect, when programming languages evolve but migration is avoided [Overbey and Johnson, 2009]. In this case, old programs use outdated constructs instead of new and better constructs which were introduced later. They study the effect for the evolution of Fortran and Java and envision a refactoring-based solution to the problem. When the language evolves, language engineers provide refactorings which replace the old constructs with the new and better ones. In a next step, these refactorings can be coupled with operations at the grammar level.

## 4.6 XMLware

In XMLware, a *schema* is an intensional definition of a *language* or *format*. An utterance of a language is called a *document*. Schemas are expressed in schema languages like *DTD* [W3C, 2008] or *XML Schema* [Walmsley, 2001], both recommended by the World Wide Web Consortium. Table 4.6 lists the XMLware approaches as well as their classification. We distinguish manual specification and operation-based approaches. Again, these categories where originally proposed for modelware approaches [Rose et al., 2009], but fit for XMLware approaches as well.

**Table 4.6:** Classification of the XMLware approaches

| | | Evolution | | Migration | | | | | Evaluation |
|---|---|---|---|---|---|---|---|---|---|
| **Approach** | | **Specific.** | **Source** | **Coupling** | **Language** | | **Target** | **Exec.** | |
| **Manual specification** | | | | | | | | | |
| Tan | XSD | declarative | user-defined | *overwritable* | *custom* | | out | offline | — |
| **Operation-based** | | | | | | | | | |
| XEM | DTD | *imperative* | user-defined | fixed | GPL | | in | offline | — |
| Lämmel | DTD | | | | TL | XSLT | out | | |
| X-Evolution | XSD | | | overwritable | GPL | XQuery | in | | |

**Manual Specification** approaches require the user to manually specify the migration

of documents from one schema version to another

*Tan* and Goh propose an extension for XML Schema to specify declaratively the differences to previous versions directly in the schema [Tan and Goh, 2005]. Additions, removals, moves, and renames of elements and attributes are supported. The information is then used for migrating documents between different schema versions.

**Operation-based** approaches provide a set of reusable coupled operations. The user specifies schema evolution imperatively by a sequence of operation applications. Since the operations work at the schema level as well as at the document level, such a sequence specifies both schema evolution and document migration.

*XEM* (XML Evolution Manager) addresses the evolution of DTD schemas [Su et al., 2001]. It provides a complete, minimal and sound suite of primitive operations. At the schema level, these operations work on DTD schemas represented as labeled graphs. At the document level, they operate on labeled ordered trees.

*Lämmel* and Lohmann suggest transformation operations for DTD schemas from which migrations for documents are induced [Lämmel and Lohmann, 2001]. The effect of the operations at the schema level are described as informal text, whereas the migrations are specified by XSLT. The operations preserve the well-formedness of both DTD schemas and XML documents. Moreover, the operations are classified whether they preserve, extend, or reduce the structure of XML documents.

*X-Evolution* is a tool addressing the evolution of schemas defined in XML Schema [Mesiti et al., 2006, Guerrini et al., 2007]. Like XEM, it provides a complete, minimal and sound suite of primitive operations. At the schema level, the operations work on schemas represented as labeled trees. At the document level, an incremental validation algorithm performs a minimal number of insertions, modifications and deletions to make a document valid again. To overwrite this default migration, a domain-specific language (DSL) provides means for the specification of custom migrations [Guerrini and Mesiti, 2008]. This DSL extends the standardised XQuery Update.

## 4.7 Modelware

In the modelware space, a *metamodel* is an intensional definition of a *modeling language*. An utterance of a modeling language is called a *model*. Metamodels are expressed in a metamodeling formalism like *MOF* as standardized by the Object Management Group [Object Management Group, 2006a], *Ecore* of the Eclipse Modeling Framework [Steinberg et al., 2009], or *MetaGME* of the Generic Modeling Environment [Ledeczi et al., 2001]. All these formalisms provide object-oriented means similar to UML class diagrams [Object Management Group, 2009].

In [Rose et al., 2009], Rose et al. compare different approaches to automate model migration in response to metamodel evolution. They identify three categories of approaches: manual specification, metamodel matching, and operation-based approaches. We take this comparison which is restricted to *Ecore* as a metamodeling formalism as a starting point, but consider the other metamodeling formalisms as well. Table 4.7 lists all the modelware approaches and groups them according to the three categories.

**Table 4.7:** Classification of the modelware approaches

<table>
<tr><th colspan="2" rowspan="2">Approach</th><th colspan="3">Evolution</th><th colspan="5">Migration</th><th rowspan="2">Evaluation</th></tr>
<tr><th>Specific.</th><th colspan="2">Source</th><th>Coupling</th><th colspan="2">Language</th><th>Target</th><th>Exec.</th></tr>
<tr><td colspan="11">Manual specification</td></tr>
<tr><td>Sprinkle</td><td>GME</td><td rowspan="2">declarative</td><td colspan="2" rowspan="2">user-defined</td><td rowspan="3">overwritable</td><td colspan="2" rowspan="3">custom</td><td rowspan="3">out</td><td rowspan="3">offline</td><td>preliminary</td></tr>
<tr><td>MCL</td><td>GME</td><td>—</td></tr>
<tr><td>Flock</td><td>Ecore</td><td colspan="3">—</td><td>comparison</td></tr>
<tr><td colspan="11">Metamodel matching</td></tr>
<tr><td>Gruschko</td><td>Ecore</td><td rowspan="4">declarative</td><td rowspan="4">det.</td><td rowspan="2">simple</td><td rowspan="2">overwritable</td><td>TL</td><td>ETL</td><td rowspan="4">out</td><td rowspan="4">offline</td><td>—</td></tr>
<tr><td>Geest</td><td>MS DSL</td><td>GPL</td><td>C#</td><td>case study</td></tr>
<tr><td>Cicchetti</td><td>Ecore</td><td rowspan="2">complex</td><td>fixed</td><td rowspan="2">TL</td><td rowspan="2">ATL</td><td>—</td></tr>
<tr><td>AML</td><td>Ecore</td><td>extendable</td><td>case study</td></tr>
<tr><td colspan="11">Operation-based</td></tr>
<tr><td>Hößler</td><td>MOF</td><td rowspan="2">imperative</td><td colspan="2" rowspan="2">user-defined</td><td rowspan="2">fixed</td><td colspan="2">—</td><td rowspan="2">out</td><td rowspan="2">offline</td><td rowspan="2">—</td></tr>
<tr><td>Wachsmuth</td><td>MOF</td><td>TL</td><td>QVT</td></tr>
</table>

**Manual Specification** approaches provide custom model transformation languages to manually specify the model migration. Thereby, specific model migration constructs reduce the effort for building a migration specification. For instance, migrations automatically copy model elements whose metamodel definition has not changed. The user then overwrites this default behavior with the intended migration.

*Sprinkle* introduces a visual language to declaratively specify the differences between two versions of a GME-based metamodel [Sprinkle, 2003, Sprinkle and Karsai, 2004]. *MCL* (Model Change Language) is another visual migration language targeting GME [Narayanan et al., 2009]. With both languages, the user does not only specify the metamodel differences, but defines a model migration based on these differences. This overwrites the default copying behaviour. The migration is performed out-of-place and offline. MCL permits a number of idioms that—according to the authors' experience—cover most common migration cases. Migration algorithms not covered by MCL can be specified imperatively using a C++ API. Sprinkle's approach is evaluated by an experience report about its application in an industrial context.

*Flock* is a textual migration language for EMF-based models [Rose et al., 2010d]. Here, only the model migration is specified. Differences between metamodel versions are not made explicit. Instead, Flock automatically copies only those model elements which conform to the evolved metamodel. The user then iteratively redefines the migration specification to migrate non-conforming elements. Using the well-known Petri net example [Wachsmuth, 2007], Flock has been compared to migration specifications in model transformation languages ATL and Ecore2Ecore as well as to the language underlying our approach that is presented in Chapter 5 (*COPE – Coupled Evolution of Metamodels and Models*).

**Metamodel Matching** approaches automatically detect the differences between two metamodel versions. These are stored in a declarative difference model from which a migration specification is generated.

*Gruschko* et al. support the automatic detection of simple changes in Ecore metamodels [Gruschko, 2006, Gruschko et al., 2007, Becker et al., 2007]. They propose auto-

matic migration steps for resolvable changes and envision to support the user in overwriting the migration for unresolvable changes. The approach is only prototypically implemented and thus not yet evaluated.

*Geest* applies a similar approach in the context of Microsoft DSL Tools [Geest et al., 2008]. The difference model is obtained by a, possibly human-aided, comparison of the metamodel versions. Only simple changes can be detected and the generated migration specification can be overwritten. The approach has been evaluated on evolving metamodels from the Web Service Software Factory (WSSF).

*Cicchetti* et al. also detect complex changes [Cicchetti et al., 2008]. Here, the difference model consists of simple changes which are interpreted in terms of complex changes. The migration specification consists of a set of model transformations to be executed consecutively. Since this is prevented by interdependent changes, Cicchetti et al. characterize dependencies between complex changes [Cicchetti et al., 2009].

*AML* (Atlas Matching Language) allows the user to parameterize the detection of complex changes [Garcés et al., 2009]. Therefore, the user combines existing or user-defined heuristics to a matching algorithm. From a difference model obtained by such an algorithm, an ATL transformation specifying the migration is automatically generated. The approach was evaluated on the well-known Petri net example [Wachsmuth, 2007], and on the Java metamodel from NetBeans.

**Operation-based** approaches provide—similar to corresponding grammarware and XMLware approaches—a set of reusable coupled operations that work at the metamodel level as well as at the model level.

*Hößler* et al. formalize a fixed suite of reusable coupled operations [Hößler et al., 2005]. Operations are grouped into metamodel extensions, projections and factorings based on their effect on the extension. The completely theoretical approach is based on a generic instance model supporting versioning and is neither implemented nor evaluated.

*Wachsmuth* presents an operation suite for the MOF metamodeling formalism [Wachsmuth, 2007]. Based on ideas from grammar evolution [Lämmel, 2001], operations are classified according to language and model preservation properties. For migration, the evolution specification is translated into a QVT Relations model transformation.

**Other Papers.** Street and Pettit analyzed the evolution of UML from version 1.4 to 2.0 [Street and Pettit, 2005]. They classified changes to the UML metamodel into additions, modifications, and deletions. Most of the changes were additions which allow UML users to improve existing models. Required migrations for modifications and deletions could be mostly automated.

## 4.8 Cross-Space Comparison

**Increasing Interest.** Coupled evolution is a topic of increasing interest. It first drew attention in the dataware space where it reached a publication peak in the 1990s. In

the same decade, coupled evolution spread into the grammarware space, before it found its way to XMLware and modelware in the last decade. Though being a new topic in the modelware space, coupled evolution now draws most attention in this technical space. Figure 4.7 illustrates the increasing interest in coupled evolution over the last decades as well as its spreading over the various technical spaces.

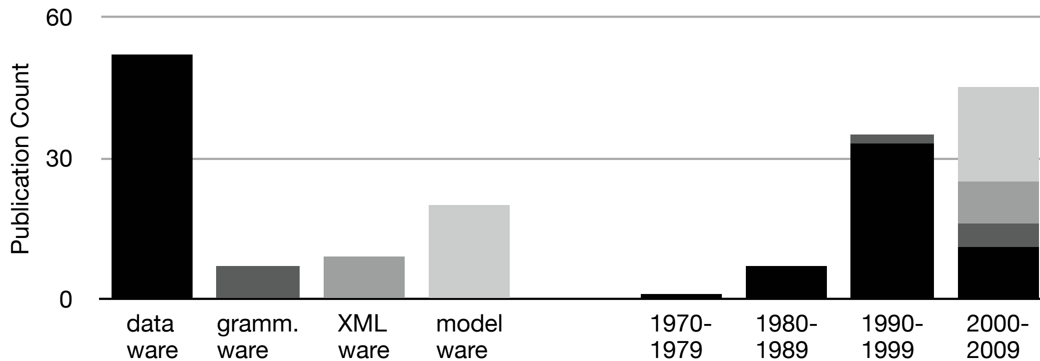



**Figure 4.7:** Interest in coupled evolution

**Categories of Approaches.** With the classification scheme from Section 4.3 (*Classification of Approaches*), we are able to classify approaches from the different technical spaces along the same criteria. Furthermore, we found approaches from all technical spaces to fit into categories which were originally proposed by Rose et al. for the modelware space [Rose et al., 2009]. These categories are *manual specification*, *matching* approaches, and *operation-based* approaches. The unique feature of manual specification approaches is a *custom* migration language for *overwriting* a default migration manually. For matching approaches, the unique feature is a *declarative* evolution specification which is either *recorded* or *detected*. The unique feature of operation-based approaches is an *imperative* evolution specification as a sequence of operation applications. Additionally, we found the categories of *class versioning* approaches, *schema versioning* approaches, and *view-based* approaches to be restricted to approaches dealing with object-oriented databases in the dataware space.

**Specifics in Technical Spaces.** In relational dataware we noticed a stronger and more explicit focus on the impact of schema evolution on other artifacts than the database—mostly queries.

To not affect the operation of the database system, most approaches from the dataware spaces focus on *online* migration of the database. Approximately half of the dataware approaches reorganizes the database, the other half avoids reorganization by versioning or views. In contrast, all approaches from the modelware space perform *offline* migration. Models are mainly used at development time, and metamodels for runtime models typically do not evolve during an application run. Thus, there is no need to migrate models online.

Most modelware approaches perform *out-of-place* migration. Model transformation languages generally do not support in-place transformations with different source and target metamodels. However, in the dataware space, in-place migrations are a

more commonly chosen alternative, when the size of the data set makes out-of-place migration hard or impossible.

Object-oriented dataware is the technical space where most of the approaches come from, followed by the modelware space. Both spaces rely on object-oriented concepts which complicates evolution and migration compared to other technical spaces. Finally, 75% of the approaches lack proper evaluation.

## 4.9 Motivation of our Approach

The analysis of the existing approaches helps us to motivate our approach targeting the modelware space. We can identify issues of existing modelware approaches and learn from approaches in the other technical spaces. The categories of approaches that are interesting for modelware are *manual specification*, *operation-based* as well as *matching* approaches. *Versioning* and *view-based* approaches known from object-oriented dataware are not interesting for modelware, as they avoid to physically migrate elements and employ specific technologies only available in this technical space. In the following, we motivate our approach by revisiting the requirements defined in Chapter 1 (*Introduction*) as well as by identifying issues in existing approaches by means of the classification.

### 4.9.1 Requirements

We analyze the candidate approaches with respect to the requirements of automation and semantics preservation.

**Automation.** Our first goal is to automate model migration in response to metamodel evolution as far as possible. In Chapter 3 (*State of the Practice: Automatability of Model Migration*), we have shown that an approach is appropriate if it is able to reuse recurring migrations and at the same time expressive enough to cater for complex migrations. *Manual specification* approaches only provide reuse by copying model elements that are not affected by metamodel evolution. However, they provide an expressive language in which the user can manually specify the model migration for model elements that are affected by metamodel evolution. *Matching* approaches support reuse by the detection patterns for which they are able to automatically infer a model migration. However, it is impossible for them to detect complex model migrations without additional information. *Operation-based* approaches reuse recurring combinations of metamodel evolution and model migration by encapsulating them in operations. The provided operations are usually not expressive enough to cover all possible semantics of model migration. As the object-oriented dataware space demonstrates, operation-based approaches can be combined with manual specification approaches to accommodate reusable operations with custom operations. *However, in the modelware space, there is not yet an operation-based approach that provides support for manually specifying custom operations.*

**Semantics Preservation.** Our second goal is to ensure semantics preservation during model migration as far as possible. *Manual specification* approaches foster cor-

rectness of the model migration, as they give the user full control over the migration semantics. *Matching* approaches completely automate the derivation of a model migration from a metamodel evolution, and thus may not lead to a correct model migration. However, both categories of approaches suppose that the metamodel evolution has already been carried out, and only the metamodel versions are available. If a lot of metamodel changes have been performed, the intention behind the metamodel evolution is already lost in the evolution process. To not lose the intention behind the metamodel evolution, *operation-based* approaches allow the user to assemble the model migration by incrementally applying coupled operations. By recording the model migration together with the metamodel evolution, operation-based approaches are best suited for ensuring semantics preservation. *However, in the modelware space, there is not yet an operation-based approach that provides support for recording the model migration.*

### 4.9.2 Classification

We identify issues in the existing approaches of the modelware space by analyzing their classification. From these issues, we derive the classification of the approach that we target which is shown in Table 4.8.

**Table 4.8:** Classification of our approach

| Approach | | Evolution | | Migration | | | | Evaluation |
|---|---|---|---|---|---|---|---|---|
| | | Specific. | Source | Coupling | Language | Target | Exec. | |
| **Operation-based** | | | | | | | | |
| Our approach | Ecore | imperative | recorded | extendable | new TL | in-place | offline | case study |

**Evolution.** Since we decided in favor of an operation-based approach due to the requirements, our approach needs to specify the evolution *imperative*ly. To preserve the model migration together with the metamodel evolution, we need to use a *recorded* evolution as source. However, recording approaches are relatively rare in all technical spaces, and especially in the modelware space, there is not yet a recording approach.

**Migration.** While matching approaches in the modelware space allow their users to overwrite and extend couplings, the existing operation-based approaches do not yet provide comparable mechanisms. However, the history of the operation-based approaches in the object-oriented dataware space shows that operation-based approaches can be *overwritable* and *extendable*. As a consequence, we need to transfer these ideas—which are necessary to fulfill the requirements—to the modelware space. Since they are based on existing exogenous model transformation languages, all the approaches in the modelware section only perform the migration out-of-place. However, the operation-based approaches in the dataware space usually perform the migration *in-place*, due to the better performance. To support in-place migration, we cannot use an existing transformation language, but need to build a new one. All the approaches in the modelware space execute the migration only *offline*, and not online like most of the approaches of the dataware space. However, online migration is not needed, since models do not need to be as highly available as databases.

**Evaluation.** The survey showed that most coupled evolution approaches are not regularly evaluated by applying them to real-life evolutions. This is common to all technical spaces, but especially dataware which is the most researched technical space does not provide much empirical evidence. Manual specification and matching approaches are evaluated more often than operation-based approaches, as they can be easier applied to already existing evolutions of intensional definitions. Therefore, even if operation-based approaches are the most promising category for fulfilling the goals, we cannot be sure that they really work in practice. In a nutshell, empirical evidence is necessary to demonstrate the applicability of operation-based approaches in practice.

## 4.10 Summary

In each technical space, various approaches to coupled evolution have been proposed. But it is largely unknown how these approaches relate to each other. To alleviate this, we performed a systematic literature review on coupled evolution approaches. We were able to derive a classification scheme which is independent of technical spaces, that is we can classify approaches from the different technical spaces along the same criteria. We showed how the various approaches can be classified according to this model. We then relied on this classification to determine commonalities and differences between the approaches. Finally, we motivated our approach from the issues of the existing modelware approaches as well as from the lessons learned from other technical spaces.

We decided in favor of an operation-based approach which records the coupled evolution as a sequence of operations. Each coupled operation encapsulates a combination of metamodel evolution and reconciling model migration. From the identified approaches, operation-based approaches are best suited to address both the challenges of automation and semantics preservation. Concerning automation, coupled operations can be either specified manually—to cater for complex migrations, or reused—to automate recurring migrations. Concerning semantics preservation, the intention behind the metamodel evolution is preserved by recording the model migration together with the metamodel evolution. The operation-based approaches currently existing in the modelware space do not support manual specification of coupled operations, recording the coupled evolution, in-place migration and are not yet implemented nor evaluated.

Chapter 5

# COPE – Coupled Evolution of Metamodels and Models

Currently, to our best knowledge, there is no approach that combines both the desired level of reuse and expressiveness for defining model migrations. To alleviate this, we present COPE, an integrated approach to model the coupled evolution of metamodels and models. COPE is based on a language that provides means to combine metamodel adaptation and model migration into so-called coupled operations. The stated requirements are fulfilled by two kinds of coupled operations: reusable and custom coupled operations. A reusable coupled operation allows the reuse of recurring coupled operations across metamodels. COPE already comes with an extensive library of reusable coupled operations that cover many migration semantics. A custom coupled operation can be manually defined by the language engineer for complex migrations that are specific to a metamodel. A language history keeps track of the consecutively performed coupled operations. This chapter is partly based on [Herrmannsdoerfer et al., 2008b], [Herrmannsdoerfer et al., 2009a], [Herrmannsdoerfer and Ratiu, 2009], [Herrmannsdoerfer and Ratiu, 2010] and [Herrmannsdoerfer et al., 2010b].

**Contents**


**5.1 COPE in a Nutshell . . . 106**

**5.2 Library of Reusable Coupled Operations . . . 116**

**5.3 Language to Specify the Coupled Evolution . . . 131**

**5.4 Limitations of Automating Model Migration . . . 140**

**5.5 Summary . . . 148**


In Section 5.1 (*COPE in a Nutshell*), we present the principles behind COPE. We introduce the extensive library of reusable coupled operations in Section 5.2 (*Library of Reusable Coupled Operations*). In Section 5.3 (*Language to Specify the Coupled Evolution*), we explain in more detail the language to implement the reusable and custom coupled operations. We identify limitations of automating model migration in Section 5.4 (*Limitations of Automating Model Migration*), before we conclude in Section 5.5 (*Summary*).

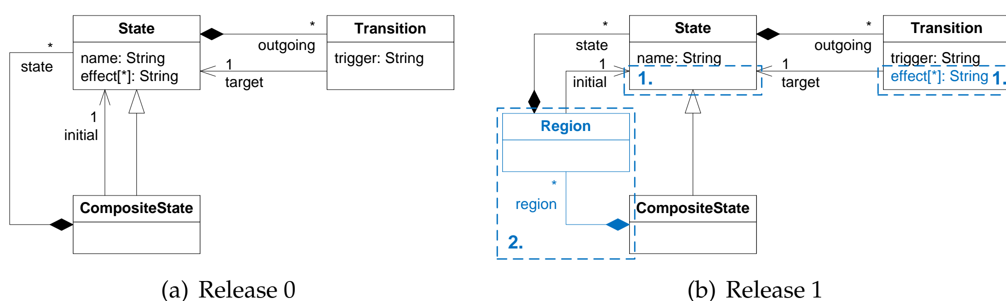
## 5.1 COPE in a Nutshell

In this section, we give an overview over COPE's language to specify the coupled evolution of metamodels and models. This language provides concepts to fulfill both requirements presented in Chapter 3 (*State of the Practice: Automatability of Model Migration*): reuse of recurring migration knowledge and expressiveness to cater for metamodel-specific migrations. Reuse is provided by an abstraction mechanism that allows language engineers to encapsulate both metamodel adaptation and model migration in a metamodel-independent way. Expressiveness is provided by embedding primitives for metamodel adaptation and model migration into a Turing-complete language.

From our experience, language engineers prefer to use the metamodel editor over specifying the coupled evolution in this language. Consequently, COPE provides further abstraction from this language by a non-invasive integration into a metamodel editor. For simplicity of presentation, in this chapter, we restrict ourselves to the language, and present the tool support in Chapter 6 (*Tool Support*).

### 5.1.1 Running Example

Throughout this section, we use a state machine metamodel as a running example. Figure 5.1 shows the metamodel before and after adaptation as a UML class diagram [Object Management Group, 2009]. In release 0 of the metamodel, a State has a name and may be decomposed into sub states through its subclass CompositeState. A Transition belongs to its source state and refers to a target state, and is activated by a triggering event. When a state is entered, a sequence of actions is performed as effect, and in case of a composite state, an initial state is entered. The trigger thus defines the input that the state machine consumes, and the effect the output that the state machine produces.




(a) Release 0 (b) Release 1

**Figure 5.1:** Running example adaptation

For release 1 of the metamodel, the following adaptations are performed[1]:

1. The state machine is changed from a Moore to a Mealy machine. In Moore machines, the effect of the state machine only depends on the current state [Moore, 1956]. In contrast, the effect of the state machine depends also on the

[1] In Figure 5.1, the differences are indicated by numbered, dashed boxes.

trigger in Mealy machines [Mealy, 1976]. Therefore, we move the attribute effect from State to Transition.

2. Regions are introduced to support concurrency within states. Therefore, we insert the class Region. We further introduce the new composite reference region so that a composite state can define a number of concurrent regions. Finally, we move the composite reference state and the reference initial to the new class Region, as regions are now composed of sub states.

In the following, we subsequently specify the coupled evolution in COPE's language in order to be able to migrate existing models in response to these adaptations.

### 5.1.2 Incremental Coupled Evolution

In practice, a modeling language is evolved by incremental adaptations to the metamodel. There are a number of primitive metamodel changes like create element, rename element, delete element, and so on. One or more such primitive changes compose a specific metamodel adaptation, like in our example the introduction of regions. COPE allows language engineers to attach information about how to migrate corresponding models in response to a metamodel adaptation. Consequently, the intended model migration can already be captured while adapting the metamodel, thus preventing the loss of intention. In COPE, such a combination of metamodel adaptation and model migration is called *coupled operation:*

**Definition 5.1** (Coupled Operation)**.** *A coupled operation $co$ is a tuple $(adm, mig)$ with*

- *a metamodel adaptation $adm : \mathcal{MM} \cup \{\bot\} \rightarrow \mathcal{MM} \cup \{\bot\}$, and*
- *a model migration $mig : \mathcal{M} \rightarrow \mathcal{M}$.*

*It is applicable to a metamodel $mm \in \mathcal{MM}$ if and only if it produces a defined metamodel, i.e. $adm(mm) \neq \bot$. $\mathcal{CO}$ denotes the set of all possible coupled operations, i.e. $co \in \mathcal{CO}$.*

In particular, $adm(\bot) = \bot$ always holds. Coupled operations can be easily composed by simply sequencing them:

**Definition 5.2** (Sequential Composition)**.** *The sequential composition $co_2 \circ co_1$ of two coupled operations $co_1 = (adm_1, mig_1)$ and $co_2 = (adm_2, mig_2)$ yields a coupled operation $co = (adm, mig)$ with*

- *metamodel adaptation $adm = adm_2 \circ adm_1$, and*
- *model migration $mig = mig_2 \circ mig_1$.*

A coupled operation is modular in the sense that the corresponding model migration can be specified independently of any coupled operation that was executed before or after the coupled operation. Due to their modularity, a comprehensive evolution can be decomposed into manageable coupled operations, thus ensuring scalability. The notion of coupled operation qualifies to fulfill the requirements of reuse and expressiveness. Certain coupled operations can be reused resulting in *reusable coupled operations*, while others have to be specified manually resulting in *custom coupled operations*.

**Example 5.1** (Sequential Composition). *Figure 5.2 illustrates how coupled operations can be used to compose the coupled evolution of our running example. The first coupled operation changes the state machine metamodel from a Moore to a Mealy machine. Since the corresponding model migration is specific to the metamodel as explained in Section 3.2.3 (Model-Independent, Metamodel-Specific Coupled Change), it has to be performed by a custom coupled operation. The last two coupled operations introduce concurrent regions to the metamodel and are invocations of reusable coupled operations. The invocation of* Extract Class *extracts the sub states including the initial state of a composite state into the new class* Region*. The invocation of* Generalize Reference *generalizes the multiplicity of the new reference from* CompositeState *to* Region *to enable concurrent regions.*

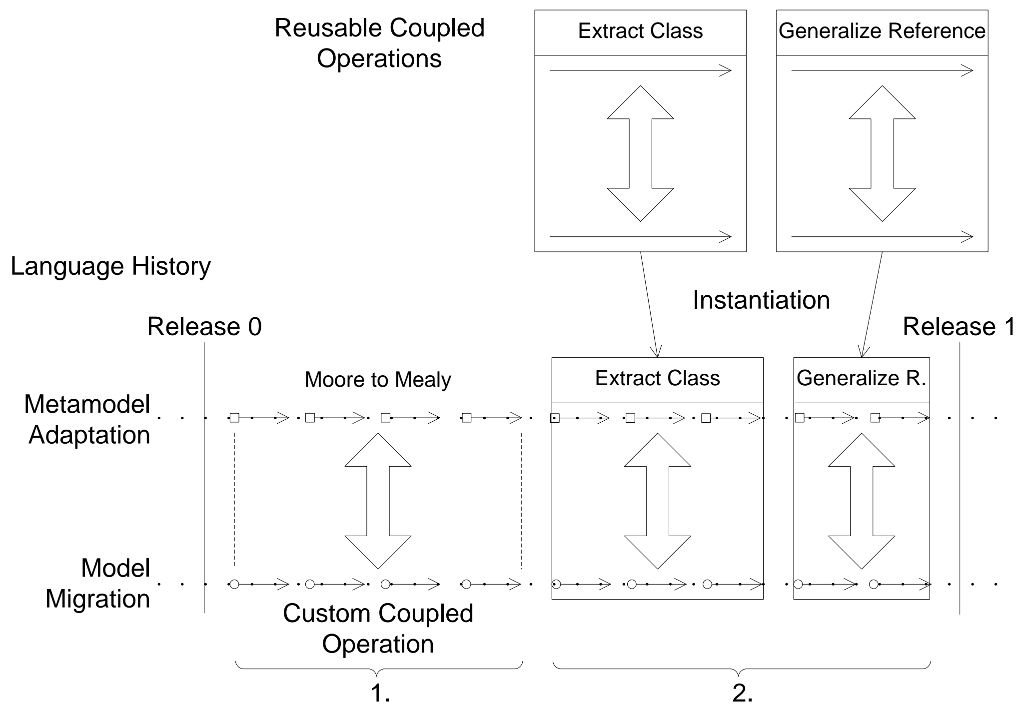



**Figure 5.2:** Language history for the running example

Keeping track of the coupled operations that lead from one metamodel release to the next results in a *language history*. The language history contains enough information to migrate a model from the metamodel version to which it conforms to any subsequent metamodel release. Hence, it is particularly suited to migrate models which are not accessible while performing the metamodel adaptation. This is the case when the modeling language and the models are developed by different distributed parties.

**Example 5.2** (Language History). *Figure 5.2 illustrates the language history for our running example which consists of the sequence of coupled operations together with markers for the different releases.*

### 5.1.3 Coupled Operations

Usually, the metamodel adaptation is manually performed in the metamodeling tool used for authoring the metamodel. The model migration can be manually encoded

as a model transformation which transforms the old model to a new model conforming to the adapted metamodel. As explained in Section 2.5.5 (*Model Transformation for Model Migration*), we distinguish between *exogenous* and *endogenous* model transformation, depending on whether source and target metamodel of the transformation are different or not [Mens and Van Gorp, 2006]. Exogenous model transformation requires to specify the mapping of all elements from the source to the target metamodel. As typically only a subset of metamodel elements are modified by the metamodel adaptation, a model migration specified as an exogenous transformation contains a high fraction of identity rules. Concerning this aspect, endogenous transformation is better suited to the nature of model migration, as it only has to address those metamodel elements for which the model needs to be modified. However, endogenous transformation requires the source and the target metamodel to be the same which is not the case for metamodel evolution. Hence, conventional languages for model transformation are not well suited to specify a model migration.

Instead, model migration is best served by a language that directly combines the properties of both exogenous and endogenous model transformation: one needs to be able to specify the transformation from a source metamodel to a different target metamodel, but only for the metamodel elements for which a migration is required. To achieve this, we propose to soften the conformance between metamodel and its model during coupled evolution: the metamodel can first be adapted regardless of its models, and the model can then be migrated to the adapted metamodel. As a consequence, only the differences need to be specified for both metamodel adaptation and model migration. However, softening the conformance during model migration comes at the price that a model may not always conform to its metamodel. To ensure conformance after a certain change to metamodel and model, we require a coupled operation to enforce the following properties:

**Definition 5.3** (Preservation of Metamodel Conformance)**.** *A coupled operation* $co = (adm, mig)$ *preserves the conformance of a metamodel* $mm \in \mathcal{MM}$ *to the metametamodel if the adapted metamodel* $adm(mm)$ *conforms to the metametamodel, in case the original metamodel* $mm$ *conforms to the metametamodel:*

$$mm \in \mathcal{MM} \Rightarrow adm(mm) \in \mathcal{MM}$$

**Definition 5.4** (Preservation of Model Conformance)**.** *A coupled operation* $co = (adm, mig)$ *preserves the conformance of a model* $m \in \mathcal{M}$ *to the metamodel* $mm \in \mathcal{MM}$ *if the migrated model* $mig(m)$ *conforms to the adapted metamodel* $adm(mm)$*, in case the original model* $m$ *conforms to the original metamodel* $mm$*:*

$$m \models mm \Rightarrow mig(m) \models adm(mm)$$

Both metamodel and model conformance thus have to hold only at operation boundaries, i.e. the metamodel may not conform to the metametamodel or the model may not conform to the metamodel during an operation.

We have implemented COPE on top of the Eclipse Modeling Framework (EMF) [Steinberg et al., 2009] which is one of the most widely used metamodeling tools (see Section 2.2.6 (*Eclipse Modeling Framework*)). In this implementation, the model conformance is softened by a generic instance model which can express every model

and is only used during migration. The implementation dynamically ensures metamodel and model conformance at operation boundaries, i.e. when the operation is executed on a metamodel and model. To specify both metamodel adaptation and model migration, COPE provides a number of expressive primitives which operate on the generic instance model. These primitives can be invoked from within the general-purpose scripting language Groovy [Koenig et al., 2007] in order to take advantage of its expressiveness. For more information about the generic instance model and a complete list of the primitives, we refer the reader to Section 5.3 (*Language to Specify the Coupled Evolution*).

### 5.1.4 Custom Coupled Operations

Expressiveness is provided by custom coupled operations, which have to be specified manually by the language engineer. Since custom coupled operations are specified manually for a certain metamodel change, they are not universally applicable:

**Definition 5.5** (Custom Coupled Operation)**.** *A custom coupled operation is a coupled operation* $co = (adm, mig)$ *that is not universally applicable:*

- *there is a metamodel to which the coupled operation is not applicable:*

$$\exists mm' \in \mathcal{MM} : adm(mm') = \bot$$

- *it provides a specific model migration for each metamodel to which it is applicable:*

$$\forall mm \in \mathcal{MM} : adm(mm) \in \mathcal{MM} \implies mig \in \{\mathcal{L}_{mm} \to \mathcal{L}_{adm(mm)}\}$$

In doing so, the language engineer can apply a number of primitives for both metamodel adaptation and model migration. The primitives are complete in the sense that every possible metamodel adaptation as well as model migration can be specified with them. Completeness can be shown by first destroying the source metamodel or model, and then rebuilding the target metamodel or model from scratch as done in [Banerjee et al., 1987] for database schema evolution. As these primitives are embedded into the Turing-complete scripting language Groovy, the resulting language is expressive enough to even cater for very specific model migrations.

**Example 5.3** (Custom Coupled Operation)**.** *Listing 5.1 shows the custom coupled operation that was performed to change the state machine from a Moore to a Mealy machine. More specifically, the depicted custom coupled operation consists of a metamodel adaptation and a reconciling model migration. This example also shows that we only have to specify the differences for both metamodel and model in this language.*

Metamodel adaptation. *The metamodel adaptation only moves the attribute* `effect` *from class* `State` *to class* `Transition` *(line 3). Note that—even though the method is called* `add`*—the attribute is also removed from the old parent class, since* `eStructuralFeatures` *is a composition, allowing a feature to be contained only once. The attribute is assigned to the variable* `effectAttribute` *in order to be able to access its values for states (line 2), even though the attribute is no longer known to the class* `State`*. Note how metamodel elements can be accessed by means of fully qualified names (e.g.* `State.effect`*).*

**Listing 5.1:** Custom coupled operation Moore to Mealy

```
// metamodel adaptation
def effectAttribute = State.effect
Transition.eStructuralFeatures.add(effectAttribute)

// model migration
getEffect = { transition ->
  def effect = []
  def state = transition.target
  effect.addAll(state.get(effectAttribute))
  while(state.instanceOf(CompositeState)) {
    effect.addAll(state.initial.get(effectAttribute))
    state = state.initial
  }
  return effect
}

for(transition in Transition.allInstances) {
  def effect = getEffect(transition)
  transition.effect = effect
}

for(state in State.allInstances) {
  state.unset(effectAttribute)
}
```

Model migration. *A Moore machine is migrated to a Mealy machine by moving the effect of each state to its incoming transitions. However, in the advent of composite states as well as initial states, the model migration is more involved. When a state machine transitions to a composite state, it not only enters the composite state but also its initial state. Consequently, we also have to take the effect of the initial state into account when calculating the effect of the transition. Note that this may have to be applied recursively, as the initial state may again be a composite state, and so on. The model migration encoded in COPE's language is thus divided into two passes: First, we set the* effect *for each transition based on the states (lines 17-20), and then we remove the* effect *from each state (lines 22-24). The language provides the primitive* allInstances *to be able to iterate over all instances of a certain type (lines 17 and 22). The* effect *of a transition is set by using* transition.effect = effect *which is a short form for* transition.set(Transition.effect, effect) *(line 19). The* effect *of a transition is calculated by means of the helper method* getEffect *(lines 6-15). As explained before, the* effect *consists of the effect of the transition's* target *state as well as the effects of the* initial *states.* transition.target *is the short form for* transition.get(Transition.target) *(line 8). However, the short forms can only be used, in case a feature of that name is currently defined by the instance's type. As the attribute* effect *is no longer defined for class* State*, we thus have to use* state.get(effectAttribute) *to be able to access the effect of a state (line 9). Furthermore, the primitive* instanceOf *can be used to check whether a state is of type* CompositeState *(line 10). The effect of a state is removed by using a primitive to* unset *the* effectAttribute *(line 23).*

### 5.1.5 Reusable Coupled Operations

Reuse is provided by an abstraction mechanism to generalize coupled operations into so-called reusable coupled operations:

**Definition 5.6** (Reusable Coupled Operation)**.** *A reusable coupled operation $rco$ is a parameterized function that is defined by*

- *a set of possible parameter assignments $\mathcal{P}$, and*
- *a function $rco : \mathcal{P} \rightarrow \mathcal{CO}$ that maps a parameter assignment $p \in \mathcal{P}$ to a coupled operation $rco(p) = (adm, mig)$.*

*It is thus applicable to multiple metamodels and in different contexts within a metamodel.*

Reusable coupled operations are specified independently of the metamodel and encapsulate both metamodel adaptation and reconciling model migration. They can be reused across metamodels, thus promising to significantly reduce effort associated with metamodel adaptation and model migration. COPE allows language engineers to declare new reusable coupled operations and make them available through a *library*. The language employs the abstraction mechanism of procedures in Groovy in order to declare reusable coupled operations. A reusable coupled operation is declared independently of the specific metamodel by means of parameters. The types of the parameters are thus elements from the metametamodel which is Ecore in our EMF-based implementation. Reusable coupled operations can be instantiated by invoking the procedure with parameters assigned to specific metamodel elements. The applicability of a reusable coupled operation can be restricted by preconditions in the form of assertions.

**Example 5.4** (Reusable Coupled Operation)**.** *The introduction of concurrent regions can be implemented completely with reusable coupled operations.*

Instantiation. *Listing 5.2 shows the invocation of the reusable coupled operations* Extract Class *and* Generalize Reference*, which correspond to the second adaptation in our example history.* Extract Class *is invoked to extract the references* state *and* initial *from* CompositeState *to the new class* Region *(line 1). Then, the extracted region is accessible from a composite state through the new single-valued containment reference named* region*.* Generalize Reference *is invoked to increase the multiplicity of this new reference in order to enable multiple concurrent regions (line 2). Note that by invoking reusable coupled operations, the language engineer does not have to specify neither metamodel adaptation nor model migration.*

**Listing 5.2:** Instantiation of reusable coupled operations

```
1 extractClass([CompositeState.state, CompositeState.initial],
      "Region", "region")
2 generalizeReference(CompositeState.region, Region, 1, INF)
```

Declaration. *Listing 5.3 shows the declaration of the reusable coupled operation* Extract Class *which we just invoked to introduce regions into our example metamodel. This reusable coupled operation—which recurred several times in our case studies—extracts a number of features from a* context class *to a new class. The extracted class is accessible from the context*

*class through a new single-valued containment reference. The reusable coupled operation declares parameters for the attributes and references to be extracted (*features*), the name of the new class (*className*) and the name of the new reference (*referenceName*) (line 1). Several preconditions in the form of assertions restrict the applicability of the reusable coupled operation, e.g. every feature has to belong to the same context class (lines 6-9).*

**Listing 5.3:** Declaration of reusable coupled operation Extract Class

```
1  extractClass = {List<EStructuralFeature> features, String
       className, String referenceName ->
2
3    def EClass contextClass = features[0].eContainingClass
4
5    // preconditions
6    assert features.every{feature -> feature.eContainingClass ==
         contextClass} :
7      "The features have to belong to the same class"
8    assert contextClass.getEStructuralFeature(referenceName) == null
         || features.contains(contextClass.getEStructuralFeature(
         referenceName)) :
9      "A feature with the same name already exists"
10
11   // metamodel adaptation
12   def extractedClass = newEClass(className)
13   def reference = contextClass.newEReference(referenceName,
         extractedClass, 1, 1, CONTAINMENT)
14   extractedClass.eStructuralFeatures.addAll(features)
15
16   // model migration
17   for(contextInstance in contextClass.allInstances) {
18     def extractedInstance = extractedClass.newInstance()
19     contextInstance.set(reference, extractedInstance)
20     for(feature in features) {
21       extractedInstance.set(feature, contextInstance.unset(
             feature))
22     }
23   }
24 }
```

Metamodel adaptation. *The metamodel adaptation creates the* extracted class *(line 12) and the new single-valued containment* reference *from the context class to the extracted class (line 13). Then, the extracted features are moved from the context class to the extracted class (line 14). For the metamodel adaptation, we use the primitives of the metametamodel implementation together with some high-level primitives to create new metamodel elements (e.g.* newEClass*). The reusable coupled operation is simplified in the sense that it leaves out the package in which the extracted class is created.*

Model migration. *The model migration pretty much modifies the model accordingly. For each instance of the context class (*contextInstance*), a* new instance *of the extracted class is created (line 18) and associated to the context instance through the new reference (line 19). Then, all the values of the extracted features are moved from the context instance to the new instance (lines 20-22). Note that due to the generic instance model the context instance's value of a feature can still be accessed by the* unset *method, even though the feature has*

*already been moved to the extracted class (line 21).*

### 5.1.6 Classification of Coupled Operations

Coupled operations can be classified according to several properties. We are interested in language preservation, model preservation, and bidirectionality. Therefore, we stick to a simplified version of the terminology from [Wachsmuth, 2007].

**Language Preservation.** A metamodel is an intensional definition of a modeling language. Its extension is a set of conforming models. When a coupled operation is applied to a metamodel, the metamodel adaptation has an impact on the language defined by the metamodel—the set of conforming models. We distinguish different classes of operations according to this impact [Wachsmuth, 2007]:

**Definition 5.7** (Refactoring / Constructor / Destructor). *A coupled operation $co = (adm, mig)$ is called a refactoring / constructor / destructor if there exists a bijective / injective / surjective mapping between the languages defined by the original and evolved metamodel:*

$$\forall mm \in \mathcal{MM} : adm(mm) \neq \bot \implies$$
$$\exists map \in \{\mathcal{L}_{mm} \to \mathcal{L}_{adm(mm)}\} : map \text{ bijective / injective / surjective}$$

*A reusable coupled operation $rco : \mathcal{P} \to \mathcal{CO}$ is called a refactoring / constructor / destructor if $rco(p)$ is a refactoring / constructor / destructor for all $p \in \mathcal{P}$.*

Language preservation properties can be used to reason about the impact on language expressiveness. In the technical space of grammarware [Klint et al., 2005], operations have been successfully used in [Lämmel and Zaytsev, 2009b] to reason about relationships between different versions of the Java grammar.

**Model Preservation.** Model preservation properties indicate whether migration is needed:

**Definition 5.8** (Model-Preserving). *A coupled operation $co = (adm, mig)$ is called model-preserving if all models conforming to the original metamodel also conform to the evolved metamodel without migration:*

$$mig = id \wedge \forall mm \in \mathcal{MM},\ adm(mm) \neq \bot : \mathcal{L}_{mm} \subseteq \mathcal{L}_{adm(mm)}$$

*A reusable coupled operation $rco : \mathcal{P} \to \mathcal{CO}$ is called model-preserving if $rco(p)$ is model-preserving for all $p \in \mathcal{P}$.*

Thus, model-preserving operations do not require migration. An operation is *model-migrating* if models conforming to an original metamodel might need to be migrated in order to conform to the evolved metamodel.

**Definition 5.9** (Safely Model-Migrating). *A coupled operation $co = (adm, mig)$ is called safely model-migrating if the migration preserves distinguishability, i.e. different models*

*(conforming to the original metamodel) are migrated to different models (conforming to the evolved metamodel):*

$$\forall mm \in \mathcal{MM},\ adm(mm) \neq \perp : mig \textit{ injective for } \mathcal{L}_{mm} \to \mathcal{L}_{adm(mm)}$$

*A reusable coupled operation $rco : \mathcal{P} \to \mathcal{CO}$ is called safely model-migrating if $rco(p)$ is safely model-migrating for all $p \in \mathcal{P}$.*

In contrast, an *unsafely model-migrating* operation might yield the same model when migrating two different models. Model preservation properties thus indicate to what extent the model migration preserves information in models. If loss of information should be avoided, the language engineer should only apply model-preserving or safely model-migrating operations.

Classification of operations with respect to model preservation is related to the classification with respect to language preservation: Refactorings and constructors are either model-preserving or safely model-migrating operations, as the migration can be injective. Destructors are unsafely model-migrating operations, as the migration cannot be injective.

**Bidirectionality.** Another property we are interested in is the reversibility of adaptation and migration. Bidirectionality properties indicate that an operation can be safely undone on the metamodel and model level [Marschall, 2005]. On the metamodel level, the adaptation needs to be undone:

**Definition 5.10** (Inverse). *A coupled operation $co_2 = (adm_2, mig_2)$ is called the inverse of another operation $co_1 = (adm_1, mig_1)$ iff their sequential composition does not change the metamodel:*

$$adm_2 \circ adm_1 = id$$

*A reusable coupled operation $rco_2 : \mathcal{P}_2 \to \mathcal{CO}$ is called the inverse of another operation $rco_1 : \mathcal{P}_1 \to \mathcal{CO}$ iff $rco_2$ is the inverse of $rco_1$ for all parameter settings:*

$$\forall p_1 \in \mathcal{P}_1, \exists p_2 \in \mathcal{P}_2 : rco_2(p_2) \textit{ is inverse of } rco_1(p_1)$$

A reusable coupled operation may also be undone by itself:

**Definition 5.11** (Self Inverse). *A reusable coupled operation $rco : \mathcal{P} \to \mathcal{CO}$ is called a self inverse iff a second application of the operation—possibly with other parameters—always yields the original metamodel:*

$$\forall p \in \mathcal{P},\ rco(p) = (adm, mig) : \exists p' \in \mathcal{P},\ rco(p') = (adm', mig') : adm' \circ adm = id$$

On the model level, the migration also needs to be undone:

**Definition 5.12** (Safe Inverse). *A coupled operation $co_2 = (adm_2, mig_2)$ is called the safe inverse of another coupled operation $co_1 = (adm_1, mig_1)$ iff $co_2$ is the inverse of $co_1$ and their sequential composition $co_2 \circ co_1$ is model-preserving. A reusable coupled operation $rco_2 : \mathcal{P}_2 \to \mathcal{CO}$ is called the safe inverse of another operation $rco_1 : \mathcal{P}_1 \to \mathcal{CO}$ iff $rco_2$ is the safe inverse of $rco_1$ for all parameter settings:*

$$\forall p_1 \in \mathcal{P}_1,\ \exists p_2 \in \mathcal{P}_2 : rco_2(p_2) \textit{ safe inverse of } rco_1(p_1)$$

Bidirectionality can be used to invert an evolution that has been specified erroneously earlier. Performed operation applications can be undone with different levels of safety by applying the inverse operations.

## 5.2 Library of Reusable Coupled Operations

The success of an operation-based approach highly depends on the library of reusable coupled operations it provides [Rose et al., 2009]. The library of an operation-based approach needs to fulfill a number of requirements. A library should seek completeness so as to be able to cover a large set of coupled evolution scenarios. However, the higher the number of reusable coupled operations, the more difficult it is to find an operation in the library. Consequently, a library should also be organized in a way that it is easy to select the right reusable coupled operation for the change at hand.

### 5.2.1 Origins of Reusable Coupled Operations

The reusable coupled operations are either motivated from the literature or from the case studies that we performed.

**Literature.** First, reusable coupled operations originate from the literature on coupled evolution in modelware, object-oriented dataware and APIware.

*Modelware.* Wachsmuth first proposes an operation-based approach for metamodel evolution and classifies a set of operations according to the preservation of metamodel expressiveness and existing models [Wachsmuth, 2007]. Gruschko et al. envision a matching approach and therefore classify all primitive changes according to their impact on existing models [Becker et al., 2007, Burger and Gruschko, 2010]. Cicchetti et al. list a set of composite changes which they are able to detect using their metamodel matching approach [Cicchetti et al., 2008].

*Object-oriented dataware.* Banerjee et al. present a complete and sound set of primitives for schema evolution in the object-oriented database system ORION and characterize the primitives according to their impact on existing databases [Banerjee et al., 1987]. Brèche introduces a set of high-level operations for schema evolution in the object-oriented system $O_2$ and shows how to implement them in terms of primitive operations [Brèche, 1996]. Pons and Keller propose a three-level catalog of operations for object-oriented schema evolution which groups operations according to their complexity [Pons, 1997]. Claypool et al. list a number of primitives for the adaptation of relationships in object-oriented systems [Claypool et al., 2000].

*APIware.* Fowler presents a catalog of operations for the refactoring of object-oriented code [Fowler, 1999]. Dig and Johnson show—by performing a case study—that most changes on object-oriented code can be captured by a rather small set of refactoring operations [Dig and Johnson, 2006].

**Case Studies.** Second, reusable coupled operations originate from the case studies that we performed as part of this thesis.

*BMW.* In Chapter 3 (*State of the Practice: Automatability of Model Migration*), we present an empirical study on the evolution of two industrial metamodels from BMW Car IT that shows that most of the changes can be captured by reusable coupled operations: Flexible User Interface Development (*FLUID*) for the specification of automotive user

interfaces, and Test Automation Framework - Generator (*TAF-Gen*) for the generation of test cases for these user interfaces.

*Evaluation.* In Chapter 7 (*Case Studies*), we present a number of case studies that evaluate our approach COPE. COPE has been used to reverse engineer the operation history of a number of metamodels: Palladio Component Model (*PCM*) for the specification of software architectures (see Section 7.1 (*GMF Generator Model and Palladio Component Model*)), and Graphical Modeling Framework (*GMF*) for the model-based development of diagram editors (see Section 7.2 (*Graphical Modeling Framework*)). Currently, COPE is applied to forward engineer the operation history of a number of metamodels: *Unicase* for UML modeling and project management (see Section 7.4 (*Unicase Unified Model*)), and *Quamoco* for modeling the quality of software products (see Section 7.3 (*Quamoco Quality Metamodel*)). We also participated with COPE in the migration case of the Transformation Tool Contest (*TTC*) (see Section 7.5 (*Transformation Tool Contest*)).

### 5.2.2 Overview of the Library

In the following, we present a library of 61 reusable coupled operations that we consider complete for practical application. First, we included all coupled operations found in nine related papers as well as all coupled operations identified by performing seven real-life case studies. Second, we added coupled operations that transfer the semantics of existing coupled operations to other metamodeling constructs, if possible. Third, we ensured that an inverse operation is included in the catalog for each coupled operation.

**Organization.** In the following, we explain the coupled operations in groups which help users to navigate the catalog. Figure 5.3 illustrates the organization of the library. We start with *primitive* operations which perform an atomic metamodel evolution step that can not be further subdivided. Here, we distinguish *structural* primitives which create and delete metamodel elements and *non-structural* primitives which modify existing metamodel elements. Afterwards, we continue with *composite* operations. These can be decomposed into a sequence of primitive operations which has the same effect at the metamodel level but not necessarily at the model level. We group complex operations according to the metamodeling techniques they address—distinguishing *specialization and generalization*, *inheritance*, and *delegation* operations—as well as their semantics—distinguishing *replacement*, and *merge and split* operations.

**Conventions.** Each group is discussed separately in the subsequent subsections. For each group, a table provides an overview over all operations in the group. Using the classifications from Section 5.1.6 (*Classification of Coupled Operations*), the table classifies each reusable coupled operation according to language preservation into refactoring (r), constructor (c) and destructor (d) as well as according to model preservation into model-preserving (p), safely (s) and unsafely (u) model-migrating. The table further indicates the safe (s) and unsafe (u) inverse of each operation by referring to its number. Finally, each paper and case study has a column in each table. A bullet point (•) in such a column denotes the occurrence of the operation in the corresponding paper or case study. Papers are referred to by citation, while case

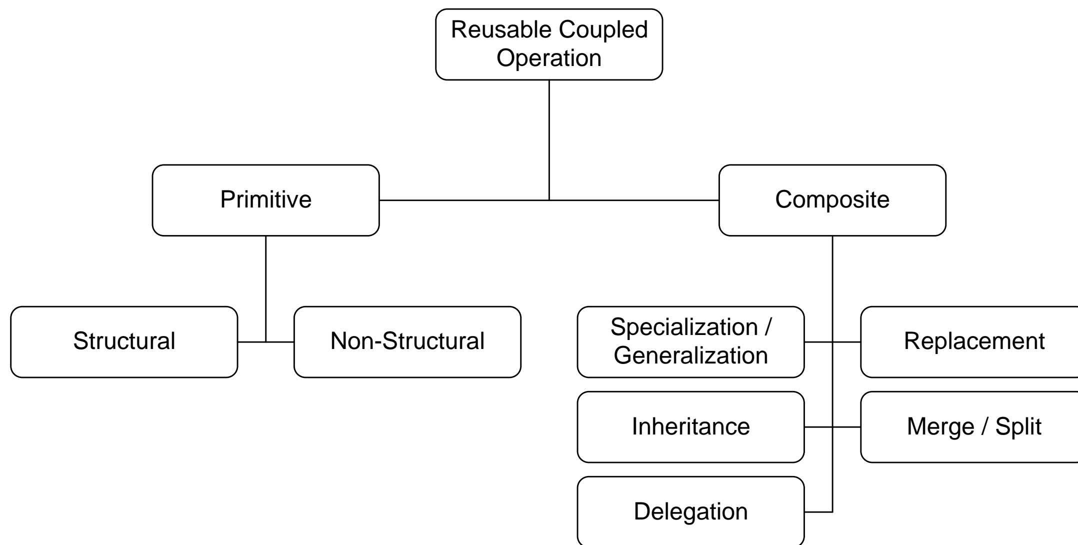



**Figure 5.3:** Organization of the library

studies are referred to by the abbreviations given in Section 5.2.1 (*Origins of Reusable Coupled Operations*). For each coupled operation, we discuss its semantics in terms of metamodel evolution and model migration.

### 5.2.3 Structural Primitives

Structural primitive operations modify the structure of a metamodel, i.e. create or delete metamodel elements. Creation operations are parameterized by the specification of a new metamodel element, and deletion operations by an existing metamodel element.

**Creation** of non-mandatory metamodel elements (packages, classes, optional features, enumerations, literals and data types) is model-preserving. Creation of mandatory features is safely model-migrating. It requires initialization of the features' values using default values or default value computations.

**Deletion** of metamodel elements requires deleting instantiating model elements, such as objects and links, by the migration. However, deletion of model elements poses the risk of migration to inconsistent models: For example, deletion of objects may cause links to non-existent objects, and deletion of references may break object containment. Therefore, deletion operations are bound to metamodel level restrictions: Packages may only be deleted, when they are empty. Classes may only be deleted, when they are outside inheritance hierarchies and are targeted neither by non-composite references nor by mandatory composite references. Several complex operations discussed in subsequent subsections can deal with classes not meeting these requirements. References may only be deleted, when they are neither composite, nor have an opposite. Enumerations and data types may only be deleted, when they are not used in the metamodel and thus obsolete.

Deletion operations of elements which may have been instantiated in the model (with

**Table 5.1:** Structural primitives

| # | Operation Name | Classific.: Language preservation | Classific.: Model preservation | Classific.: Inverse | Modelw.: [Wachsmuth, 2007] | Modelw.: [Becker et al., 2007] | Modelw.: [Cicchetti et al., 2008] | OO dataware: [Banerjee et al., 1987] | OO dataware: [Brèche, 1996] | OO dataware: [Pons, 1997] | OO dataware: [Claypool et al., 2000] | APIw.: [Fowler, 1999] | APIw.: [Dig and Johnson, 2006] | BMW: FLUID | BMW: TAF-Gen | Evaluation: PCM | Evaluation: GMF | Evaluation: Unicase | Evaluation: Quamoco | Evaluation: TTC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Create Package | r | p | 2s | | • | | | | | | | | | • | • | | | | |
| 2 | Delete Package | r | p | 1s | | • | | | | | | | | | • | | | | | |
| 3 | Create Class | c | p | 4s | • | • | • | • | | | | | | • | • | • | • | • | • | |
| 4 | Delete Class | d | u | 3u | • | • | • | • | • | | | | • | • | | • | | • | • | |
| 5 | Create Attribute | c | s | 7s | • | • | • | • | | | | | | • | • | • | • | • | • | |
| 6 | Create Reference | c | s | 7s | • | • | • | • | | | | | | • | • | • | • | • | • | |
| 7 | Delete Feature | d | u | 5/6u | • | • | • | • | | | | | | • | • | • | • | • | • | • |
| 8 | Create Opposite Reference | d | u | 9u | | • | | | | | • | • | | | | | • | • | • | |
| 9 | Delete Opposite Reference | c | p | 8s | | • | | | | | • | • | | | • | | | | | |
| 10 | Create Data Type | r | p | 11s | | • | | | | | | | | | | | | | | |
| 11 | Delete Data Type | r | p | 10s | | • | | | | | | | | | | | | | • | |
| 12 | Create Enumeration | r | p | 13s | | • | | | | | | | | | • | • | • | • | • | |
| 13 | Delete Enumeration | r | p | 11s | | • | | | | | | | | | | | | • | | |
| 14 | Create Literal | c | p | 15s | | • | | | | | | | | | | | | • | | |
| 15 | Merge Literal | d | u | 14u | | • | | | | | | | | | | | • | | | |

the exception of *Delete Opposite Reference*) are unsafely model-migrating due to loss of information. Deletion provides a safe inverse to its associated creation operation. Since deletion of metamodel elements which may have been instantiated in a model is unsafely model-migrating, creation of such elements provides an unsafe inverse to deletion—lost information cannot be restored.

**Opposite References.** Creating and deleting references which have an opposite are different from other creation and deletion operations. *Create Opposite Reference* restricts the set of valid links and is thus an unsafely model-migrating destructor, whereas *Delete Opposite Reference* removes a constraint from the model and is thus a model-preserving constructor.

**Data Types, Enumerations and Literals.** *Create / Delete Data Type* and *Create / Delete Enumeration* are refactorings, as restrictions on these operations prevent usage of created or deleted elements. Deleting enumerations and data types is thus model-preserving. *Merge Literal* deletes a literal and replaces its occurrences in a model by another literal. Merging a literal provides a safe inverse to *Create Literal*.

### 5.2.4 Non-Structural Primitives

Non-structural primitive operations modify a single, existing metamodel element, i.e. change properties of a metamodel element. All non-structural operations take the affected metamodel element—their subject—as parameter.

**Table 5.2:** Non-structural primitives

| | | **Classific.** | | | **Modelw.** | | | **OO dataware** | | | | **APIw.** | | **BMW** | | **Evaluation** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **#** | **Operation Name** | Language preservation | Model preservation | Inverse | [Wachsmuth, 2007] | [Becker et al., 2007] | [Cicchetti et al., 2008] | [Banerjee et al., 1987] | [Brèche, 1996] | [Pons, 1997] | [Claypool et al., 2000] | [Fowler, 1999] | [Dig and Johnson, 2006] | FLUID | TAF-Gen | PCM | GMF | Unicase | Quamoco | TTC |
| 1 | Rename | r | s | 1s | • | • | • | • | | | | • | • | • | • | • | • | • | • | • |
| 2 | Change Package | r | s | 2s | | • | | | | | | | | • | • | • | | • | | |
| 3 | Make Class Abstract | d | u | 4u | | • | | | | | | | | • | | • | | | • | • |
| 4 | Drop Class Abstract | c | p | 3s | | • | | | | | | | | | | | • | | • | |
| 5 | Add Supertype | c | p | 6s | | • | | • | | | | | | | • | • | • | • | • | |
| 6 | Remove Supertype | d | u | 5u | | • | | • | | | | | | | • | • | • | • | • | |
| 7 | Make Attribute Identifier | d | u | 8u | | • | | | | | | | | • | | | | | | |
| 8 | Drop Attribute Identifier | c | p | 7s | | • | | | | | | | | • | | | • | | | |
| 9 | Make Reference Composite | d | u | 10u | | • | | | | | | • | | • | • | | • | | • | |
| 10 | Switch Reference Composite | c | s | 9s | | • | | • | | | | • | | • | • | | | | • | |
| 11 | Make Reference Opposite | d | u | 12u | | • | | | | | • | | | | • | | | | | |
| 12 | Drop Reference Opposite | c | p | 11s | | • | | | | | • | | | | | | • | | • | |

*Change Package* can be applied to both package and type. Additionally, the value-changing operations *Rename*, *Change Package* and *Change Attribute Type* are parameterized by a new value. *Make Class Abstract* requires a subclass parameter indicating to which class objects need to be migrated. *Switch Reference Composite* requires an existing composite reference as target.

**Naming and Packaging.** Packages, types, features and literals can be renamed. *Rename* is safely model-migrating and finds a self-inverse in giving a subject its original name back. *Change Package* changes the parent package of a package or type. Like renaming, it is safely model-migrating and a safe self-inverse.

**Classes** can be made abstract, requiring migration of objects to a subclass, because otherwise, links targeting the objects may have to be removed. Consequently, mandatory features that are not available in the superclass have to be initialized to default values. *Make Class Abstract* is unsafely model-migrating, due to loss of type information, and has an unsafe inverse in *Drop Class Abstract*.

Super type declarations may become obsolete and may need to be removed. *Remove Supertype* S from a class C implies removing values of features inherited from S. Additionally, references targeting type S, referring to objects of type C, need to be removed. To prevent breaking multiplicity restrictions, *Remove Supertype* is restricted to types S which are not targeted by mandatory references—neither directly, nor through inheritance. The operation is unsafely model-migrating and can be unsafely inverted by *Add Super Type*.

**Attributes** defined as identifier need to have unique values. *Make Attribute Identifier* requires a migration which ensures uniqueness of the attribute's values and is thus

unsafely model-migrating. In contrast, *Drop Attribute Identifier* removes the uniqueness restriction and is thus model-preserving.

**References** can have an opposite and can be composite. An opposite reference declaration defines the inverse of the declaring reference. References combined with a multiplicity restriction on the opposite reference restrict the set of valid links. *Make Reference Opposite* needs a migration to make the reference set satisfy the added multiplicity restriction. The operation is thereby unsafely model-migrating. *Drop Reference Opposite* removes cardinality constraints from the link set and does not require migration, thus being model-preserving.

*Make Reference Composite* ensures containment of referred objects. Since all referred objects were already contained by another composite reference, all objects must be copied. To ensure the containment restriction, copying has to be recursive across composite references (deep copy). Furthermore, to prevent cardinality failures on opposite references, there may be no opposite references to any of the types of which objects are subject to deep copying. *Switch Reference Composite* changes the containment of objects to an existing composite reference. If objects of a class A were originally contained in class B through composite reference b, *Switch Reference Composite* changes containment of A objects to class C, when it is parameterized by reference b and a composite reference c in class C. After applying the operation, reference b is no longer composite. *Switch Reference Composite* provides an unsafe inverse to *Make Reference Composite*.

### 5.2.5 Specialization / Generalization Operations

Specializing a metamodel element reduces the set of possible models, whereas generalizing expands the set of possible models. Generalization and specialization can be applied to features and super type declarations. All specialization and generalization operations take two parameters: a subject and a generalization or specialization target. The first is a metamodel element and the latter is a class or a multiplicity (lower and upper bound).

Generalization of features does not only generalize the feature itself, but also generalizes the metamodel as a whole. Feature generalizations are thus model-preserving constructors. Generalizing a super type declaration may require removal of feature values and is only unsafely model-migrating. Feature specialization is a safe inverse of feature generalization. Due to the unsafe nature of the migration resulting from feature specialization, generalization provides an unsafe inverse to specialization. Super type generalization is an unsafe inverse of super type specialization which is a safe inverse vice versa.

**Attributes.** *Specialize Attribute* either reduces the attribute's multiplicity or specializes the attribute's type. When reducing multiplicity, either the lower bound is increased or the upper bound is decreased. When specializing the type, a type conversion maps the original set of values onto a new set of values conforming to the new attribute type. Specializing type conversions are surjective. *Generalize Attribute* extends the attribute's multiplicity or generalizes the attribute's type. Generalizing

**Table 5.3:** Generalization / Specialization operations

| | | Classific. | | | Modelw. | | | OO dataware | | | | APIw. | | BMW | | Evaluation | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | **Operation Name** | Language preservation | Model preservation | Inverse | [Wachsmuth, 2007] | [Becker et al., 2007] | [Cicchetti et al., 2008] | [Banerjee et al., 1987] | [Brèche, 1996] | [Pons, 1997] | [Claypool et al., 2000] | [Fowler, 1999] | [Dig and Johnson, 2006] | FLUID | TAF-Gen | PCM | GMF | Unicase | Quamoco | TTC |
| 1 | Generalize Attribute | c | p | 2s | • | • | • | | | | | | | • | • | • | • | | • | |
| 2 | Specialize Attribute | d | u | 1u | • | • | • | | | | | | | • | | • | • | | • | |
| 3 | Generalize Reference | c | p | 4s | • | • | • | | | | | | | • | • | | • | | • | |
| 4 | Specialize Reference | d | u | 3u | • | • | • | | | | | | | • | • | • | • | | • | • |
| 5 | Specialize Composite Reference | d | u | 3u | | | | | | | | | | • | | • | | | • | |
| 6 | Generalize Supertype | d | u | 7u | | • | | | | | | | | | | • | | | | |
| 7 | Specialize Supertype | c | s | 6s | | • | | | | • | | | | • | • | • | • | | • | |

an attribute's type involves an injective type conversion. Type conversions are generally either implemented by transformations for each type to an intermediate format (e.g. by serialization) or by transformations for each combination of types. The latter is more elaborate to implement, yet less fragile. Most generalizing type conversions from type x to y have a specializing type conversion from type y to x as safe inverse. Applying the composition vice versa yields an unsafe inverse.

**References.** Similar to attributes, reference multiplicity can be specialized and generalized. *Specialize / Generalize Reference* can additionally specialize or generalize the type of a reference by choosing a sub type or super type of the original type, respectively. Model migration of reference specialization requires deletion of links not conforming the new reference type. *Specialize Composite Reference* is a special case of reference specialization at the metamodel level, which requires contained objects to be migrated to the targeted subclass at the model level, to ensure composition restrictions. *Specialize Composite Reference* is unsafely model-migrating.

**Supertypes.** Super type declarations are commonly adapted, while refining a metamodel.

**Example 5.5** (Specialize / Generalize Supertype)**.** *Consider the example shown in Figure 5.4, in which classes A, B and C are part of a linear inheritance structure and remain unadapted.*

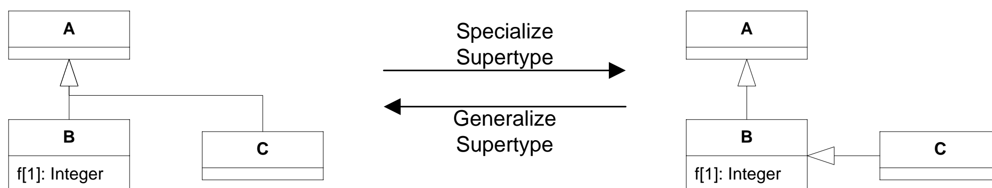



**Figure 5.4:** Operations *Specialize / Generalize Supertype*

*From left to right,* Specialize Supertype *changes a declaration of super type A on class C to*

*B, a sub type of A. Consequently, a mandatory feature f is inherited, which needs the setting of values by the migration. In general, super type specialization requires addition of feature values which are declared mandatory by the new super type. From right to left,* Generalize Supertype *changes a declaration of super type B on class C to A, a super type of B. In the new metamodel, feature f is no longer inherited in C. Values of features which are no longer inherited need to be removed by the migration. Furthermore, links to objects of A that target class B, are no longer valid, since A is no longer a sub type of B. Therefore, these links need to be removed, if multiplicity restrictions allow, or adapted otherwise.*

### 5.2.6 Inheritance Operations

Inheritance operations move features along the inheritance hierarchy. Most of them are well-known from refactoring object-oriented code. There is always a pair of a constructor and destructor, where the destructor is the safe inverse of the constructor, and the constructor is the unsafe inverse of the destructor.

**Table 5.4:** Inheritance operations

| | | **Classific.** | | | **Modelw.** | | | **OO dataware** | | | | **APIw.** | | **BMW** | | **Evaluation** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **#** | **Operation Name** | Language preservation | Model preservation | Inverse | [Wachsmuth, 2007] | [Becker et al., 2007] | [Cicchetti et al., 2008] | [Banerjee et al., 1987] | [Brèche, 1996] | [Pons, 1997] | [Claypool et al., 2000] | [Fowler, 1999] | [Dig and Johnson, 2006] | FLUID | TAF-Gen | PCM | GMF | Unicase | Quamoco | TTC |
| 1 | Pull up Feature | c | p | 2s | • | | • | | | | | • | • | • | • | | • | | • | |
| 2 | Push down Feature | d | u | 1u | • | | • | | | | | • | • | | • | | • | | | |
| 3 | Extract Superclass | c | p | 4s | • | | • | | • | • | | • | • | • | • | | • | | • | |
| 4 | Inline Superclass | d | u | 3u | • | | • | | | • | | • | • | | • | | • | • | • | • |
| 5 | Fold Superclass | c | s | 6s | | | | | | | | | | • | | | | | • | |
| 6 | Unfold Superclass | d | u | 5u | | | | | | | | | | | | | • | | • | |
| 7 | Extract Subclass | c | s | 8s | | | | | • | • | | • | | | | | • | | • | |
| 8 | Inline Subclass | d | u | 7u | | | | | | • | | • | | | • | | | | • | |

**Pull up / Push down Feature.** *Pull up Feature* is a constructor which moves a feature that occurs in all subclasses of a class to the class itself. For migration, values for the pulled up feature are added to objects of the class and filled with default values. The corresponding destructor *Push down Feature* moves a feature from a class to all its subclasses. While objects of the subclasses stay unaltered, values for the original feature must be removed from objects of the class itself.

**Extract / Inline Superclass.** *Extract Superclass* is a constructor which introduces a new class, makes it the superclass of a set of classes, and pulls up one or more features from these classes. The corresponding destructor *Inline Superclass* pushes all features of a class into its subclasses and deletes the class afterwards. References to the class are not allowed but can be generalized to a superclass in a previous step. Objects of the class need to be migrated to objects of the subclasses. This might require the addition of values for features of the subclasses.

**Fold / Unfold Superclass.** The constructor *Fold Superclass* is related to *Extract Superclass*. Here, the new superclass is not created but exists already. This existing class has a set of (possibly inherited) features. In another class, these features are defined as well. The operation then removes these features and adds instead an inheritance relation to the intended superclass. In the same way, the destructor *Unfold Superclass* is related to *Inline Superclass*. This operation copies all features of a superclass into a subclass and removes the inheritance relation between both classes.

**Example 5.6** (Fold / Unfold Superclass)**.** *Figure 5.5 depicts an example for both operations.*

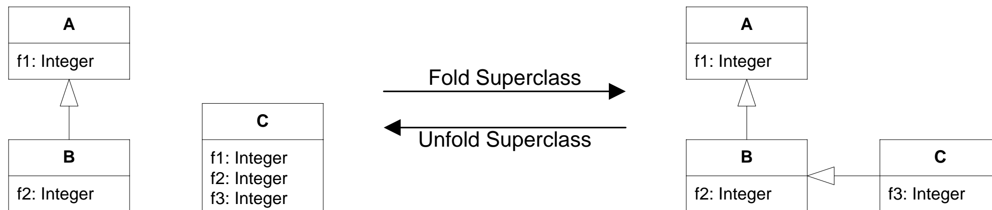



**Figure 5.5:** Operations *Fold / Unfold Superclass*

*From left to right, the superclass* B *is folded from class* C *which includes all the features of* B*. These features are removed from* C*, and* B *becomes a superclass of* C*. From right to left, the superclass* B *is unfolded into class* C *by copying features* f1 *and* f2 *to* C*.* B *is not longer a superclass of* C*.*

**Extract / Inline Subclass.** The constructor *Extract Subclass* introduces a new class, makes it the subclass of another, and pushes down one or more features from this class. Objects of the original class must be converted to objects of the new class. The corresponding destructor *Inline Subclass* pulls up all features from a subclass into its non-abstract superclass and deletes the subclass afterwards. References to the class are not allowed but can be generalized to a superclass in a previous step. Objects of the subclass need to be migrated to objects of the superclass.

### 5.2.7 Delegation Operations

Delegation operations move metamodel elements along compositions or ordinary references. Most of the time, they come as pairs of corresponding refactorings being safely inverse to each other.

**Extract / Inline Class.** *Extract Class* moves features to a new delegate class and adds a composite reference to the new class together with an opposite reference. During migration, an object of the delegate class is created for each object of the original class, values for the moved features are moved to the new delegate object, and a link to the delegate object is created. The corresponding *Inline Class* removes a delegate class and adds its features to the referring class. There must be no other references to the delegate class. On the model level, values of objects of the delegate class are moved to objects of the referring class. Objects of the delegate class and links to them are deleted. The operations become a pair of constructor and destructor, if the composite reference has no opposite.

**Table 5.5:** Delegation operations

| | | Classific. | | | Modelw. | | | OO dataware | | | | APIw. | | BMW | | Evaluation | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| # | Operation Name | Language preservation | Model preservation | Inverse | [Wachsmuth, 2007] | [Becker et al., 2007] | [Cicchetti et al., 2008] | [Banerjee et al., 1987] | [Brèche, 1996] | [Pons, 1997] | [Claypool et al., 2000] | [Fowler, 1999] | [Dig and Johnson, 2006] | FLUID | TAF-Gen | PCM | GMF | Unicase | Quamoco | TTC |
| 1 | Extract Class | r | s | 2s | • | | • | | • | • | | • | • | • | • | | • | | • | |
| 2 | Inline Class | r | s | 1s | • | | • | | | • | | • | • | | • | | | | • | |
| 3 | Fold Class | r | s | 4s | | | | | | | | | | • | • | • | | | | |
| 4 | Unfold Class | r | s | 3s | | | | | | | | | | | | | | | | • |
| 5 | Move Feature over Reference | c | s | 6s | • | | • | | | • | | • | • | | • | | • | | | |
| 6 | Collect Feature over Reference | d | u | 5u | | | | | | | | | | | | | • | | • | |

**Fold / Unfold Class.** *Fold* and *Unfold Class* are quite similar to *Extract* and *Inline Class*. The only difference is, that the delegate class exists already and thus is not created or deleted.

**Example 5.7** (Extract / Inline / Fold / Unfold Class)**.** *The example in Figure 5.6 illustrates the difference between the operations. From left to right, the features* a1 *and* r1 *of class* A *are folded to a composite reference* c *to class* C *which has exactly these two features. In contrast, the features* a2 *and* r2 *of class* A *are extracted into a new delegate class* D*. From right to left, the composite reference* c *is unfolded which keeps* C *intact while* d *is inlined which removes* D*.*

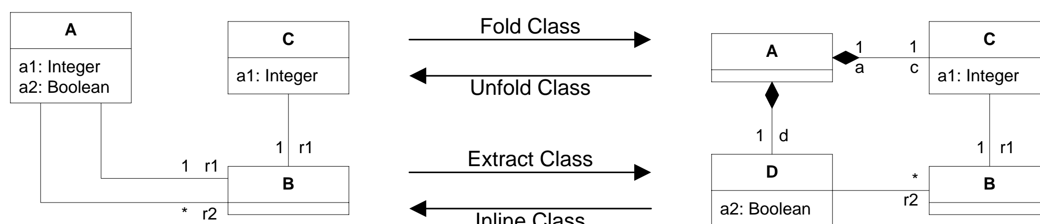



**Figure 5.6:** Operations *Extract / Inline Class* and *Fold / Unfold Class*

**Move / Collect Feature over Reference.** *Move Feature over Reference* is a constructor which moves a feature over a single-valued reference to a target class. Slots of the original feature must be moved over links to objects of the target class. For objects of the target class which are not linked to an object of the source class, default values must be added. The destructor *Collect Feature over Reference* is a safe inverse of the last operation. It moves a feature backwards over a reference. The multiplicity of the feature might be altered during the move depending on the multiplicity of the reference. For optional and/or multi-valued references, the feature becomes optional respectively multi-valued, too. Slots of the feature must be moved over links from objects of the source class. If an object of the source class is not linked from objects of the target class, values of the original feature are removed.

**Example 5.8** (Move / Collect Feature over Reference)**.** *Figure 5.7 depicts an example for*

*both operations. From left to right, the feature* f1 *is moved along the reference* r1 *to class* B*. Furthermore, the feature* f2 *is collected over the reference* r2 *and ends up in class* A*. Since* r2 *is optional and multi-valued,* f2 *becomes optional and multi-valued, too. From right to left, the feature* f1 *is collected over the reference* r1*. Its multiplicity stays unaltered. Note that there is no single operation for moving* f2 *to class* C *which makes* Collect Feature over Reference *in general uninvertible. For the special case of a single-valued reference,* Move Feature over Reference *is an unsafe inverse.*

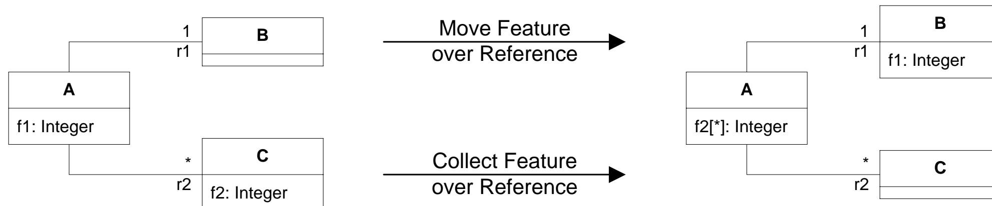



**Figure 5.7:** Operations *Move / Collect Feature over Reference*

### 5.2.8 Replacement Operations

Replacement operations replace one metamodeling construct by another, equivalent construct. Thus replacement operations typically are refactorings and safely model-migrating. With the exception of the last two operations, an operation to replace the first construct by a second always comes with a safe inverse to replace the second by the first, and vice versa.

**Table 5.6:** Replacement operations

| # | Operation Name | Classific. | | | Modelw. | | | OO dataware | | | | APIw. | | BMW | | Evaluation | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Language preservation | Model preservation | Inverse | [Wachsmuth, 2007] | [Becker et al., 2007] | [Cicchetti et al., 2008] | [Banerjee et al., 1987] | [Brèche, 1996] | [Pons, 1997] | [Claypool et al., 2000] | [Fowler, 1999] | [Dig and Johnson, 2006] | FLUID | TAF-Gen | PCM | GMF | Unicase | Quamoco | TTC |
| 1 | Subclasses to Enumeration | r | s | 2s | | | | | | | | • | | | | | | | | |
| 2 | Enumeration to Subclasses | r | s | 1s | | | | | | | | • | | | | | | • | | • |
| 3 | Reference to Class | r | s | 4s | • | | | | | | | | | | • | | | | • | |
| 4 | Class to Reference | r | s | 3s | • | | | | | | | | | • | | | | | | |
| 5 | Inheritance to Delegation | r | s | 6s | | | | | | • | | • | | • | • | | • | | | |
| 6 | Delegation to Inheritance | r | s | 5s | | | | | | • | | • | | | | | | | | |
| 7 | Reference to Identifier | c | s | 8s | | | | | | | | | | • | | | | | | |
| 8 | Identifier to Reference | d | u | 7u | | | | | | | | | | • | | | • | | • | |

**Subclasses vs. Enumerations.** To be more flexible, empty subclasses of a class can be replaced by an attribute which has an enumeration as type, and vice versa. *Subclasses to Enumeration* deletes all subclasses of the class and creates the attribute in the class as well as the enumeration with a literal for each subclass. In a model, objects of a certain subclass are migrated to the superclass, setting the attribute to

the corresponding literal. Thus, the class is required to be non-abstract and to have only empty subclasses without further subclasses. *Enumeration to Subclasses* does the inverse and replaces an enumeration attribute of a class by subclasses for each literal.

**Example 5.9** (Subclasses to Enumeration / Enumeration to Subclasses)**.** *The example in Figure 5.8 demonstrates both directions. From left to right,* Subclasses to Enumeration *replaces the subclasses* S1 *and* S2 *of class* C *by the new attribute* e *which has the enumeration* E *with literals* S1 *and* S2 *as type. In a model, objects of a subclass* S1 *are migrated to class* C*, setting the attribute* e *to the appropriate literal* S1*. From right to left,* Enumeration to Subclasses *introduces a subclass to* C *for each literal of* E*. Next, it deletes the attribute* e *as well as the enumeration* E*. In a model, objects of class* C *are migrated to a subclass according to the value of attribute* e*.*

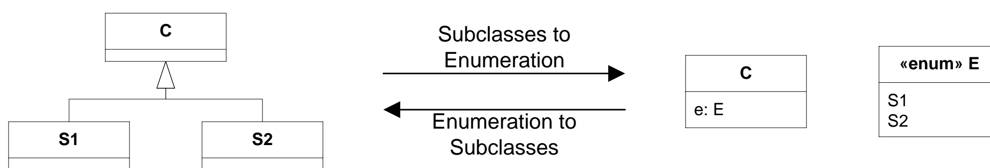



**Figure 5.8:** Operations *Subclasses to Enumeration / Enumeration to Subclasses*

**References vs. Classes.** To be able to extend a reference with features, it can be replaced by a class, and vice versa. *Reference to Class* makes the reference composite and creates the reference class as its new type. Single-valued references are created in the reference class to target the source and target class of the original reference. In a model, links conforming to the reference are replaced by objects of the reference class, setting source and target reference appropriately. *Class to Reference* does the inverse and replaces the class by a reference. To not lose expressiveness, the reference class is required to define no features other than the source and target references.

**Example 5.10** (Reference to Class / Class to Reference)**.** *The example in Figure 5.9 demonstrates both directions. From left to right,* Reference to Class *retargets the reference* r *to a new reference class* R*. Source and target of the original reference can be accessed via references* s *and* t*. In a model, links conforming to the reference* r *are replaced by objects of the reference class* R*. From right to left,* Class to Reference *removes the reference class* R *and retargets the reference* r *directly to the target class* T*.*

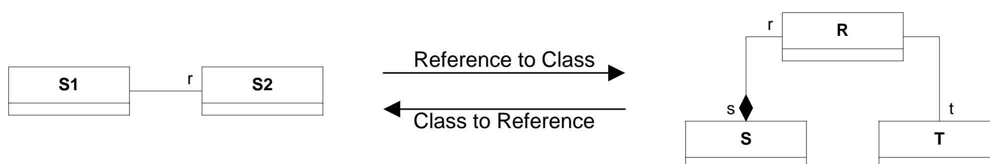



**Figure 5.9:** Operations *Reference to Class / Class to Reference*

**Inheritance vs. Delegation.** Inheriting features from a superclass can be replaced by delegating them to the superclass, and vice versa. *Inheritance to Delegation* removes the inheritance relationship to the superclass and creates a composite, mandatory

single-valued reference to the superclass. In a model, the values of the features inherited from the superclass are extracted to a separate object of the superclass. By removing the super type relationship, links of references to the superclass are no longer allowed to target the original object, and thus have to be retargeted to the extracted object. *Delegation to Inheritance* does the inverse and replaces the delegation to a class by an inheritance link to that class.

**Example 5.11** (Inheritance to Delegation / Delegation to Inheritance)**.** *The example in Figure 5.10 demonstrates both directions. From left to right,* Inheritance to Delegation *replaces the inheritance link of class* C *to its superclass* S *by a composite, single-valued reference from* C *to* S*. In a model, the values of the features inherited from the superclass* S *are extracted to a separate object of the superclass. From right to left,* Delegation to Inheritance *removes the reference* s *and makes* S *a superclass of* C*.*

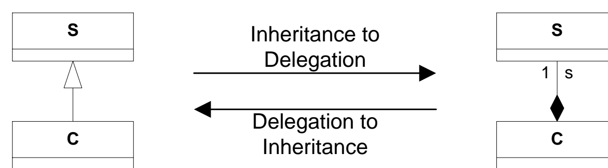



**Figure 5.10:** Operations *Inheritance to Delegation / Delegation to Inheritance*

**References vs. Identifiers.** To decouple a reference, it can be replaced by an indirect reference via identifier, and vice versa. *Reference to Identifier* deletes the reference and creates an attribute in the source class whose value refers to an identifier attribute in the target class. In a model, links of the reference are replaced by setting the attribute in the source object to the identifier of the target object. *Identifier to Reference* does the inverse and replaces an indirect reference via identifier by a direct reference. Our metamodeling formalism does not provide a means to ensure that there is a target object for each identifier used by a source object. Consequently, *Reference to Identifier* is a constructor and *Identifier to Reference* a destructor, thus being an exception in the group of replacement operations.

### 5.2.9 Merge / Split Operations

Merge operations merge several metamodel elements of the same type into a single element, whereas split operations split a metamodel element into several elements of the same type. Consequently, merge operations typically are destructors and split operations constructors. In general, each merge operation has an inverse split operation. Split operations are more difficult to define, as they may require metamodel-specific information about how to split values. There are different merge and split operations for the different metamodeling constructs.

**Features.** *Merge Features* merges a number of features defined in the same class into a single feature. In the metamodel, the source features are deleted and the target feature is required to be general enough—through its type and multiplicity—so that the values of the other features can be fully moved to it in a model. Depending on the type of feature that is merged, a repeated application of *Create Attribute* or

**Table 5.7:** Merge / Split operations

| # | Operation Name | Classific. | | | Modelw. | | | OO dataware | | | | APIw. | | BMW | | Evaluation | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Language preservation | Model preservation | Inverse | [Wachsmuth, 2007] | [Becker et al., 2007] | [Cicchetti et al., 2008] | [Banerjee et al., 1987] | [Brèche, 1996] | [Pons, 1997] | [Claypool et al., 2000] | [Fowler, 1999] | [Dig and Johnson, 2006] | FLUID | TAF-Gen | PCM | GMF | Unicase | Quamoco | TTC |
| 1 | Merge Features | d | u | | | | | | | • | | | | • | | | | | • | |
| 2 | Split Reference by Type | r | s | 1s | | | | | | | | | | | | | | | • | • |
| 3 | Merge Classes | d | u | 4u | | | | | • | • | | | | • | | • | • | | • | |
| 4 | Split Class | c | p | 3s | | | | | | | | | | | | | | | | • |
| 5 | Merge Enumerations | d | u | | | | | | | | | | | | | | • | | | |

*Create Reference* provides an unsafe inverse. *Split Reference by Type* splits a reference into references for each subclass of the type of the original reference. In a model, each link of the reference is moved to the corresponding target reference according to its type. If we require that the type of the reference is abstract, this operation is a refactoring and has *Merge Features* as a safe inverse.

**Classes.** *Merge Classes* merges a number of sibling classes—i.e. classes sharing a common superclass—into a single class. In the metamodel, the sibling classes are deleted and their features are merged to the features of the target class according to name equality. Each of the sibling classes is required to define the same features so that this operation is a destructor. In a model, objects of the sibling classes are migrated to the new class. *Split Class* is a safe inverse and splits a class into a number of classes. A function that maps each object of the source class to one of the target classes needs to be provided to the migration.

**Enumerations.** *Merge Enumerations* merges a number of enumerations into a single enumeration. In the metamodel, the source enumerations are deleted, and their literals are merged to the literals of the target enumeration according to name equality. Each of the source enumerations is required to define the same literals so that this operation is a destructor. Additionally, attributes that have the source enumerations as type have to be retargeted to the target enumeration. In a model, the values of these attributes have to be migrated according to how literals are merged. A repeated application of *Create Enumeration* provides a safe inverse.

### 5.2.10 Discussion

We discuss the completeness of the reusable coupled operations provided by the library as well as the dependency of the operations on the used metamodeling formalism.

**Completeness.** The reusable coupled operations in the library may need to be complete on both the metamodel and model level.

*Metamodel adaptation.* On the metamodel level, a library of operations is complete, if any source metamodel can be evolved to any target metamodel. This kind of completeness is achieved by the library presented in this section. An extreme strategy would be the following [Banerjee et al., 1987]: In a first step, the original metamodel needs to be discarded. Therefore, we delete opposite references and features. Next, we delete data types and enumerations and collapse inheritance hierarchies by inlining subclasses. We can now delete the remaining classes. Finally, we delete packages. In a second step, the target metamodel is constructed from scratch by creating packages, enumerations, literals, data types, classes, attributes, and references. Inheritance hierarchies are constructed by extracting empty subclasses.

*Model migration.* Completeness is much harder to achieve, when we take the model level into account. Here, a library of operations is complete, if any model migration corresponding to an evolution from a source metamodel to a target model can be expressed. In this sense, a complete library needs to provide a full-fledged model transformation language based on operations. A first useful requirement is Turing completeness. But reaching for this kind of completeness comes at the price of usability. Given an existing operation, one can always think of a different operation having the same effect on the metamodel level but a slightly different migration. But the higher the number of coupled operations, the more difficult it is to find an appropriate operation in the library. And with many similar operations, it is hard to decide which one to apply. We therefore do not target theoretical completeness—to capture all possible migrations—but rather practical completeness—to capture migrations that likely happen in practice. Theoretical completeness can still be achieved by providing a means for defining custom coupled operations. This way, the user can manually specify a custom model migration for a metamodel adaptation specified as a sequence of primitive operations.

**Metamodeling Formalism.** In this section, we focus only on core metamodeling constructs that are interesting for coupled evolution of metamodels and models.

*E-MOF.* A metamodel defines not only the abstract syntax of a modeling language, but also an API to access models expressed in this language. For this purpose, concrete metamodeling formalisms like Ecore or MOF provide additional metamodeling constructs like interfaces, operations, derived features, volatile features, or annotations. A library will need additional operations addressing these metamodeling constructs in order to reach full compatibility with Ecore or MOF. These additional operations are relevant for practical completeness. In the GMF case study explained in Section 7.2 (*Graphical Modeling Framework*), we found 25% of the applied operations to address changes in the API. We have not included these operations into the library, since most of them do not require migration. The only exceptions were annotations containing constraints. An operation catalog accounting for constraints needs to deal with two kinds of migrations: First, the constraints need migration when the metamodel evolves. Operations need to provide this migration in addition to model migration. Second, evolving constraints might invalidate existing models and thus require model migration. Here, new coupled operations for the evolution of constraints are needed.

*C-MOF.* Things become more complicated when it comes to Complete MOF (C-MOF) [Object Management Group, 2006a]. Concepts like package merge, feature subset-

ting, and visibility affect the semantics of existing operations in the library. To deal with these concepts, additional operations are needed, and existing operations need to be refined. For example, we would need four different kinds of *Rename* due to the package merge: 1) Renaming an element which is involved in a merge neither before nor after the renaming (*Rename Element*). 2) Renaming an element which is not involved in a merge in order to include it into a merge (*Include by Name*). 3) Renaming an element which is involved in a merge in order to exclude it from the merge (*Exclude by Name*). 4) Renaming all elements which are merged to the same element (*Rename Merged Element*).

## 5.3 Language to Specify the Coupled Evolution

In this section, we present in more detail COPE's language to specify the coupled evolution of metamodels and models. COPE implements the concept of coupled operations and is based on the Eclipse Modeling Framework (EMF) [Steinberg et al., 2009]. In order to achieve in-place transformation, COPE softens the conformance of a model to its corresponding metamodel during coupled evolution. Based on this decoupling of metamodel and model, COPE provides expressive primitives for both metamodel adaptation and model migration. These primitives can be combined to encode custom and reusable coupled operations. To simplify encoding, metamodel and model conformance are only required at operation boundaries.

### 5.3.1 Decoupling Metamodel and Model

Figure 5.11 depicts the relationship between a model and its metamodel during coupled evolution. Inside operation boundaries, model and metamodel can be modified independently of each other, whereas conformance is required at operation boundaries. As a consequence, we are able to perform in-place transformation, i.e. direct updates of the models. In-place transformation is more efficient than out-of-place transformation, which requires to rebuild the migrated model from scratch.

**Metamodel.** Since COPE is implemented based on the Eclipse Modeling Framework (EMF) [Steinberg et al., 2009], we use Ecore as a metametamodel. Ecore is a Java-based implementation of the metamodeling language presented in Section 2.2 (*Metamodeling – Modeling the Abstract Syntax of Modeling Languages*). To distinguish them from Java classes, all classes defined by Ecore start with a prefix E. Other than that, there are only small naming differences to our simplified metametamodel: an EClassifier is a Type, an EStructuralFeature is a Feature, and containment references are composite references. However, in principle, our approach is not restricted to Ecore, but can be transferred to any object-oriented metamodeling language.

**Model.** Models are expressed in a generic instance structure that is independent of the specific metamodel. As a consequence, this generic instance structure can be used to migrate all possible models, independently of the metamodel. This is consistent with the formalization of coupled operations in Definition 5.1 which uses the set of all models as domain and codomain of the model migration. For COPE, we had to

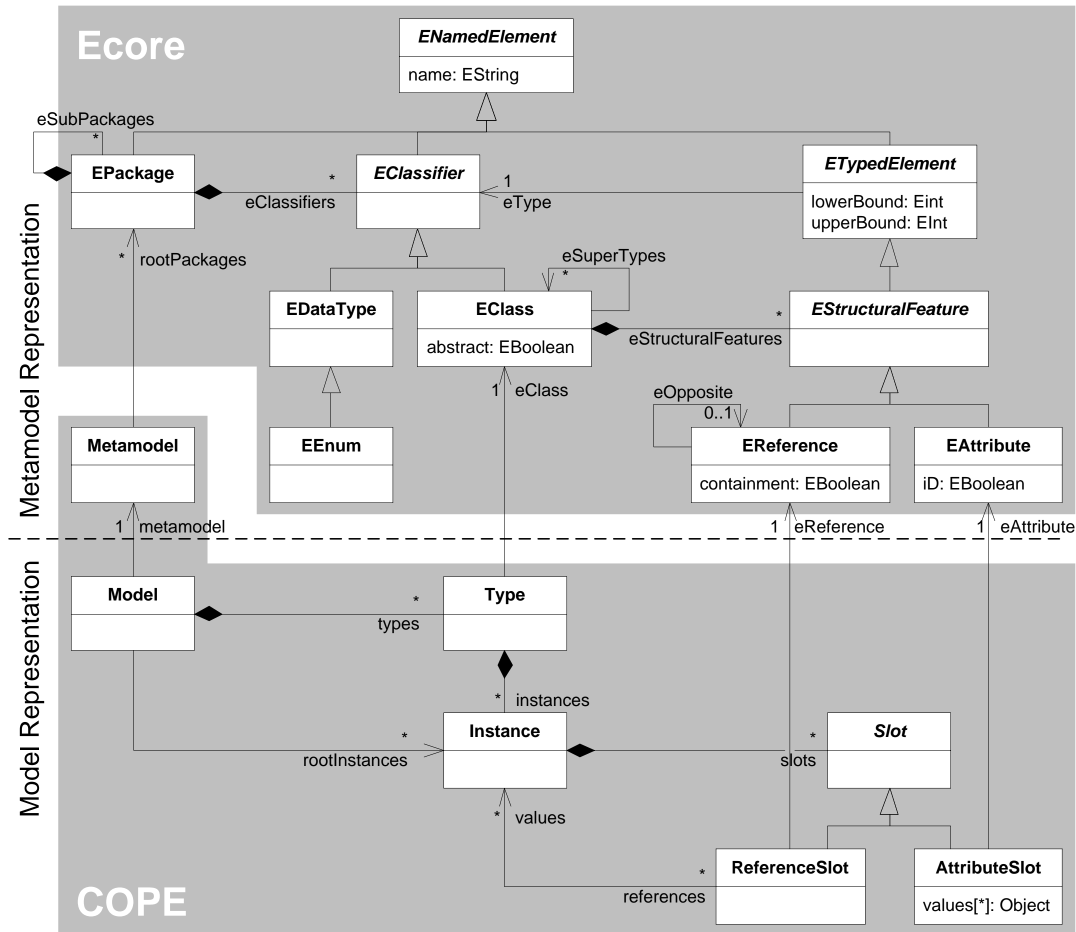



**Figure 5.11:** Generic instance model

implement our own model representation, since EMF does not allow applications to change the metamodel, after the model is loaded.

A Model consists of a number of types each of which is a container for instances of a certain type. A Type thus provides efficient access to the instances of a certain class. A model has a number of rootInstances, i.e. instances which are not referenced from a parent via a containment reference. Each Instance has a number of slots (Slot) which are the valuations of either attributes (AttributeSlot) or references (ReferenceSlot). Valuations of attributes are maintained as Java Objects, whereas valuations of references are maintained as links to the corresponding instances. The reference values from ReferenceSlot to Instance is bidirectional to efficiently access the references to an instance of a certain reference. Instances and slots are associated to their corresponding metamodel elements (eClass, eReference and eAttribute).

However, these associations do not constrain an instance to always conform to its type in the metamodel. This loose association allows us to first modify the metamodel without affecting the model, and then migrate the model to the evolved metamodel. Since this decoupling can lead to states where the model does not conform to its metamodel, conformance is checked at operation boundaries. If the model does not conform to the metamodel at operation boundaries, COPE throws an exception,

notifying the language engineer about a problem.

**Metamodel Conformance.** The metamodel conforms to the Ecore metametamodel if it fulfills the constraints defined by the metametamodel. These constraints are similar to the ones mentioned in Section 2.2.4 (*Complete E-MOF Metametamodel*). Examples for constraints are that no two classes may have identical names, or that a class may neither directly nor transitively be a supertype of itself. Due to space constraints, we refer the reader to the Java documentation of the EMF source code for a complete list of the constraints[2]. EMF provides a facility to easily check for the violation of these constraints which we employ in the implementation of COPE.

**Model Conformance.** The loose association between metamodel and model may lead to states where the model does not conform to its metamodel. However, we can define what it means for a model to conform to its metamodel based on the association between metamodel and model elements.

- A Model conforms to its metamodel if each instance conforms to its type which has to be a non-abstract class defined by the model's metamodel.
- An Instance conforms to its type if
  - each slot conforms to its feature which is defined by the instance's type or its super types,
  - for each mandatory feature defined by the instance's type there is a corresponding slot,
  - either it is a root element of the model or it is referenced exactly once by a containment reference slot of another instance, and
  - it fulfills all further constraints which are defined in the context of the instance's type or its super types.
- An AttributeSlot conforms to its attribute if the value of the slot is consistent with the attribute's type and multiplicity.
- A ReferenceSlot conforms to its reference if
  - the value of the slot is consistent with the attribute's type and multiplicity, and
  - the instance belongs to the opposite reference slot of each value.

While on purpose not enforced by the loose association between metamodel and model, these constraints can be checked at operation boundaries.

### 5.3.2 Breaking Metamodel Changes Revisited

In Section 2.5.3 (*Breaking Metamodel Changes*), we listed the metamodel changes that are breaking existing models for the metamodeling language presented in Section 2.2 (*Metamodeling – Modeling the Abstract Syntax of Modeling Languages*). In this sec-

[2] The source distribution of EMF can be downloaded from `http://www.eclipse.org/modeling/emf/`.

tion, we revisit the breaking metamodel changes with respect to the generic instance model presented in the last subsection. There are changes which have been breaking previously that are no longer breaking due to certain techniques used in the generic instance model. Table 5.8 gives an overview over the breaking (B) and non-breaking (NB) metamodel changes for the generic instance model.

**Table 5.8:** Breaking and non-breaking metamodel changes revisited

| Class | Feature | Change | NB | B | Condition |
|---|---|---|---|---|---|
| Metamodel | packages | add | • | | |
| | | move | • | | |
| | | remove | • | | package is empty |
| | | | | • | package is not empty |
| Package | subPackages | add | • | | |
| | | move | • | | |
| | | remove | • | | package is empty |
| | | | | • | package is not empty |
| | types | add | • | | |
| | | move | • | | |
| | | remove | • | | type is a primitive type |
| | | | | • | type is a class |
| Type | name | modify | • | | type is a class |
| | | | | • | type is a primitive type |
| Enumeration | literals | add | • | | |
| | | remove | • | | enumeration is not used |
| | | | | • | enumeration is used |
| Class | abstract | modify | • | | new value is false |
| | | | | • | new value is true |
| | features | add | • | | new feature is optional |
| | | | | • | new feature is mandatory |
| | | remove | | • | |
| | superTypes | add | • | | new superclass has no mandatory feature |
| | | | | • | new super class has mandatory features |
| | | remove | • | | superclass defines no features |
| | | | | • | superclass defines features |
| Feature | lowerBound | modify | • | | lower bound is decreased |
| | | | | • | lower bound is increased |
| | upperBound | modify | • | | upper bound is increased |
| | | | | • | upper bound is decreased |
| Attribute | id | modify | • | | new value is false |
| | | | | • | new value is true |
| | type | modify | | • | |
| Reference | composite | modify | | • | |
| | type | modify | • | | new value is super class of old value |
| | | | | • | new value is not super class of old value |
| | opposite | modify | • | | new value is null |
| | | | | • | new value is not null |

In the metamodeling language presented in Section 2.2 (*Metamodeling – Modeling the Abstract Syntax of Modeling Languages*), objects and slots refer to their metamodel elements by name. As a consequence, renaming classes and features are breaking changes. However, in the generic instance model, objects and slots directly refer to their metamodel elements. As a consequence, renaming classes and features are non-breaking changes due the generic instance model. Since objects refer to their class by fully qualified name—taking the package hierarchy into account, the generic in-

stance model is also non-breaking for renaming packages as well as moving classes and packages.

### 5.3.3 Primitives for Metamodel Adaptation and Model Migration

To preserve conformance of a model to its metamodel, a model migration needs to be specified for breaking metamodel changes. COPE provides both expressive primitives to specify metamodel adaptation and to specify model migration. The primitives are complete in the sense that every possible metamodel adaptation and every possible model migration can be encoded.

**Metamodel Adaptation.** For metamodel adaptation, COPE provides the following primitives to query the metamodel:

- `«qualifiedName»` to access a metamodel element by means of its qualified name, i.e. a package, class or feature.
- `«element».«featureName»` to access the value of a feature of a metamodel element as defined by the metametamodel.
- `«element».getInverse(«reference»)` to access the metamodel elements which refer to a metamodel element through a reference as defined by the metametamodel.

COPE provides the following primitives to modify the metamodel that perform an in-place transformation:

- `«package».newEClass(...)`, `«class».newEAttribute(...)` or `«class».newEReference(...)` to create a new class, attribute or reference. There are primitives to create instances of each class defined by the metametamodel.
- `«element».delete()` to delete a metamodel element.
- `«element».«featureName» = «value»` to modify the value of a feature of a metamodel element.
- `«element».«featureName».add(«value»)` and `«element».«featureName».remove(«value»)` to modify the value of a multi-valued feature of a metamodel element. When the feature is a containment reference, an addition implies a removal from the previous parent metamodel elements.

These primitives are complete to describe every possible evolution from a source metamodel to a target metamodel. This can be easily shown by first completely deleting the source metamodel and then creating the target metamodel from scratch, as proved in [Banerjee et al., 1987].

**Model Migration.** For model migration, COPE provides the following primitives to query a model:

- `«class».instances` to access all instances of a class.
- `«class».allInstances` to access all instances of a class or any of its subclasses.

- `«instance».instanceOf(«class»)` to check whether the type of the instance is the class or any of its subclasses.
- `«instance».get(«feature»)` or `«instance».«featureName»` to access the value of a feature of an instance. The short form can be used if the feature with that name is available in the instance's type.
- `«instance».getInverse(«reference»)` to access the instances which refer to an instance by a reference as defined by the metamodel.

COPE provides the following primitives to modify the model that perform an in-place transformation:

- `«class».newInstance()` to create a new instance of a class.
- `«instance».delete()` to delete an instance from the model.
- `«instance».migrate(«class»)` to change the type of an instance to a different class.
- `«instance».set(«feature», «value»)` or `«instance».«featureName» = «value»` to modify the value of a feature of an instance. The short form can be used if the feature with that name is available in the instance's type.
- `«instance».add(«feature», «value»)` or `«instance».«featureName».remove(«value»)` to modify the value of a multi-valued feature of an instance.
- `«instance».unset(«feature»)` to unset and return the value of a feature of an instance.

These primitives are constructed in a way that they also provide access to model information which currently does not conform to the metamodel. They are expressive enough to describe every possible evolution from a source model to a target model, using the same explanation as on the metamodel-level. However, rather than describing a single model evolution, a model migration language needs to describe the migration of all models conforming to a source metamodel. As a consequence, the primitives alone are not expressive enough to describe any possible model migration.

### 5.3.4 Implementing Coupled Operations

To describe any possible model migration, the primitives can be invoked from within the general-purpose scripting language Groovy [Koenig et al., 2007]. We have decided in favor of Groovy due to the following reasons:

- Groovy is Turing-complete and thus expressive enough to specify even complex model migrations.
- Groovy is a dynamically typed language and thus allows language engineers to encode shorter migration code by not requiring type declarations. A statically typed language does not bring an advantage anyway, since the in-place transformation cannot be easily expressed in a type-safe manner.
- Groovy is similar to Java and thus easy to learn for language engineers using EMF, which is implemented in Java.
- Groovy allows language engineers to compactly specify expressions, similar to

OCL, that are required to navigate the metamodel and model.

The metamodel adaptation and model migration of a coupled operation are specified by the application of the metamodel and model change primitives from within Groovy code. The interpreter of COPE ensures that a coupled operation can only be successfully completed, in case it preserves metamodel and model conformance. For the metamodel adaptation, preservation of metamodel conformance can already be checked independently of a specific model. For the model migration, preservation of model conformance can only be checked, when the coupled operation is applied to a specific model.

**Custom Coupled Operation.** A custom coupled operation is a coupled operation that is specific to a certain metamodel. As a consequence, the custom coupled operation directly refers to the metamodel that it adapts and, as such, is only applicable to that metamodel. A custom coupled operation is required if the model migration cannot be specified by a reusable coupled operation or a sequence of reusable coupled operations. A custom coupled operation is specified manually by the language engineer as a script that uses the primitives to specify both metamodel adaptation and model migration.

Since a custom coupled operation cannot be expressed by reusable coupled operations, it tends to encapsulate more complex migrations. To master custom coupled operations, the following guidelines need to be considered when implementing a custom coupled operation:

- If a custom coupled operation is rather complex, a language engineer should try to decompose it into smaller coupled operations. Smaller coupled operations are easier to implement and maintain. Maybe some of the coupled operations that are composed to a complex custom coupled operation can be even covered by reusable coupled operations.
- If a complex custom coupled operation cannot be decomposed into smaller coupled operations, the model migration can be modularized by defining helper functions in Groovy:

```
1   helperFunction = { «parameter» ->
2     ...
3     return «expression»
4   }
```

- If a class is deleted by the metamodel adaptation, the model migration needs to remove all the instances of the class. Usually, these instances are migrated to another class which may be created by the metamodel adaptation:

```
1   for(instance in «class».instances) {
2     instance.migrate(«anotherClass»)
3   }
```

  If the information defined by these instances is no longer required, they can also be deleted.
- If a feature is deleted from a class by the metamodel adaptation, the model migration needs to unset its values from the instances of the class:

```
1  for(instance in «class».allInstances) {
2    def value = instance.unset(«feature»)
3    ...
4  }
```

If the information defined by these values should be preserved, they need to be transferred to other features. Usually these other features have been created by the metamodel adaptation.

- If a feature is moved from one class to another, the model migration needs to unset its values from the instances of the source class. To preserve information, the values are usually transferred to instances of the target class.
- Changing the containment structure of a model usually requires complex migrations. More specifically, changing the containment of a reference by the metamodel adaptation requires careful migration of the instances of the reference's type. The language engineer needs to ensure that each instance is contained only once—either by another instance or directly by the model—after the migration. If a reference is made containment, its values have to be removed from other containers. Likewise, if the containment of a reference is dropped, the values have to be added to other containers.

**Reusable Coupled Operation.** We use the reuse mechanism of functions of the host language in order to declare reusable coupled operations. Reusable coupled operations can be instantiated by simply invoking the corresponding function. To register them through the library mechanism of COPE, reusable coupled operations need to follow a certain structure. Figure 5.12 illustrates the abstract syntax of this structure as a UML class diagram, and Listing 5.4 illustrates its concrete syntax.

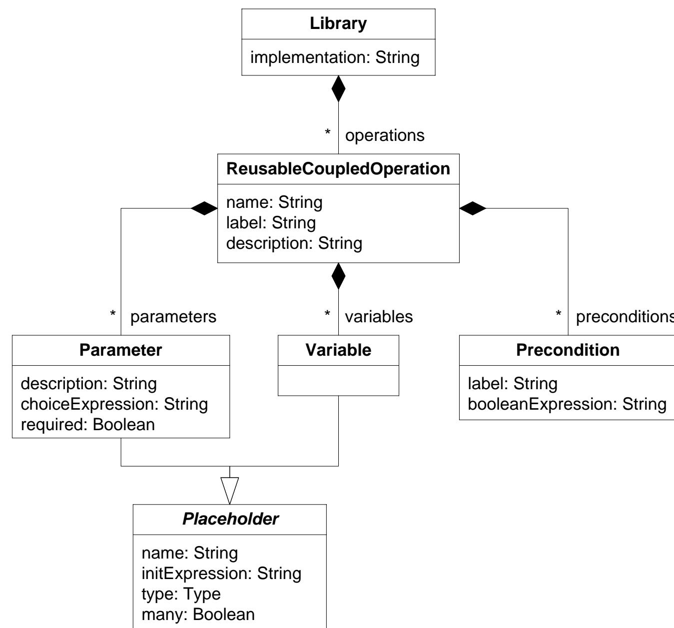



**Figure 5.12:** Signature of reusable coupled operations (abstract syntax)

**Listing 5.4:** Signature of reusable coupled operations (concrete syntax)

```
// reusable coupled operation
@label("«label»")
@description("«description»")
«name» = {
  // parameters
  @description("«description»") «type» «name» = «initExpression»,
  ... ->

  // variables
  «type» «name» = «initExpression»
  ...

  // preconditions
  assert «booleanExpression» : «label»
  ...

  // implementation
  ...
}
```

In the following, we describe the different constituents of the signature of a reusable coupled operation:

- *Library:* A set of reusable coupled operations are organized into a Library. A library provides the implementation of its operations as a single Groovy script. There may be multiple libraries; e.g. we can create a library for each group of reusable coupled operations mentioned in Section 5.2 (*Library of Reusable Coupled Operations*).
- *Reusable coupled operation:* Like a Groovy function, a ReusableCoupledOperation is identified by a unique name. Moreover, a reusable coupled operation has a label and a description so that the language engineer can understand the effect of the operation on metamodels and models. In the concrete syntax, label and description are declared by special annotations.
- *Parameter:* Parameters make the reusable coupled operation independent of the specific metamodel. Like a Groovy parameter, a parameter is identified by a name and has a type. COPE only supports types defined by the metametamodel as well as sequences of these types (many). In the concrete syntax, the type is identified by its name and, in case of sequences, by `List<«name»>`. The values of a parameter can not only be restricted by its type, but also by a choice expression which is derived from the preconditions as explained below. A parameter can have an init expression to initialize its value based on the value of previously defined parameters of the same operation. If the init expression is **`null`**, then the parameter is not required to be set.
- *Variable:* A Variable is similar to a Parameter in terms of name and type. In contrast to a parameter, it cannot be set by a language engineer that invokes the reusable coupled operation, but is derived from the values of parameters and previously defined variables. Thereby, it does not require neither description

nor choice expression, and the init expression is mandatory. A variable factors out recurring expressions from the preconditions. In the concrete syntax, variables are defined as Groovy variables right after the parameters, but before the preconditions.

- *Precondition:* A Precondition restricts the applicability of the reusable coupled operation. A precondition is defined by a boolean expression and a label that helps the language engineer to resolve a violation of the precondition. In the concrete syntax, preconditions are defined as Groovy assertions that have the label as message. A precondition defines a choice expression for a parameter if it is of the following form:
  - `«choiceExpression».contains(«parameter»)` for single-valued parameters, and
  - `«choiceExpression».containsAll(«parameter»)` for multi-valued parameters.

## 5.4 Limitations of Automating Model Migration

The evolution of modeling languages occasionally leads to metamodel changes for which the migration of models inherently cannot be fully automated. In these cases, the migration of models requires information which is not available in the model. However, manually migrating a potentially unknown number of models imposes a heavy burden on the language users. Consequently, the language engineers are tempted to avoid these model-specific coupled changes by adapting the metamodel in a way that the model migration does not require information from the users. However, not being able to implement such changes limits modeling language evolution, and threatens the simplicity and quality of the metamodel. In this section, we formally characterize metamodel adaptations in terms of the effort needed for model migration. We focus on the problem of metamodel changes that prevent the automatic migration of models when their semantics needs to be preserved. Therefore, in order to be able to characterize these changes, we need to take the semantics of the evolving modeling language into account. We outline different possibilities to systematically cope with these kinds of metamodel changes.

### 5.4.1 Considering Semantics of Modeling Languages

In Section 2.5.4 (*Model Migration*), we defined preservation properties that a model migration needs to fulfill for the evolution of a modeling language. Of course, these preservation properties can be transferred to coupled operations if a coupled operation is used to implement a certain metamodel change:

**Definition 5.13** (Syntax-Preserving Coupled Operation)**.** *A coupled operation $co = (adm, mig)$ is called syntax-preserving for a metamodel $mm \in \mathcal{MM}$ if the migration transforms all models that conform to the original metamodel $mm$ to models that conform to the evolved metamodel $adm(mm) \in \mathcal{MM}$:*

$$\forall m \in \mathcal{M} : m \vDash mm \Rightarrow mig(m) \vDash adm(mm)$$

A model conforms to the metamodel if it is built from the constructs and fulfills the constraints defined by the metamodel. For the description of the relationship between model and metamodel, we refer the reader to Section 2.2.2 (*Abstract Syntax of a Modeling Language*). Note that, in practice, usually only a small subset $\mathcal{L}_{built} \subset \mathcal{L}_{mm}$ of the possible models of a metamodel $mm \in \mathcal{MM}$ are actually built.

**Definition 5.14** (Semantics-Preserving Coupled Operation)**.** *A syntax-preserving coupled operation $co = (adm, mig)$ is called semantics-preserving for a metamodel $mm \in \mathcal{MM}$ and a semantics change $S_1 \mapsto S_2$ from semantics version $S_1 : \mathcal{L}_{mm} \to \mathcal{SD}_1$ to $S_2 : \mathcal{L}_{adm(mm)} \to \mathcal{SD}_2$ if the migration preserves the meaning of all models:*

$$\forall m \in \mathcal{L}_{mm} : S_1(m) \cong S_2(mig(m))$$

Since the semantics $S_1$ and $S_2$ are total functions, each syntactically correct model has an associated meaning. Before we can examine whether we can define a semantics-preserving model migration for a metamodel adaptation, we have to explicitly define the semantics of a modeling language.

**Example 5.12** (Semantics Preservation)**.** *We specify the semantics $S : \mathcal{L}_{mm} \to \mathcal{SD}$ for the example modeling language presented in Section 3.2.1 (Running Example) that defines a dialect of I/O state machines. Figure 5.13 gives an overview over the different versions of the metamodel that defines the abstract syntax of the modeling language. We use the same convention as in Section 2.4.1 (Semantics of a Modeling Language) to map a metamodel to functions for navigating models. The instances of a class in the metamodel can be accessed through a set with its name: e.g. $CompositeState$ to access all instances of the class* ***CompositeState*** *and its subclasses. The associations in the metamodel can be followed by functions applied to instances: e.g. $state : CompositeState \to \mathcal{P}(State)$ to access the* ***state****s of a composite state. Notationally, $cs.state$ is the same as $state(cs)$. In addition to these functions, we also need the function $parent : State \to CompositeState \cup \{\bot\}$ to access the parent state of a state $s \in State$:*

$$s.parent = cs \in CompositeState :\Leftrightarrow Edge(cs, s, \textbf{state})$$

*Based on this function, we also define the function $root : \mathcal{L}_{mm} \to CompositeState \cup \{\bot\}$ to access the root state of a state machine model $m \in \mathcal{L}_{mm}$:*

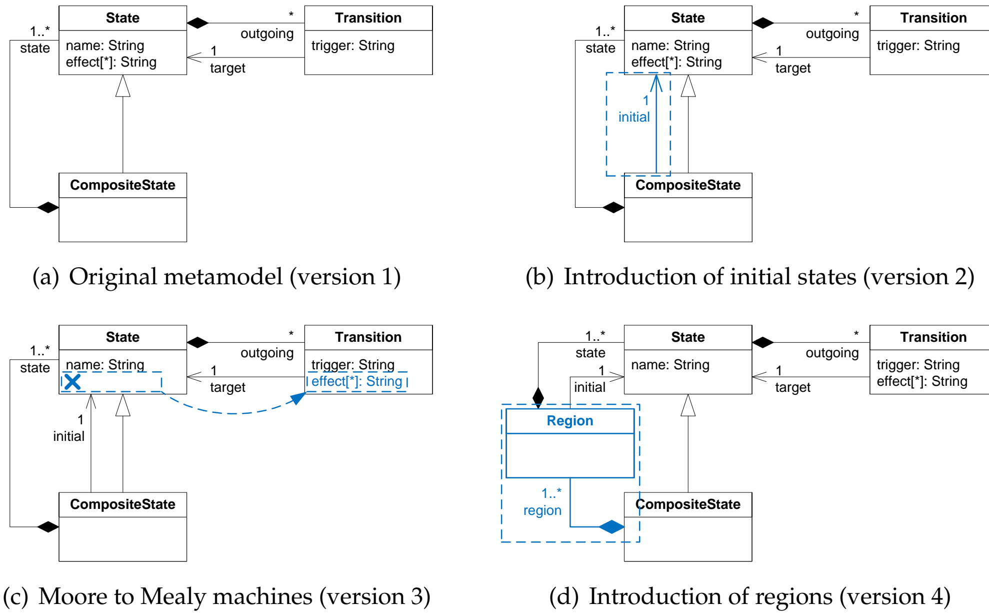
$$root(m) = cs \in CompositeState :\Leftrightarrow cs.parent = \bot$$

*We adopted the definition of the state machine semantics from [Rumpe, 1998]. The semantic domain is the behavior of I/O state machines which is defined through a function that maps a stream of input events onto sets of streams of output actions:*

$$\mathcal{SD} := \{Event^* \to \mathcal{P}(Action^*)\}$$

*Note that we can specify non-deterministic behavior using this semantic domain, as an event stream can lead to a set of action streams. The semantics $S : \mathcal{L}_{mm} \to \mathcal{SD}$ maps a state machine model $m \in \mathcal{L}_{mm}$ to its behavior $S(m) \in \mathcal{SD}$. Two state machine models $m_1, m_2 \in \mathcal{L}_{mm}$ are semantically equivalent if they produce the same set of actions for all possible event streams:*

$$m_1 \equiv m_2 :\Leftrightarrow \forall es \in Event^* : S(m_1)(es) = S(m_2)(es)$$




(a) Original metamodel (version 1)

(b) Introduction of initial states (version 2)

(c) Moore to Mealy machines (version 3)

(d) Introduction of regions (version 4)

**Figure 5.13:** Metamodel adaptation of the running example

*In the following, we define the function $T : State \rightarrow \mathcal{SD}$ in a way that it can be applied to all composite and non-composite states. The semantics of a model is defined by its root composite state $root(m) \in CompositeState$:*

$$S(m) := T(root(m))$$

*To distinguish the sets and functions from different language versions, we use the version number as index, e.g. $CompositeState_1$ to denote the composite states conforming to metamodel version 1. We use the same semantic domain for all language versions, and thus semantics preservation is given by the equivalence relation, i.e. $\cong = \equiv$.*

Original metamodel. *The semantics is defined by induction over the stream of input events. In the base case, there are no more events left to be processed for a state $s \in State_1$:*

$$T_1(s)(\langle\rangle) \quad := \quad \{\langle\rangle\} \tag{5.1}$$

*The inductive step for a state $s \in State_1$ consumes the next event $e \in Event$. Based on the transitions activated in the state, we can decide whether the event leads to a state change. The set of transitions activated by an event $e \in Event$ in a state $s \in State_1$ is the set of outgoing transitions having the event as trigger:*

$$activated_1(s, e) \quad := \quad \bigcup_{p \in s.parent_1^*} \{t \in p.outgoing_1 \mid t.trigger_1 = e\} \tag{5.2}$$

*where $^*$ is the transitive closure. Due to hierarchical states, also the outgoing transitions of all parent states have to be taken into account.*

*In case there is at least one transition activated by event $e \in Event$—i.e. $activated_1(s, e) \neq \emptyset$—the behavior of a state $s \in State_1$ is defined as the union of the behavior of the states to which the control can transition:*

$$T_1(s)(\langle e\rangle \circ es) \quad := \quad \bigcup_{t \in activated_1(s,e)} t.target_1.effect_1 \circ T_1(t.target_1)(es) \tag{5.3}$$

*The operator $\circ$ concatenates streams and is lifted to sets of streams if required. Otherwise—in case there is no transition activated—the state machine remains in the same state:*

$$T_1(s)(\langle e\rangle \circ es) \quad := \quad T_1(s)(es) \tag{5.4}$$

*Finally, the behavior of a composite state $cs \in CompositeState_1$ for an event stream $es = \langle e_1, e_2, \ldots\rangle \in Event^*$ is defined as the union of the behavior of its substates:*

$$T_1(cs)(es) \quad := \bigcup_{s \in cs.state_1} s.effect_1 \circ T_1(s)(es) \tag{5.5}$$

*Note that the formulas are polymorphic, i.e. $T_1(t.target_1)(es)$ in formula* (5.3) *is automatically redirected to formula* (5.5)*, in case $t.target_1 \in CompositeState_1$.*

*Since we assume that models are finite in size, the state hierarchy is finite in height. Thus there is always an application of formulas* (5.3) *or* (5.4) *after a finite sequence of applications of formula* (5.5)*. Consequently, an event is consumed after a finite number of formula applications. If we also assume that the stream of input events is finite, the execution of any state machine terminates after a finite number of applications.*

Introduction of initial states. *To introduce initial states, we only have to adapt formula* (5.5) *to take the initial state of a composite state into account:*

$$T_2(cs)(es) \quad := \quad cs.initial_2.effect_2 \circ T_2(cs.initial_2)(es) \tag{5.6}$$

*To ensure semantics preservation, the migration needs to preserve the set of action streams for each event stream. However, in this case, we refine the behavior of a state machine by removing possible executions. To be semantics-preserving, the migration needs to significantly extend the model to keep the same executions. However, in this case, semantics preservation is too strict, since we want to remove non-determinism generated by the choice between more initial states. The developer of the model has to decide which executions to remove by choosing an initial state for each composite state. We analyze this problem in more detail in the next section and propose an extension to be able to perform such changes.*

Moore to Mealy machines. *To transition from Moore to Mealy machines, we only have to adapt formulas* (5.3) *and* (5.6) *to consider the effect of the transition instead of the effect of its target state:*

$$T_3(s)(\langle e\rangle \circ es) \quad := \bigcup_{t \in activated_3(s,e)} t.effect_3 \circ T_3(t.target_3)(es) \tag{5.7}$$

$$T_3(cs)(es) \quad := \quad T_3(cs.initial_3)(es) \tag{5.8}$$

*The transition requires a migration function that maps all models with effects in states to models with effects in transitions. To preserve the meaning of all models, the migration function has to determine the effect of a transition based on the state it enters. For a transition $t \in Transition_2$ which is mapped to $t' \in Transition_3$ by the function $mig_{23}$ to migrate models from language version 2 to 3, $t'.effect_3 = effect_2(t.target_2)$ is fulfilled with:*

$$effect_2(s) \quad := \quad \begin{cases} effect_2 \circ effect_2(s.initial_2), & \text{if } s \in CompositeState_2 \\ effect_2, & \text{if } s \in State_2 \end{cases} \tag{5.9}$$

*When a transition enters a composite state, we do not only have to take its effect into account but also the effect of its initial state. Note that this may have to be applied recursively, as the*

*initial state may again be a composite state, and so on. Using the definition of the semantics, we can easily prove that this migration function—which maps state $s \in State_2$ to state $s' \in State_3$—preserves the meaning of all models:*

$$T_2(s)(\langle e \rangle \circ es) := \bigcup_{t \in activated_2(s,e)} t.target_2.effect_2 \circ T_2(t.target_2)(es) \quad (5.10)$$

$$= \bigcup_{t' \in activated_3(s',e)} t'.effect_3 \circ T_3(t'.target_3)(es) \quad (5.11)$$

$$=: T_3(s')(\langle e \rangle \circ es) \quad (5.12)$$

*The implemented algorithm is depicted in Listing 5.1.*

Introduction of regions. *To introduce concurrent regions, we have to adopt formula* (5.8) *for instances $r \in Region_4$ of the new class:*

$$T_4(r)(es) := T_4(r.initial_4)(es) \quad (5.13)$$

*Additionally, we have to replace formula* (5.8) *by:*

$$T_4(cs)(es) := \mathcal{I}\{T_4(r)(es) \mid r \in cs.region_4\} \quad (5.14)$$

*where $\mathcal{I} : \mathcal{P}(\mathcal{P}(Action^*)) \rightarrow \mathcal{P}(Action^*)$ produces all possible interleavings of different sets of traces.*

*The introduction requires a migration function $mig_{34}$ that creates a region for each composite state. To preserve the meaning of all models, $mig_{34}$ has to correctly extract the substates and the initial state of the composite state to the newly created region. A model that has been migrated from language version 3 to 4 thus can only have one region $r' \in Region_4$ for each composite state $cs' \in CompositeState_4$:*

$$T_4(cs')(es) = \mathcal{I}\{T_4(r')(es)\} = T_4(r')(es) = T_4(r'.initial_4)(es) \quad (5.15)$$

*Since there is only one region, there is no interleaving. Because $r'.initial_4$ returns the same state as $cs.initial_3$ from the composite state $cs \in CompositeState_3$ before migration ($cs' = mig_{34}(cs)$), the migration function is semantics-preserving:*

$$T_4(r')(es) := T_4(r'.initial_4)(es) = T_3(cs.initial_3)(es) =: T_3(cs)(es) \quad (5.16)$$

*The algorithm is implemented by applying reusable coupled operations which is shown in Listing 5.2.*

### 5.4.2 Characterizing Model-Specific Migration

As we have seen before, models may have to be migrated to preserve their meaning in response to metamodel evolution. A model migration is defined by a migration function $mig$ which maps the models from the old language $\mathcal{L}_{mm_1}$ with $S_1 = \mathcal{L}_{mm_1} \rightarrow \mathcal{SD}_1$ to models of the new language $\mathcal{L}_{mm_2}$ with $S_2 : \mathcal{L}_{mm_2} \rightarrow \mathcal{SD}_2$. As mentioned before, we require that the migration function respects the relation $\cong \subseteq \mathcal{SD}_1 \times \mathcal{SD}_2$ between the two versions of the semantic domain. Otherwise, information is lost during migration which needs to be avoided. If the relation $\cong$ preserves the equivalence relation on the semantic domains, we call the semantics change a semantics refactoring:

**Definition 5.15** (Semantics Refactoring)**.** *Let $S_1 \mapsto S_2$ be a semantics change from $S_1 : \mathcal{L}_{mm_1} \to \mathcal{SD}_1$ to $S_2 : \mathcal{L}_{mm_2} \to \mathcal{SD}_2$. The semantics change is called semantics refactoring if it preserves the equivalence relation for both semantic domains:*

$$\forall s_1, t_1 \in \mathcal{SD}_1, s_2, t_2 \in \mathcal{SD}_2 : s_1 \equiv t_1 \wedge s_1 \cong s_2 \wedge t_1 \cong t_2 \Rightarrow s_2 \equiv t_2$$

However, there are cases in which the semantics of the language is changed in a way that does not preserve the equivalence relation on the semantic domains. We call such a case a semantics refinement:

**Definition 5.16** (Semantics Refinement)**.** *Let $S_1 \mapsto S_2$ be a semantics change from $S_1 : \mathcal{L}_{mm_1} \to \mathcal{SD}_1$ to $S_2 : \mathcal{L}_{mm_2} \to \mathcal{SD}_2$. The semantics change is called semantics refinement if it cannot always preserve the equivalence relation for both semantic domains:*

$$\exists s_1, t_1 \in \mathcal{SD}_1, s_2, t_2 \in \mathcal{SD}_2 : s_1 \equiv t_1 \wedge s_1 \cong s_2 \wedge t_1 \cong t_2 \wedge s_2 \not\equiv t_2$$

That means that an element of the source semantic domain is associated to different elements in the target semantic domain which are not semantically equivalent. When the semantics is refined, often the migration cannot be performed without information which is not available in the original model as shown below:

**Definition 5.17** (Model-Specific Migration)**.** *Let $S_1 \mapsto S_2$ be a semantics refinement from $S_1 : \mathcal{L}_{mm_1} \to \mathcal{SD}_1$ to $S_2 : \mathcal{L}_{mm_2} \to \mathcal{SD}_2$. Let $R : \mathcal{L}_{mm_1} \to \mathcal{P}(\mathcal{L}_{mm_2})$ be the models from $\mathcal{L}_{mm_2}$ that refine a model from $\mathcal{L}_{mm_1}$ due to semantics refinement:*

$$R(m_1) := \{m_2 \in \mathcal{L}_{mm_2} \mid S_1(m_1) \cong S_2(m_2)\}$$

*A model migration $mig : \mathcal{L}_{mm_1} \to \mathcal{L}_{mm_2}$ is called model-specific if and only if there is a model that is refined by multiple models:*

$$\exists m_1 \in \mathcal{L}_{mm_1} : \|R(m_1)\| > 1$$

*A migration is thus model-independent if $\forall m_1 \in \mathcal{L}_{mm_1} : \|R(m_1)\| = 1$.*

A migration has to be model-specific, if several models in $\mathcal{L}_{mm_2}$ exist which semantically refine a model in $\mathcal{L}_{mm_1}$. In this case, additional information is necessary during migration to choose one of these models which are not semantically equivalent. The set for which the migration has to be model-specific is $\mathcal{L}_{ms} = \{m_1 \in \mathcal{L}_{mm_1} \mid \|R(m_1)\| > 1\}$. Again, only those models are problematic which are built and require model-specific migration, i.e. $\mathcal{L}_{built} \cap \mathcal{L}_{ms}$.

**Example 5.13** (Model-Specific Migration)**.** *We analyze whether the migrations involved in the running example are model-specific or semantics-preserving.*

Introduction of initial states. *The introduction of mandatory initial states is a semantics refinement. In this particular case, the semantics refinement is defined by the subset relationship on sets of streams: a model $m_1 \in \mathcal{L}_{mm}$ refines another model $m_2 \in \mathcal{L}_{mm}$ if it exhibits less non-determinism:*

$$m_1 \leqq m_2 :\Leftrightarrow \forall es \in Event^* : S(m_1)(es) \subseteq S(m_2)(es)$$

*A model migration $mig_{12}$ from language version 1 to 2 cannot always preserve the semantics, but has to refine it, as can be seen for a composite state $cs$ which is mapped to $cs'$ by the migration:*

$$T_2(cs')(es) := T_2(cs'.initial_2)(es) \subseteq \bigcup_{s \in cs.state_1} T_1(s)(es) =: T_1(cs)(es) \tag{5.17}$$

*As a consequence, a number of models with different choices for initial states are possible refinements for each model without initial states. Model-specific information is required to choose the one that is intended by the developer of the model.*

Moore to Mealy machines. *The transition from Moore to Mealy machines is a semantics refactoring, since the relation $\cong$ between the semantic domains preserves the equivalence relation $\equiv$, as shown in Example 5.12. No additional information is required during migration, and we can thus specify an algorithm that automatically performs the migration.*

Introduction of concurrent regions. *The introduction of concurrent regions is also a semantics refactoring, since again $\cong$ preserves $\equiv$.*

### 5.4.3 Coping with Model-Specific Migration

We outline a number of possible solutions to cope with model-specific coupled evolution. The solutions take advantage of particular situations which are however encountered very often in practice.

**Effort Analysis.** In practice, only a small subset $\mathcal{L}_{built} \subset \mathcal{L}_{mm}$ of all possible models of a metamodel $mm$ are actually built. Only the existing models need to be migrated. In many practical situations (e.g. for languages developed only for use inside an organization), the language engineers and users are quite close to each other. In these situations, it is often the case that the entire set of the existing models is known to the language engineers. For instance, whenever the language is used only in-house, all models are contained in a central repository.

In case all models are known, language engineers can make informed language improvements also with respect to the effort needed for model migration. Based on the existing models, they can assess the manual migration effort required after metamodel adaptation. For instance, the effort needed to introduce initial states is proportional to the number of composite states in the existing models, as an initial state has to be chosen for each composite state. They can decide whether a language improvement is worth making given the amount of manual work necessary to migrate the existent models. However, in a lot of cases, language engineers and users are decoupled which makes this approach infeasible.

**Interactive Migration.** One possibility to support model-specific coupled evolution is to provide user interaction during migration. The migration algorithm automatically migrates the model as far as possible, and whenever it needs supplementary information, it asks the language user to provide the missing information. It can also suggest a number of alternatives from which the language user has to choose. We have extended COPE's language to implement a model migration with a primitive to trigger such an interaction during migration.

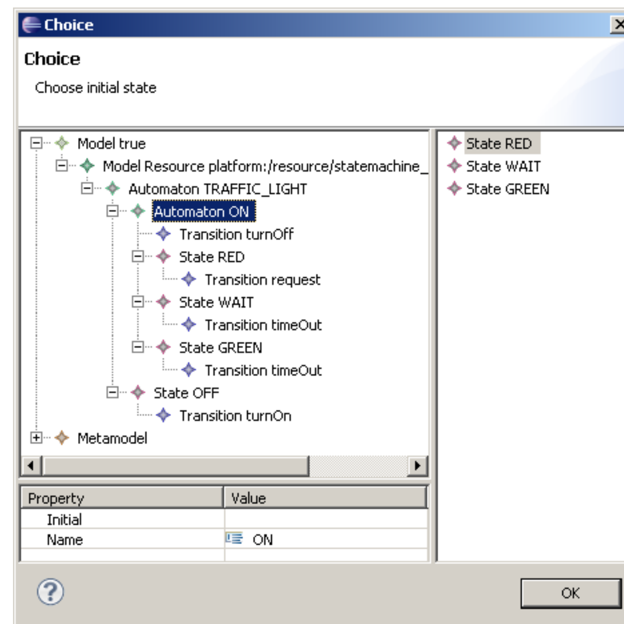



**Figure 5.14:** User interface for interactive coupled operations

**Example 5.14** (Interactive Migration)**.** *Listing 5.5 shows the use of this primitive in an interactive coupled operation that introduces initial states in the language.*

**Listing 5.5:** Interactive coupled operation Introduce Initial States

```
1 // metamodel adaptation
2 newReference(CompositeState, "initial", State, 1, 1, !CONTAINMENT)
3
4 // model migration
5 for(a in CompositeState.allInstances) {
6   a.initial = choose(a, a.state, "Choose initial state")
7 }
```

*The metamodel adaptation creates the new reference* initial *which is mandatory (lower bound 1), single-valued (upper bound 1) and not a composition (not* containment*). During model migration, the developer of the model has to* choose *an initial state for each composite state. The primitive* choose *takes three parameters as input—namely the context element, the values to choose from and a message, and returns the chosen value.*

*Figure 5.14 shows the dialog that is opened during model migration to let the language user make a choice. The dialog shows the current state of the model (selecting the context element), the list of values to choose from, as well as the message. The language user is required to choose a value from the list to determine the initial state for a composite state.*

Currently, the dialog to answer the questions shows the model in abstract syntax. However, the language user would probably prefer to see the model in concrete syntax, when answering the questions. To support that, the concrete syntax needs to be available for the version of the modeling language before the coupled operation. If this is not possible, the interactive coupled operations may also be moved to the beginning or end of the coupled evolution, if dependencies permit. If they are moved

to the end of the coupled evolution, we can also add the questions as todos to the model, so that the language user can answer them after migration.

**Implicit Information.** Often, the language users employ different conventions (e.g. naming conventions) to capture more information than is made explicit through the metamodel. In these cases, the language user can incorporate implicit information to help automate the migration. For example, language users might have already named all initial states with a prefix "I", before the initial states were explicitly introduced in the metamodel. They can then refine the migration in a way that for each composite state, it automatically chooses the marked substate. We thus plan to introduce a means to allow the language user to upfront establish certain choices introduced by the language engineer.

## 5.5 Summary

Just as other software artifacts, modeling languages and thus their metamodels have to be adapted. In order to reduce the effort for the resulting migration of models, adequate tool support is required. In Chapter 3 (*State of the Practice: Automatability of Model Migration*), we have performed a study on the histories of two industrial metamodels to determine requirements for adequate tool support. Adequate tool support needs to support the reuse of migration knowledge, while at the same time being expressive enough for complex migrations. To the best of our knowledge, existing approaches for model migration do not cater for both reuse and expressiveness.

This chapter presented COPE, an integrated approach fulfilling these requirements. Using COPE, the coupled evolution can be incrementally composed of coupled operations that only require specification of the differences of metamodel and models in consecutive versions. The resulting modularity of coupled operations ensures scalability, and is particularly suited to combine reuse with expressiveness. Reuse is provided by reusable coupled operations that encapsulate recurring migration knowledge. Expressiveness is provided by a complete set of primitives embedded into a Turing-complete language, which can be used to specify custom coupled operations. Tracking the performed coupled operations in an explicit language history allows language users to migrate models at a later instant, and provides better traceability of metamodel adaptations.

We implemented these language concepts based on the Eclipse Modeling Framework (EMF). To ease its application, COPE was seamlessly integrated into the metamodel editor, shielding the language engineer from technical details as far as possible.

# Chapter 6

# Tool Support

In the last chapter, we presented a language for the coupled evolution of metamodels and models fulfilling the requirements of reuse and expressiveness. From our experience, language engineers do not want to encode the coupled evolution, but rather prefer to adapt the metamodel directly in an editor. Consequently, COPE is implemented as a non-invasive integration into the existing EMF metamodel editor. Even though COPE is based on the language presented in Chapter 5 (*COPE – Coupled Evolution of Metamodels and Models*), it shields the language engineers from this language as far as possible. COPE is open source and can be obtained from our website[1]. The web site also provides a screencast, documentation and several examples (including the running example from this dissertation). This chapter is partly based on [Herrmannsdoerfer, 2009], [Herrmannsdoerfer and Koegel, 2010a] and [Herrmannsdoerfer, 2011].

**Contents**


**6.1 Recording the Coupled Evolution** . . . . . . . . . . . . . . . . . . . . 149
**6.2 Maintaining the Coupled Evolution** . . . . . . . . . . . . . . . . . . . 152
**6.3 Operation-based Metamodel Versioning** . . . . . . . . . . . . . . . 158
**6.4 Summary** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 166


While Section 6.1 (*Recording the Coupled Evolution*) describes the basic functions of COPE to record the coupled evolution, Section 6.2 (*Maintaining the Coupled Evolution*) describes more advanced functions to inspect, refactor and recover the coupled evolution. Section 6.3 (*Operation-based Metamodel Versioning*) presents the versioning model by means of which the coupled evolution is recorded and stored, before Section 6.4 (*Summary*) concludes this chapter.

## 6.1 Recording the Coupled Evolution

We first describe the workflow that is supported by COPE, before detailing on its integration into the user interface of the EMF metamodel editor.

[1] see COPE web site: `http://cope.in.tum.de`

### 6.1.1 Tool Workflow

Figure 6.1 illustrates the tool workflow using the running example from Chapter 5 (*COPE – Coupled Evolution of Metamodels and Models*). The workflow contributes to all layers of the meta hierarchy.

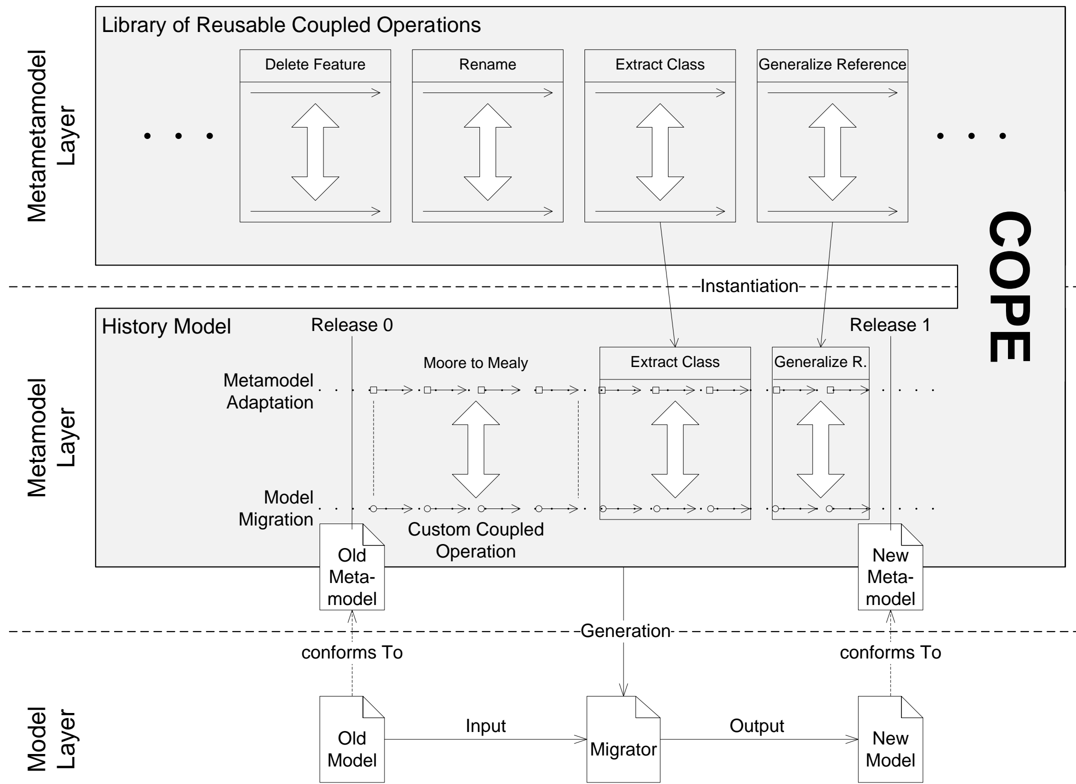



**Figure 6.1:** Tool workflow

**Metametamodel Layer.** COPE provides a *library* of reusable coupled operations that can be invoked on a specific metamodel. Therefore, the library is aware of the signature and the preconditions of the reusable coupled operations. Besides the operations required for the running example, the current library contains a number of other reusable coupled operations like e.g. Rename or Delete Feature. The library is extensible in the sense that new reusable coupled operations can be declared and registered. Reusable coupled operations are declared independently of the specific metamodel, i.e. on the level of the metametamodel.

**Metamodel Layer.** All operations applied to the metamodel are maintained in an explicit *history model*. The history model keeps track of the coupled operations which encapsulate both metamodel adaptation and model migration. It is structured according to the language releases, i.e. the language versions which were deployed. All previous versions of the metamodel can be easily reconstructed from the information available in the history model. In Figure 6.1, the evolution from release 0 to release 1 is the sequence of coupled operations we performed in Chapter 5 (*COPE – Coupled Evolution of Metamodels and Models*).

**Model Layer.** A *migrator* can be generated from the history model that allows for the batch migration of models. The migrator packages the sequence of coupled operations which can be executed to automatically migrate existing models.

### 6.1.2 User Interface

Figure 6.2 shows an annotated screen shot of COPE's user interface. COPE has been integrated into the existing structural *metamodel editor* provided by EMF. This metamodel editor has been extended so that it also provides access to the *history model*. Reusable coupled operations are made available to the language engineer through a special view called *operation browser*. A *migration editor* with syntax highlighting is provided for the specification of custom coupled operations.

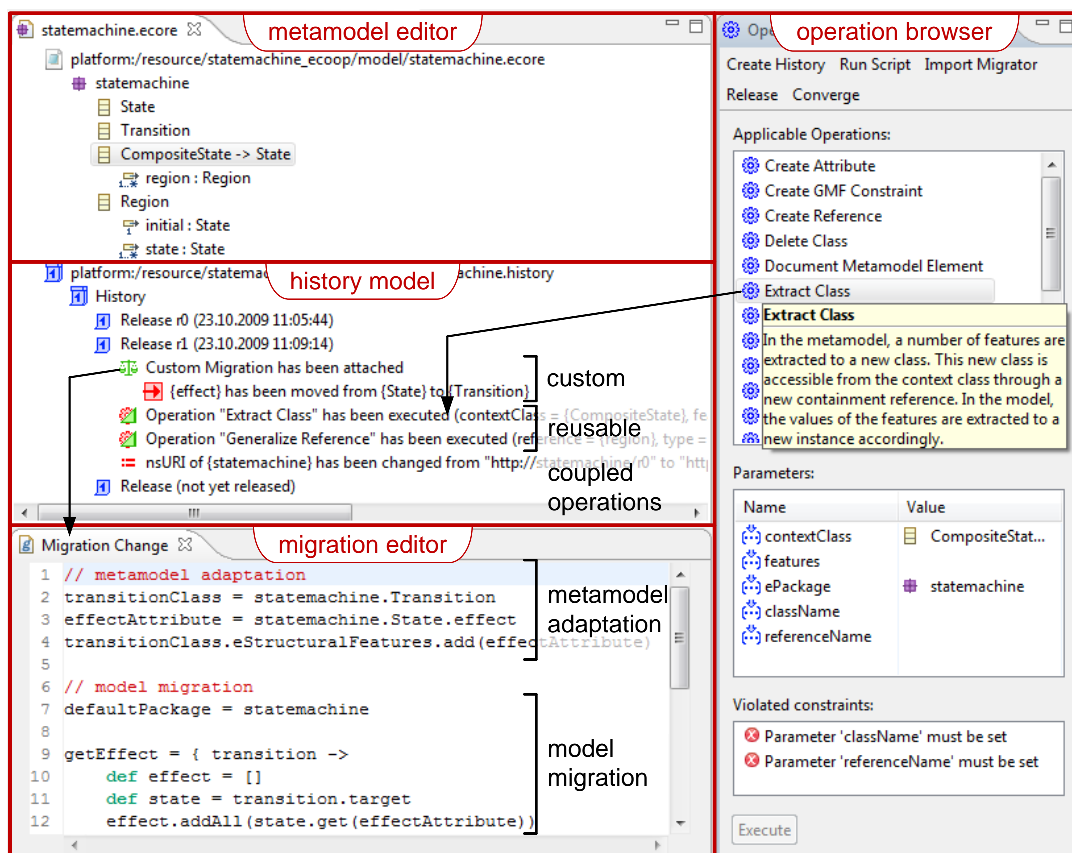



**Figure 6.2:** Integration of COPE into the EMF metamodel editor

**Enabling Reuse.** The language engineer can adapt the metamodel by invoking reusable coupled operations through the operation browser. The browser is context-sensitive, i.e. offers only those reusable coupled operations that are applicable to the elements currently selected in the metamodel editor. The operation browser allows to set the parameters of reusable coupled operations based on their type, and gives feedback on its applicability based on the preconditions. When a reusable coupled operation is executed, its invocation is automatically tracked in the history model. Figure 6.2 shows the Extract Class operation being available in the operation browser,

and the reusable coupled operations stored in the history model. Note that the language engineer does not have to know about the coupled evolution language if he or she is only invoking reusable coupled operations.

**Supporting Expressiveness.** In case no reusable coupled operation is available for the coupled evolution at hand, the language engineer can perform a custom coupled operation. First, the metamodel is directly adapted in the metamodel editor, in response to which the metamodel change operations are automatically tracked in the history. A migration can later be attached to the sequence of metamodel change operations by encoding it in the language presented in Chapter 5 (*COPE – Coupled Evolution of Metamodels and Models*). Note that the metamodel adaptation is automatically generated from the change operations tracked in the history model. In order to allow for different metamodeling habits, adapting the metamodel and attaching a model migration is temporally decoupled such that a model migration can be attached at any later instant. Figure 6.2 shows the model migration attached to the manual metamodel change operations recorded in the history model in a separate editor with syntax highlighting.

**Migrator Generation.** The operation browser provides a release button to create a released version of the metamodel. After release, the language engineer can initiate the automatic generation of a migrator.

## 6.2 Maintaining the Coupled Evolution

After testing the model migration, the language engineer might find out that the recorded history does not perfectly specify the intended model migration, and thus needs to modify it. When the metamodel is derived from another artifact, the history cannot be recorded and hence needs to be recovered from the metamodel versions before and after the adaptation. To address these issues, COPE provides advanced tool support to inspect, refactor and recover the coupled evolution.

### 6.2.1 Inspecting the Coupled Evolution

The recorded history model allows the language engineer to understand the intention behind the metamodel adaptation. Based on the history model, COPE provides the following functions to ease understanding the coupled evolution.

**Identifying Breaking Operations.** Breaking metamodel change operations perform changes which can possibly invalidate existing models. To prevent errors during coupled evolution, these operations need to have a model migration attached. Based on Section 5.3.2 (*Breaking Metamodel Changes Revisited*), COPE provides an analysis to identify breaking operations which do not yet have a model migration attached. Breaking operations are identified on the metamodel-level, i.e. independently of the existing models. Figure 6.3 shows how the user interface displays breaking operations. When the custom migration is not yet attached to the movement of the attribute effect from class State to Transition, we can identify the metamodel adaptation

as a breaking operation.

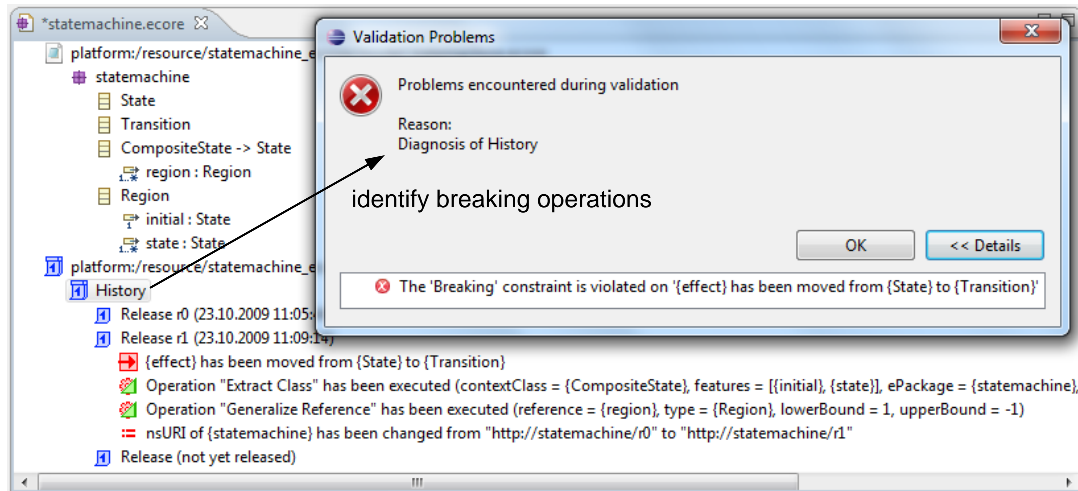



**Figure 6.3:** Identifying breaking operations

**Metamodel Reconstruction.** To understand the evolution, COPE allows the language engineer to reconstruct metamodel versions from the history model. Earlier metamodel versions can be simply reconstructed by interpreting the primitive operations recorded in the history model. Thus it is not necessary to store all the intermediate metamodel versions which would require a large memory footprint. This reconstruction is interactive, allowing the language engineer to browse through the history model. When the language engineer selects an operation in the user interface, COPE reconstructs the snapshot of the metamodel right after the operation. The metamodel snapshot is shown in a separate view through which it can be inspected. Figure 6.4 shows a screen shot of the so-called *reconstruction view* in action. It displays the reconstructed version of the metamodel after the transition from Moore to Mealy machine, but before the introduction of concurrent regions.

**History Differencing.** Comparing two metamodel snapshots in EMF does not always yield an accurate difference model. This is due to the fact that—in the absence of universally unique identifiers—the matching between the metamodel elements from the two snapshots has to be inferred. To produce a more accurate difference model, the matching can be generated from the history model. In the user interface, COPE allows the language engineer to select the source and target version for the comparison directly in the history model. COPE produces a view showing the difference model between the two metamodel versions. The so-called *comparison view* displayed in Figure 6.5 shows the *differences* between the version *before* the first operation and the version *after* the last operation. For instance, the highlighted operation—moving the reference state from class CompositeState into the new class Region—can currently not be inferred correctly by EMF Compare[2] (see Figure 6.7 for the differences as inferred by EMF Compare).

**Checking Integrity.** COPE stores both the current version of the metamodel as well

[2]see EMF Compare web site: `http://wiki.eclipse.org/EMF_Compare`

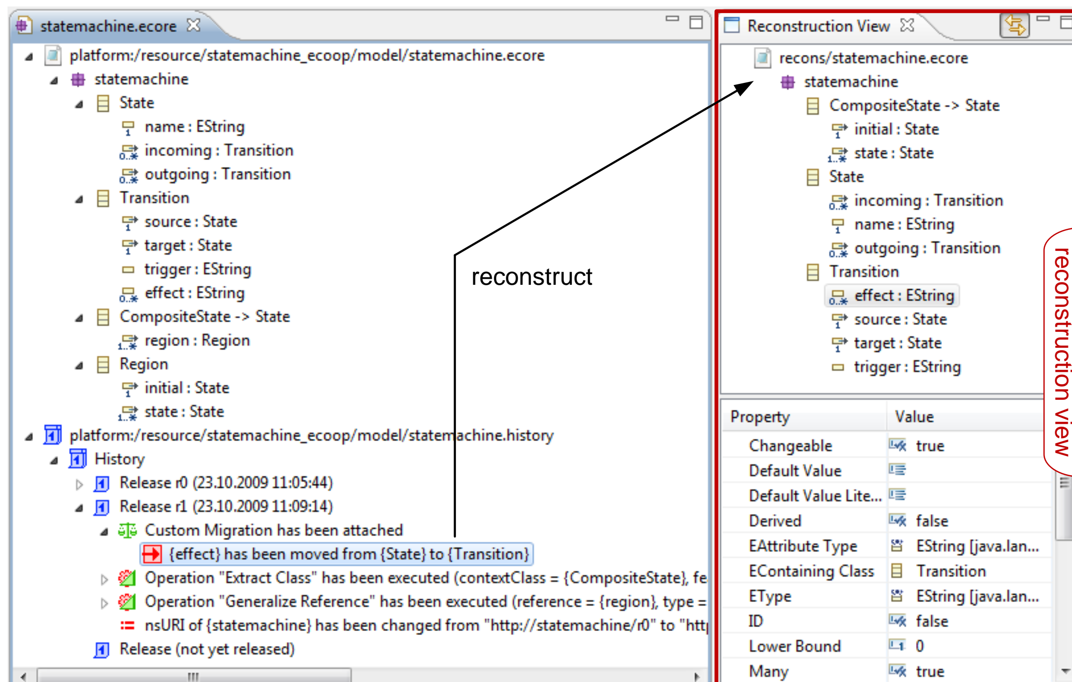



**Figure 6.4:** Metamodel reconstruction

as its history model. This is redundant, as the current metamodel version could be reconstructed from the history model. However, other artifacts—like e.g. models to define the concrete syntax—depend on the current version of the metamodel. Due to the redundancy, the history model must be consistent with respect to the metamodel: it must reconstruct the current version of the metamodel. COPE thus provides a function to check the integrity of the history model. This function reconstructs the current metamodel version from the history and compares it with the stored metamodel version.

### 6.2.2 Refactoring the Coupled Evolution

As the recorded history does not always perfectly specify the intended model migration, COPE provides functions to refactor the history model. The refactorings have to preserve the meaning of the history model: the history model has to reconstruct the current metamodel version.

**Flattening Operations.** When the language engineer records the coupled evolution, he or she might define an incorrect migration by applying the wrong reusable coupled operation or by incorrectly encoding a custom coupled operation. To remove the incorrect migration, a custom or reusable coupled operation can be flattened in COPE. In order to do so, the coupled operation is replaced by the equivalent sequence of primitive operations. Thereby, the metamodel adaptation is preserved, and only the model migration is altered. The resulting primitive operations can be enriched by a different custom migration or replaced by a different reusable coupled operation as explained in the following paragraph.

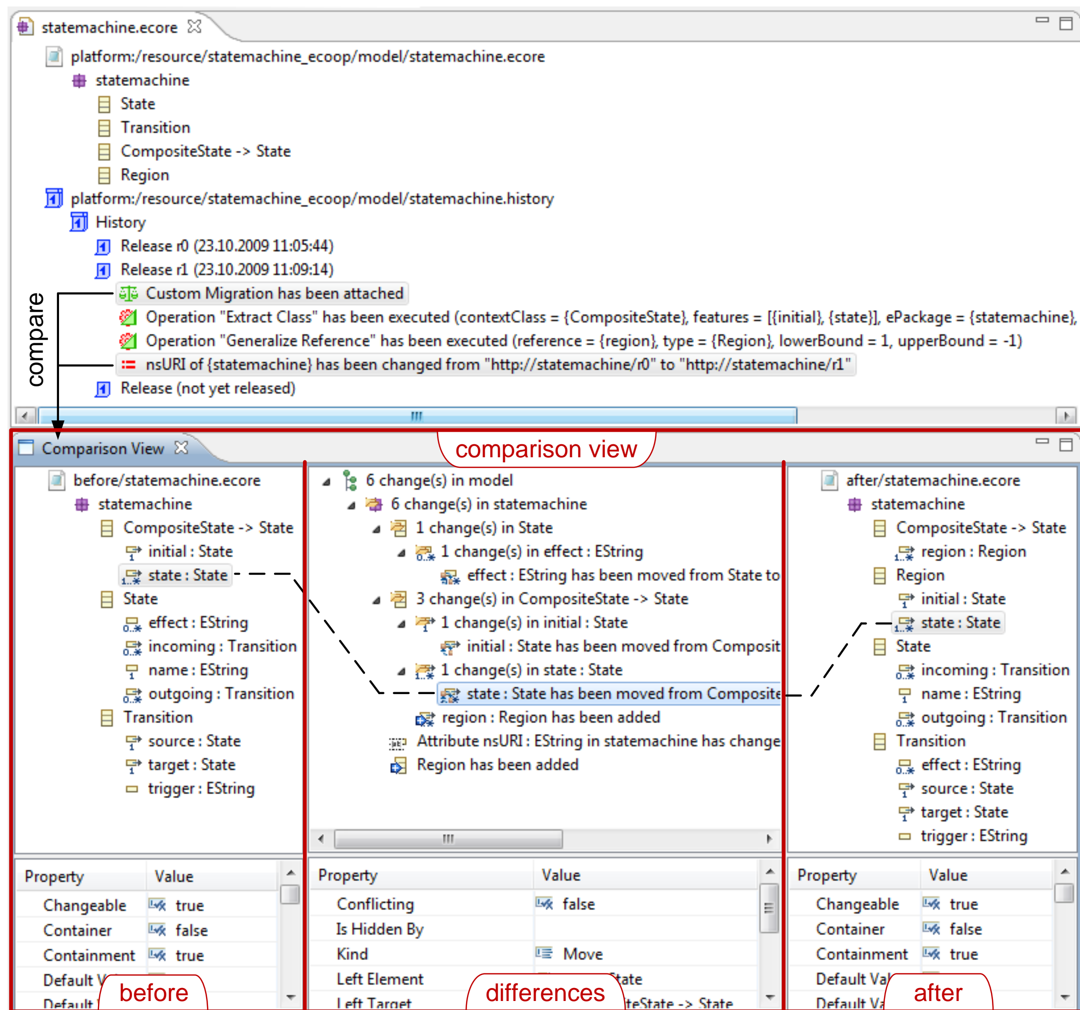



**Figure 6.5:** History differencing

**Replacing Operations.** For certain breaking operations, we might later identify a reusable coupled operation that provides the intended model migration. Rather than manually reimplementing the model migration, it is better to apply the reusable coupled operation instead. The primitive operations, however, have already been recorded to the history model. COPE thus provides support to replace a sequence of primitive operations with the application of a reusable coupled operation. COPE reconstructs the metamodel version before the operations and presents it to the language engineer in a dialog where he or she can select and apply the appropriate reusable coupled operation. Figure 6.6 depicts the *replacement dialog* to replace primitive operations by an application of the Extract Class operation. The dialog shows the metamodel version *before* the operations, the *primitive operations* that need to be replaced, and the *operation browser* to select the operation. To keep the history model consistent, the primitive operations can only be replaced, in case the application of the reusable coupled operation yields the same result.

**Reordering Operations.** Only consecutive primitive operations can be replaced by a reusable coupled operation or can be enriched by a custom model migration. When

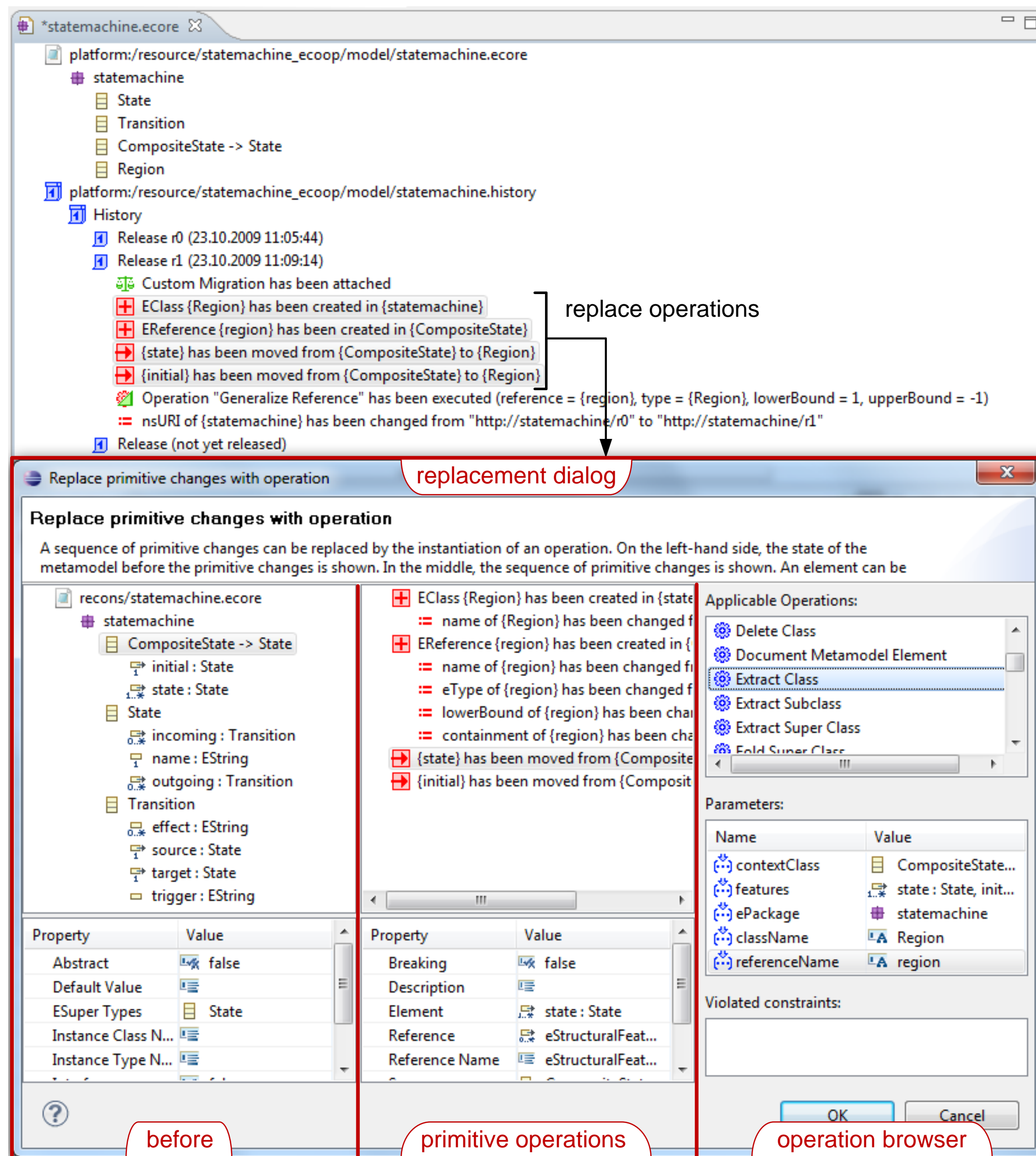



**Figure 6.6:** Replacing operations

the operations which we want to replace or enrich are not performed together, they may not be consecutive. Certain operations, however, are independent of each other and thus can be reordered to make them consecutive. COPE therefore provides support to move operations to another position in the history model. To ensure the overall consistency of the history model, the following constraints need to be fulfilled: The operations can only be moved to an earlier position, if they do not depend on the operations that are jumped over, and the operations can only be moved to a later position, if the operations that are jumped over do not depend on them.

**Undoing Operations.** When performing further operations, earlier operations might prove to be wrong. Manually performing the reverse operations is a possible solution, but might lead to a different intention when regarding the model migration. For

example, deleting an attribute and creating it again would lead to the loss of the attribute's values in the model. Therefore, COPE provides support to undo operations after they have already been recorded. To ease undoing operations, the operations are stored in the history model both in forward and reverse direction. Then the operations to be undone need to be simply applied in the reverse direction. To preserve the consistency of the history model, operations can only be undone if no later operations depend on them.

### 6.2.3 Recovering the Coupled Evolution

There are certain cases where COPE cannot be used during the adaptation of the metamodel. For example, the metamodel might be generated from another artifact, and therefore a different tool is used to edit the artifact. In these cases, the history model needs to be recovered from the metamodel versions before and after the adaptation.

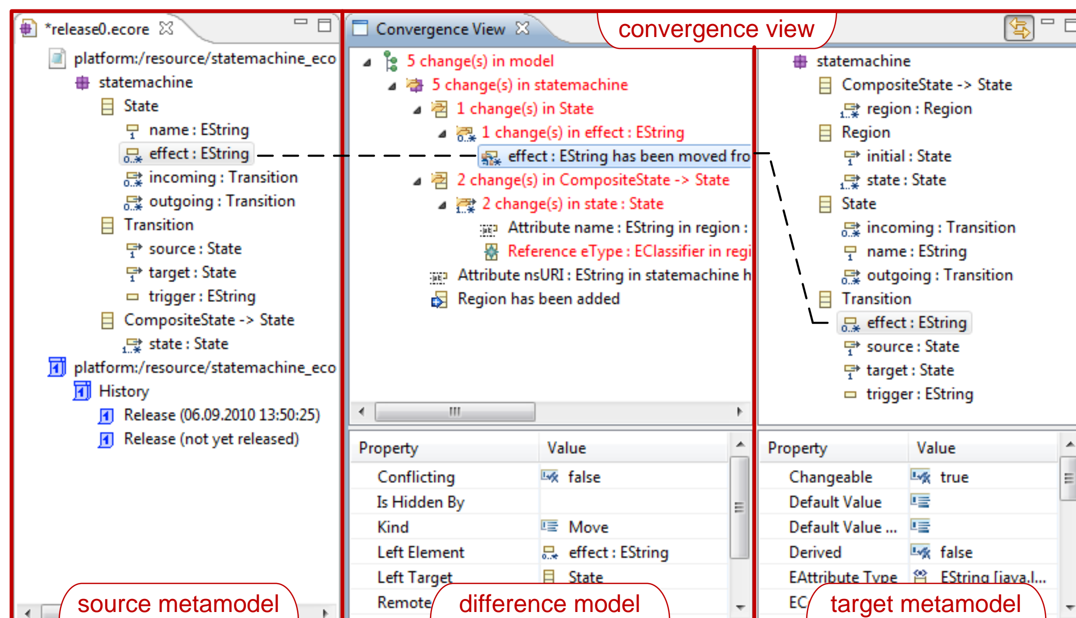



**Figure 6.7:** Metamodel convergence

**Metamodel Convergence.** COPE provides advanced tool support to reverse engineer the history model. Figure 6.7 depicts the user interface to let a source metamodel version converge to a target metamodel version. The *source metamodel* version is loaded directly in the metamodel editor, whereas the *target metamodel* is displayed in a separate view. This so-called *convergence view* also displays the current *difference model* which results from the comparison between the source and target metamodel. The differences are linked to the metamodel elements from both source and target version to which they apply. Breaking changes in the difference model—which necessitate a model migration—are highlighted in red. By means of the operation browser, the language engineer can apply reusable coupled operations to bring the source metamodel nearer to the target metamodel. After an operation is executed on

the source metamodel, the difference is automatically updated to reflect the operations. Changes in the difference model which are not breaking can be easily applied to the source metamodel by double-clicking on them.

## 6.3 Operation-based Metamodel Versioning

Figure 6.8 shows the architecture of the tool. In the center, the *history model* stores the operations applied to a metamodel. The history model conforms to the *history metamodel* which provides the constructs to version metamodels. Based on the metamodel, a number of components are necessary to implement the functions presented in the previous two sections. A *recording* component records the operations when they are applied to the metamodel in the editor. To reconstruct metamodel versions, the *reconstruction* component interprets the recorded history model. The *refactoring* component allows the language engineer to safely modify the history model.

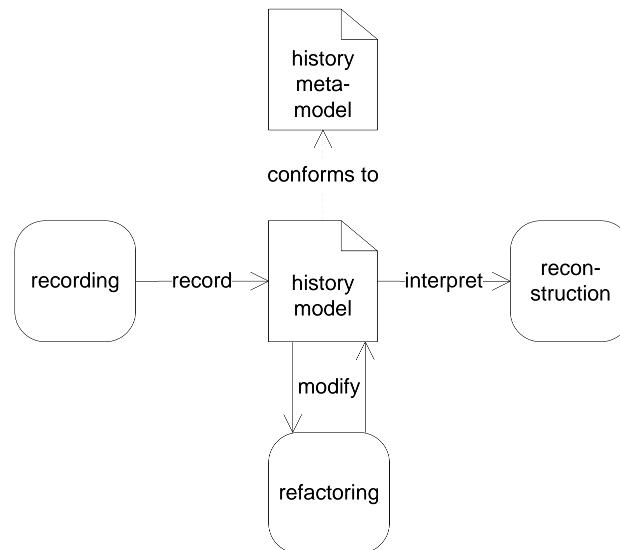



**Figure 6.8:** Tool architecture

Section 6.3.1 (*History Metamodel*) presents the history metamodel. The recording and reconstruction components are introduced in Section 6.3.2 (*Recording and Interpreting the History*). Finally, Section 6.3.3 (*Preserving the History*) explains how the consistency of the history model can be preserved when refactoring it.

### 6.3.1 History Metamodel

The history of a metamodel is maintained in a separate model which conforms to a special metamodel called the *history metamodel*. The history model decorates the metamodel, i.e. directly refers to the elements of the current metamodel version. In contrast, the metamodel is not aware of the history, i.e. there are no links from metamodel to history elements. In the following, we introduce the different concepts provided by the history metamodel.

**Metamodel History.** COPE provides linear versioning of a metamodel, i.e. the history is a sequence of metamodel versions. As is depicted in Figure 6.9, the History of a metamodel consists of a sequence of releases. A Release is a metamodel version which has been deployed and for which models can thus exist. When a metamodel version is released, the date is stored and an identifying label can be provided. Moreover, the namespace URI of the metamodel has to be changed, as it is used to version models. A Release consists of the sequence of operations which have been performed since the previous release. Operation is an abstract class having a number of concrete subclasses which are introduced in the following. The rationale behind an Operation can be documented by a comment.

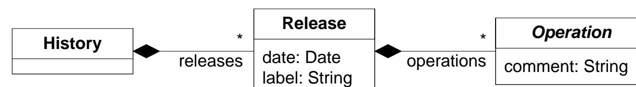



**Figure 6.9:** Metamodel history

Metamodel change operations are bidirectional, i.e. maintain information to be applied in both forward and reverse direction. The reversibility of operations is required for the function *Undoing Operations* explained in Section 6.2.2 (*Refactoring the Coupled Evolution*). To be able to reconstruct earlier metamodel versions from the beginning, the history has thus to start with an empty metamodel. That means that the first release consists of the operations which have led to the initial metamodel version. Nevertheless, COPE does not have to be used from the beginning, as a default history can be easily generated for an existing metamodel. The last Release within a History is always a container for the current operations and is not yet released. When the language engineer requests a release, its date and label are initialized and a new Release is appended to the History.

**Primitive Operations.** A PrimitiveOperation is an operation which can not be decomposed. The primitive operations provided by COPE are complete in the sense that every metamodel adaptation can be described by composing them. As was already mentioned before, the primitive operations directly refer to the elements (called EModelElement in EMF) from the current metamodel version to which they apply. As is depicted in Figure 6.10, there are two basic kinds of primitive operations: ValuePrimitive and ContentPrimitive.

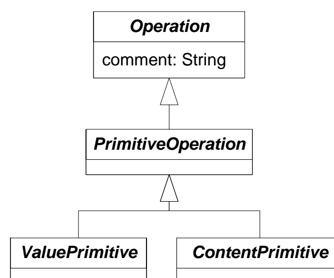



**Figure 6.10:** Metamodel for primitive operations

*ValuePrimitive.* A ValuePrimitive modifies the property of an existing metamodel element. Figure 6.11 illustrates the supported ValuePrimitives. A feature denotes a property of an element which may either be an attribute or a reference. This leads to

two kinds of values: attributeValue and referenceValue. Set sets the value of a single-valued feature of an element. To be reversible, Set needs to maintain the old value which is either oldAttributeValue or oldReferenceValue depending on the type of the feature. Add adds a value to a multi-valued feature of an element, while Remove removes a value from it. For multi-valued features, we abstract away the indices to which values are added or from which they are removed, since the order of elements does not change the semantics of a metamodel.

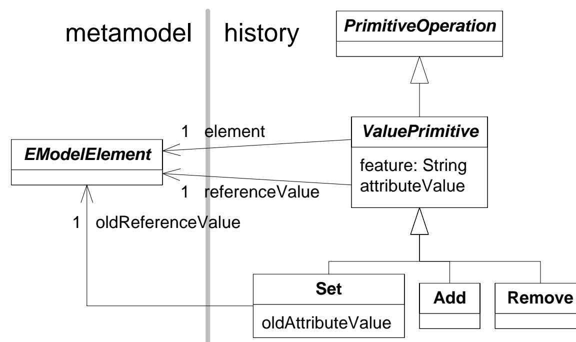



**Figure 6.11:** Metamodel for value primitives

*ContentPrimitive.* A ContentPrimitive modifies the structure of the metamodel. Figure 6.12 illustrates the supported ContentPrimitives. Create creates an element as a child of a target element. Move moves an element to a different parent—the target element. To be reversible, Move also needs to maintain the source element. Delete deletes an element together with its children and the links targeting them. To be reversible, Delete also needs to maintain the target which denotes the parent of the deleted element. As element is a composition, the deleted element is saved in the history and can still be referred to by other earlier operations. Through their superclass InitializerPrimitive, Create and Delete can contain a number of ValuePrimitives: in case of Create, the operations initialize the properties of the newly created element, while in case of Delete, the operations remove links to deleted elements. The latter are only required to support reversibility.

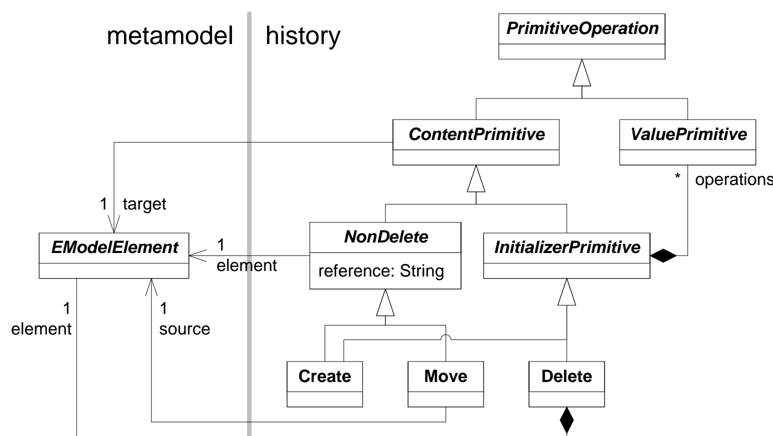



**Figure 6.12:** Metamodel for content primitives

**Composite Operations.** A CompositeOperation is composed of a sequence of operations. Figure 6.13 depicts the metamodel extract to express CompositeOperations. The operations are of type PrimitiveOperation, i.e. we only support one level of composition. CompositeOperation has a subclass for each kind of coupled operation: CustomCoupledOperation and ReusedCoupledOperation.

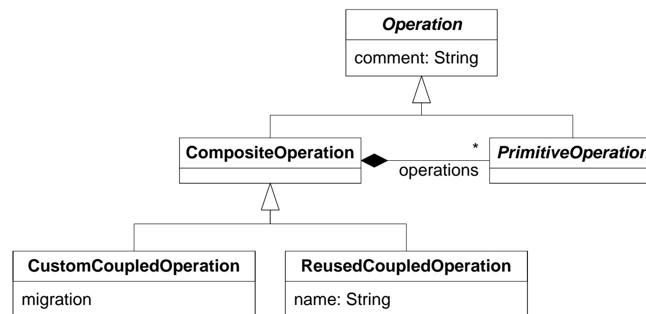



**Figure 6.13:** Metamodel for composite operations

*Custom coupled operation*s allow language engineers to attach a custom migration to a metamodel adaptation. Thereby, a CustomCoupledOperation provides instructions for model migration. The migration is encoded in the coupled evolution language presented in Chapter 5 (*COPE – Coupled Evolution of Metamodels and Models*). Given a model conforming to the metamodel version before the operations, the migration has to transform it to a model conforming to the metamodel version after the operations.

*Reusable coupled operation*s allow language engineers to reuse recurring combinations of metamodel adaptation and model migration. Figure 6.14 depicts the metamodel extract to express reusable coupled operations in the history model. A ReusableCoupledOperation—that is introduced in Section 5.3.4 (*Implementing Coupled Operations*)—has a name and abstracts from a specific metamodel by means of parameters. For each Parameter, a name as well as a type have to be declared. Both metamodel adaptation and model migration of a reusable coupled operation are implemented in the language presented in Chapter 5 (*COPE – Coupled Evolution of Metamodels and Models*). A number of reusable coupled operations are made available to the language engineer through a Library.

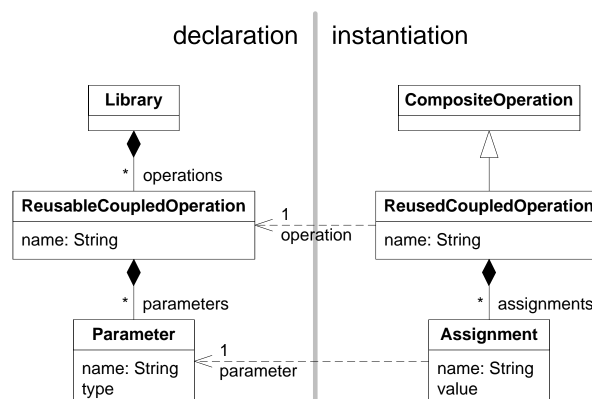



**Figure 6.14:** Metamodel for reusable coupled operations

When a reusable coupled operation is executed, its instantiation is recorded in the

history as a ReusedCoupledOperation. The ReusedCoupledOperation encapsulates the assignments of the parameters of the reusable coupled operation. For each parameter of the operation, its value is maintained by an Assignment. Both ReusedCoupledOperation and Assignment refer to the declaration of the reusable coupled operation only by name. However, the direct links can be established lazily through the library. Therefore, the corresponding references are dashed in Figure 6.14. To be able to reconstruct a metamodel in the absence of the library, we also record the sequence of primitive operations which is equivalent to the execution of the reusable coupled operation.

### 6.3.2 Recording and Interpreting the History

To realize the functions presented in Section 6.1 (*Recording the Coupled Evolution*) and Section 6.2 (*Maintaining the Coupled Evolution*), COPE needs to be able to record the history of a metamodel and to reconstruct earlier metamodel versions from the history.

**History Recording.** COPE allows language engineers to change the metamodel directly in the editor, and automatically records the operations in the history model. Figure 6.15 depicts the flow of information while recording the history model. The *metamodel* is modified by executing commands through the *editor*. Before each executed *command*, the recorder is started, and is stopped right after the execution. The recorder uses the *observer* mechanism provided by EMF to listen to operations applied to the metamodel. EMF provides information about each primitive operation applied to the metamodel through a *notification*.

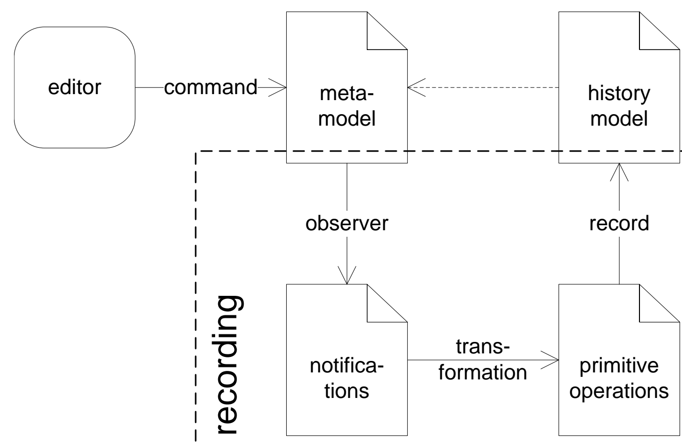



**Figure 6.15:** History recording

The kinds of EMF notifications are similar to the ValuePrimitives of the history metamodel. Consequently, the recording mechanism for those operations can be implemented in a straightforward manner. Only ContentPrimitives cannot be read directly from the notifications and thus have to be inferred. If an element is added to a composite reference and has not been removed from another composite reference before, then we infer a Create of this element. If an element has been removed from a composite reference and is not added to another composite reference afterwards, then we

infer a Delete of this element. In the remaining case, we infer a Move of the element.

Finally, the obtained *primitive operations* are recorded in the *history model*. In case a command denotes the execution of a reusable coupled operation, the recorded operations have to be enclosed in a ReusedCoupledOperation encapsulating the instantiation of the operation.

**Metamodel Reconstruction.** Earlier versions of the metamodel can be easily reconstructed by interpreting the history, as required by the function *Metamodel Reconstruction* explained in Section 6.2.1 (*Inspecting the Coupled Evolution*) and the functions *Replacing Operations* and *Checking Integrity* explained in Section 6.2.2 (*Refactoring the Coupled Evolution*). To rebuild a certain metamodel version, we have to execute all primitive operations starting from the beginning up to this version. Figure 6.16 depicts the flow of information while reconstructing a metamodel version from the history. The *reconstructor* traverses the *history model* and reconstructs the metamodel from scratch. When a new metamodel element is created in the course of the history, a correspondence can be established between the reconstructed element and the original element. During reconstruction, the reconstructor maintains all such correspondences in a bidirectional *mapping*.

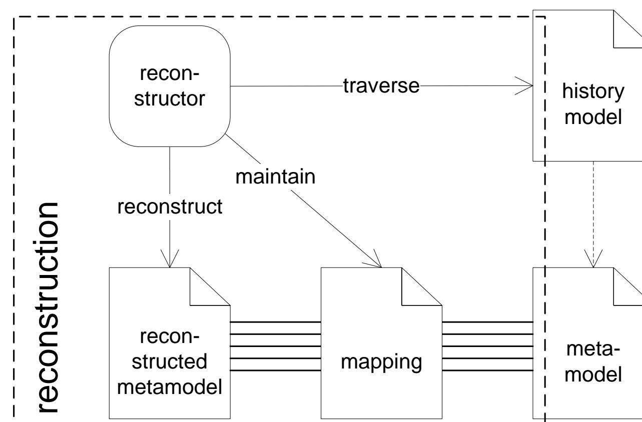



**Figure 6.16:** Metamodel reconstruction

*Mapping.* The concept of *mapping* is particularly useful for the comparison of two metamodel versions. A mapping to the current metamodel is maintained for each version during reconstruction. The two mappings can be composed to a single mapping stating the correspondences between the two metamodel versions. This mapping can be used as a starting point to calculate and present the differences between them, as used by the function *History Differencing* in Section 6.2.1 (*Inspecting the Coupled Evolution*). This technique overcomes the limitation of EMF that the differences cannot be unambiguously determined due to the lack of unique identifiers for metamodel elements.

*Reconstructor.* The *reconstructor* can be parametrized in order to perform arbitrary tasks during reconstruction. One such parametrization—that is necessary for the function *Identifying Breaking Operations* explained in Section 6.2.1 (*Inspecting the Coupled Evolution*)—checks whether operations are breaking, while reconstructing the different metamodel versions. Another parametrization generates the migrator

which consists of earlier metamodel releases and migration scripts in between. The migration scripts simply invoke the sequence of coupled operations from one metamodel release to the next to automatically migrate an existing model.

### 6.3.3 Preserving the History

Since the purpose of the recorded history model is to specify the coupled evolution, it may need to be modified to change the migration. This is different from the operation-based versioning of models [Koegel et al., 2009b], where the history model is not changed, after it has been recorded. When the evolution of a model is recorded, the history model usually fulfills certain constraints and has a certain meaning. Transformations on the history model need to ensure that they preserve the constraints as well as the meaning of the history model.

**Preserving the History's Constraints.** A history model is meaningful if it conforms to the history metamodel, i.e. fulfills the constraints defined by the history metamodel. To ensure conformance of a history model, a transformation needs to preserve the constraints. Besides the constraints defined by the class diagrams in Section 6.3.1 (*History Metamodel*), the following constraints are required:

1. The sequence of primitive operations that specifies the evolution of the metamodel and that can be obtained by flattening the composite operations needs to be consistent:
    a) An element can only be used after it is created and before it is deleted. An element is used in a primitive operation if
        i. in case of a ValuePrimitive, it is either the element, the referenceValue or the oldReferenceValue (Set) of the operation, or
        ii. in case of a ContentPrimitive, it is either the element, the target or the source (Move) of the operation.
    b) A value can only be removed from the feature of an element if it has been added to the same element and feature before.
    c) The information to reverse the primitive operations needs to be consistent with the information to reconstruct the metamodel, i.e.
        i. the old value of a Set needs to be the new value of the last Set on the same element and feature, and
        ii. the source of a Move or the target of a Delete needs to be the previous parent of the element, i.e. where it was created or moved to last.
2. The composite operations need to be consistent within their context in the history model:
    a) A CustomCoupledOperation needs to be attached consistently in the history model, i.e. the invocation of the migration must have the same effect as the enclosed primitive operations.
    b) A ReusedCoupledOperation needs to be applied consistently in the history model, i.e.

i. the applied reusable coupled operation defined by the ReusedCoupledOperation must be applicable, meaning that all its preconditions need to be fulfilled, and

ii. the application of the reusable coupled operation defined by the ReusedCoupledOperation must have the same effect as the enclosed primitive operations.

*Recording functions.* The functions to form coupled operations explained in Section 6.1 (*Recording the Coupled Evolution*) need to preserve constraint 2. The function to attach a custom migration to a sequence of primitive operations preserves constraint 2a by initializing the migration with the primitive operations encoded in the coupled evolution language. The function to apply a reusable coupled operation preserves constraint 2b by checking the preconditions and recording the primitive operations that are equivalent to the application of the operation.

*Refactoring functions.* The functions to refactor the history model explained in Section 6.2.2 (*Refactoring the Coupled Evolution*) need to preserve different constraints. The function *Flattening Operations* automatically preserves all the constraints, as it only removes composite operations that need to be kept consistent with the primitive operations. The function *Replacing Operations* only needs to preserve constraint 2b, as it only introduces a ReusedCoupledOperation. The function *Undoing Operations* needs to preserve all the constraints for the primitive and composite operations that occur after the operations to be undone. We only allow language engineers to undo top-level operations directly contained by a Release, since undoing operations contained by composite operations violates constraint 2. The function *Reordering Operations* needs to preserve all the constraints for the primitive and composite operations that occur between the source and target of the movement of an operation. With the same explanation as above, only top-level positions can serve as source and target of the movement.

**Preserving the History's Semantics.** A history model is interpreted by reconstructing metamodel versions. In particular, the current version of the metamodel can be reconstructed by traversing the complete history model. We define the semantics of a history model to be the reconstructed current version of the metamodel. However, since the metamodel is subject to evolution, the current version of the metamodel may also change. This is consistent with the fact that COPE maintains the current metamodel version separately from the history model.

*Recording functions.* The functions to record the history model explained in Section 6.1 (*Recording the Coupled Evolution*) change both the current metamodel version as well as the history model. The changed history model needs to reconstruct the changed current version of the metamodel. This needs to be ensured by the recorder that is explained in Section 6.3.2 (*Recording and Interpreting the History*).

*Refactoring functions.* Moreover, the functions to refactor the history model explained in Section 6.2.2 (*Refactoring the Coupled Evolution*) need to preserve the meaning of the history model with respect to the current version of the metamodel. The functions *Flattening Operations* and *Replacing Operations* are safe, since they neither change the meaning of the history model nor the current metamodel version. However, the function *Undoing Operations* changes both, while the function *Reordering Opera-*

*tions* changes only the history model. Of course, *Undoing Operations* needs to apply the reversed operations to the current metamodel version to preserve semantics. In addition, both functions have to ensure that the removed or moved operations do not alter the reconstruction. They alter the reconstruction, in case the effect of operations depends on their order. These operations are called *conflicting* operations [Koegel et al., 2010b]. Two primitive operations are conflicting if

- both are Sets and change the same feature of the same element to a different value, or
- both are Moves and move the same element to a different target.

For *Undoing Operations*, the undone primitive operations should not conflict with the operations after them. For *Reordering Operations*, the moved primitive operations should not conflict with the operations between the source and target position of the movement. Due to the reversibility of the primitive operations, the conflicting operations are already identified by the constraints mentioned above.

## 6.4 Summary

To ease modeling language evolution, adequate tool support is required to automate the migration of models. To obtain a correct model migration, COPE records the coupled evolution of metamodels and models in an explicit history model. As a result, the history model stores the sequence of coupled operations that have been performed during evolution. To make the operation-based approach usable in practice, more advanced tool support is necessary to maintain the history model. COPE provides functions to inspect, refactor and recover the history model to better understand, correct and reverse engineer the coupled evolution. These functions are required to be able to perform real-life case studies in order to evaluate the approach.

Chapter 7

# Case Studies

To demonstrate the applicability of COPE, we have performed six case studies. In each case study, we have applied COPE to automate the coupled evolution of metamodels and models. The case studies cover modeling languages from different domains and have been performed with different goals in mind. The domains are software architecture, graphical syntax, software quality, project planning and software in general. We performed the following kinds of case studies:

- *Reverse engineering:* To demonstrate the viability of the approach, we performed two case studies that reverse engineer the model migration, after the metamodel adaptation has already been carried out: Palladio Component Model and Graphical Modeling Framework.
- *Forward engineering:* Since reverse engineering is not the primary scope of COPE, we performed two more case studies that forward engineer the model migration by adapting the metamodel directly with COPE: Quamoco Quality Metamodel and Unicase Unified Model.
- *Comparison:* To gain insight into the advantages and disadvantages over other approaches, we compared COPE with other model migration tools in two more case studies: Transformation Tool Contest and a Comparison of Model Migration Tools.

Some of the case studies have been published in [Herrmannsdoerfer et al., 2009a], [Herrmannsdoerfer et al., 2009c], [Herrmannsdoerfer, 2010] and [Rose et al., 2010a].

**Contents**


**7.1 GMF Generator Model and Palladio Component Model . . . . . . 168**
**7.2 Graphical Modeling Framework . . . . . . . . . . . . . . . . . . . . 175**
**7.3 Quamoco Quality Metamodel . . . . . . . . . . . . . . . . . . . . . 189**
**7.4 Unicase Unified Model . . . . . . . . . . . . . . . . . . . . . . . . . 197**
**7.5 Transformation Tool Contest . . . . . . . . . . . . . . . . . . . . . . 204**
**7.6 Comparison of Model Migration Tools . . . . . . . . . . . . . . . . 215**
**7.7 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 228**

In Section 7.1 (*GMF Generator Model and Palladio Component Model*), we investigate the automatability of model migration by reverse engineering the coupled evolution of the Palladio Component Model and the GMF Generator Model. In Section 7.2 (*Graphical Modeling Framework*), we examine GMF in more detail by considering its other three metamodels and by analyzing the causes and impacts of metamodel adaptation. In Section 7.3 (*Quamoco Quality Metamodel*), we investigate the automatability of model migration by forward engineering the coupled evolution of the Quamoco metamodel for specifying software quality. In Section 7.4 (*Unicase Unified Model*), we forward engineer not only the model migration, but also the migration of changes maintained by Unicase to version models for project planning and software modeling. In Section 7.5 (*Transformation Tool Contest*), we report the results of the participation of COPE in the migration case study of the Transformation Tool Contest (TTC). In Section 7.6 (*Comparison of Model Migration Tools*), we compare COPE to other EMF-based model migration tools by evaluating them in a common case study along different criteria. We sum up the central results of all case studies in Section 7.7 (*Summary*).

All the case studies follow the typical structure of empirical studies:

1. *Study goal:* The research questions that should be answered by the case study.
2. *Study object:* The objects that are used as input to the case study (and eventually the subjects that participate in the case study).
3. *Study execution:* The steps that are applied to the objects to answer the research questions.
4. *Study result:* The results that are obtained from applying the steps to the objects.
5. *Study discussion:* The interpretation of the results with respect to the research questions.
6. *Threats to validity:* The threats that may affect the validity of the result together with the measures taken to mitigate them.

## 7.1 GMF Generator Model and Palladio Component Model

COPE has been designed based on the requirements resulting from the empirical study presented in Chapter 3 (*State of the Practice: Automatability of Model Migration*). To revalidate these results on the evolutions of different modeling languages, we used COPE to reverse engineer the coupled evolution of two existing metamodels: a metamodel developed as part of an open source project, and another one developed as part of a research project.

### 7.1.1 Study Goal

The study was performed to evaluate the applicability of COPE to real-world coupled evolution and to better understand the potential for reuse of recurring migration knowledge. More specifically, the study was performed to answer the following

research questions:

- **RQ1.** *Which fraction of the changes are metamodel extensions that do trivially not require a migration of models?* Since simple metamodel extensions do not require a model migration, they have to be regarded separately.
- **RQ2.** *Which fraction of the changes can be reused by means of reusable coupled operations?* The higher the fraction of reusable coupled operations, the higher the degree of potential automation for the model migration.
- **RQ3.** *Which fraction of the changes have to be implemented by means of custom coupled operations?* Custom coupled operations have to be implemented manually and thus increase the effort for model migration.
- **RQ4.** *Can COPE be applied to specify the complete coupled evolution of real-world metamodels, i.e. including all intermediate versions?* If COPE cannot be used to specify the complete coupled evolution, it could not have been used for forward engineering the coupled evolution.

### 7.1.2 Study Object

We chose two EMF-based metamodels that already have an extensive evolution history as study objects. We deliberately chose metamodels from completely different backgrounds in order to achieve more representative results.

**GMF Generator Model.** The first metamodel is developed as part of the open source project Graphical Modeling Framework (GMF)[1]. It is used to define generator models from which code for a graphical editor is generated. For our case study, we modeled the coupled evolution from release 1.0 over 2.0 to release 2.1, covering a period of 2 years.

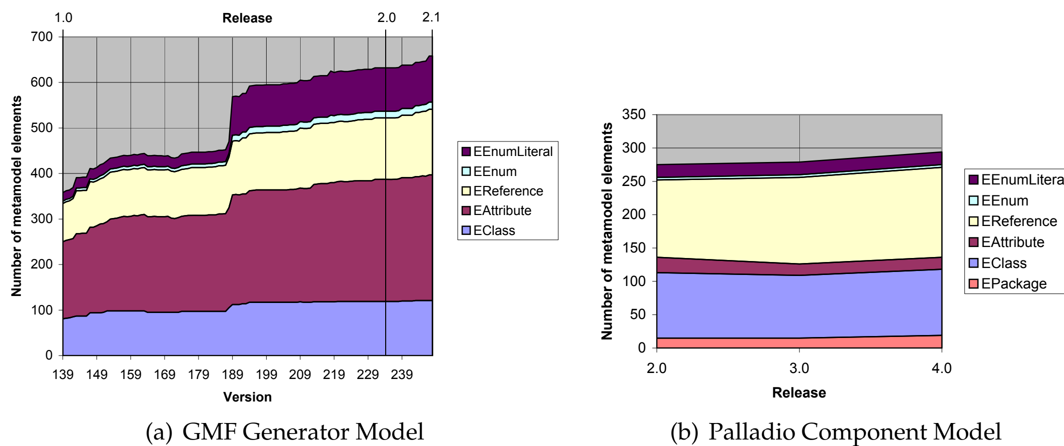
Figure 7.1(a) gives an impression of the size of the studied metamodel and its evolution over all the metamodel versions. In addition, the figure indicates the different releases of the metamodel. The metamodel is quite extensive, growing to more than a hundred classes in the course of its history.

There exist a significant number of models conforming to this metamodel, most of which are not under control of the language engineers. In order to be able to migrate these models, the language engineers have handcrafted a migrator with test cases that we used for validation.

**Palladio Component Model.** The second metamodel is developed as part of the research project Palladio Component Model (PCM)[2] and is used for the specification and analysis of component-based software architectures. For our case study, we modeled the coupled evolution from release 2.0 over 3.0 to release 4.0, covering a period of 1.5 years.

Figure 7.1(b) gives an impression of the size of the metamodel and its evolution over the studied releases. Similar to GMF, the PCM metamodel is quite extensive, being

[1]see GMF web site: `http://www.eclipse.org/modeling/gmp/`
[2]see PCM web site: `http://www.palladio-approach.net`




(a) GMF Generator Model (b) Palladio Component Model

**Figure 7.1:** Metamodel evolution of GMF and PCM in numbers

split up in a number of packages and defining more than a hundred classes throughout the history.

As the language engineers control all existing models—which are relatively few—they were not forced to handcraft a migrator until now, but manually migrated the models instead. Since no migrator could be used for validation, the modeled coupled evolution was validated by the language engineers of PCM.

### 7.1.3 Study Execution

The evolution of the metamodels was only available in the form of snapshots that depict the state of the metamodel at a particular point in time. Therefore, we had to infer both the metamodel adaptation as well as the corresponding model migration. We used the following procedure to reverse engineer the coupled evolution:

1. *Extraction of metamodel versions:* We extracted versions of the metamodel from the version control system.
2. *Comparison of consecutive metamodel versions:* Since the version control systems of both projects are snapshot-based, they do not contain a history of change operations between consecutive metamodel versions. To infer them, consecutive metamodel versions had to be compared to obtain a difference model. The difference model consists of a number of primitive changes between consecutive metamodel versions and was obtained using the tool EMF Compare[3].
3. *Generation of metamodel adaptation:* A first version of the history was obtained by generating a metamodel adaptation from the difference model between consecutive metamodel versions. For this purpose, a transformation was implemented that translates each of the primitive changes from the difference model to metamodel adaptation primitives specified in COPE.
4. *Detection of coupled operations:* The generated metamodel adaptation was refined by combining adaptation primitives to coupled operations based on the

[3]see EMF Compare web site: `http://wiki.eclipse.org/EMF_Compare`

information on how corresponding models are migrated. In doing so, we tried to map the compound changes to reusable coupled operations already available in the library. If not possible, we tried to identify and develop new reusable coupled operations. In case a certain model migration was too specific to be reused, it was realized as a custom coupled operation.

5. *Validation of the history:* The validity of the obtained coupled evolution was tested on both levels. The metamodel adaptation is easy to validate, because the history can be executed and the result can be compared to the metamodel snapshots. Test models before and after model migration were used to validate whether the model migration performs as intended.

Steps 1 to 3 and 5 are fully automated, whereas step 4 had to be performed manually. In addition, there is an iteration over steps 4 and 5, as a failed validation leads to corrections of the history. It took roughly one person week for each studied metamodel to reach the fix point during the iteration.

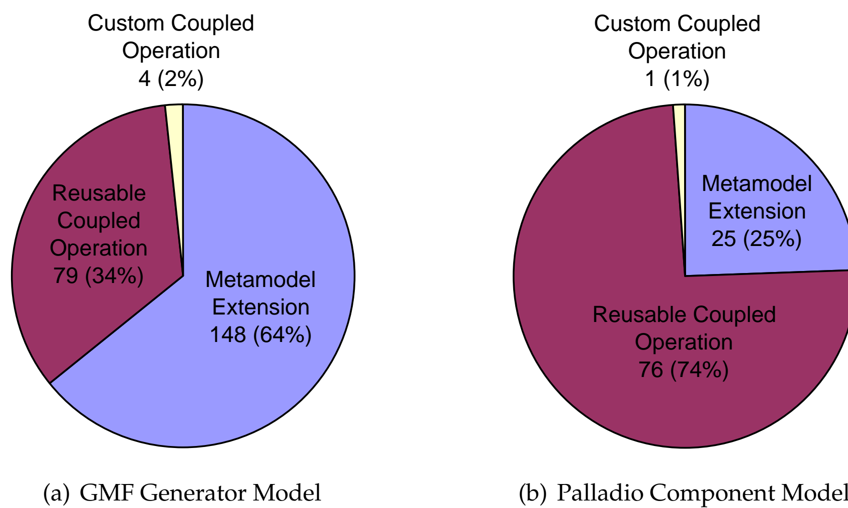
### 7.1.4 Study Result

**GMF Generator Model.** Figure 7.2(a) depicts the number of the different classes of metamodel adaptations that were used to model the coupled evolution with COPE. The metamodel extensions make up 64% of the adaptations, whereas reusable coupled operations account for 34%. Table 7.1 refines this classification by listing the names and the number of occurrences of the different kinds of metamodel adaptations. The dashed line separates the reusable coupled operations already available in the library from those which have been implemented while conducting the case study. For the GMF metamodel, these new reusable coupled operations cover 5 of 79 occurrences (6% of the applications of reusable coupled operations). The remaining 2% of the metamodel adaptations consist of only 4 custom coupled operations for which the model migration had to be implemented manually. The model migration code handcrafted for these custom coupled operations amounts to 103 lines of code.

The GMF engineers employ a systematic change management process, as they do not have all the models under control: the language engineers discuss metamodel adaptations and their impact on models thoroughly before actually carrying them out. Consequently, we found no destructive change at any instant in the history, which was reversed at a later instant. Thus, the obtained language history comprises all the intermediate versions.

As the GMF engineers do not have all the models under their control, they have manually implemented a migrator. This migrator constitutes a very technical solution and is based on different mechanisms for the two stages. For the migration from release 1.0 to 2.0, the migrator patches the model while deserializing its XML representation. For the migration from release 2.0 to 2.1, a generic copy mechanism is used that first filters out non-conforming parts of the model and later rebuilds them. Even though this migrator is very optimized, it is difficult to understand and maintain due the low abstraction level of its implementation.

**Palladio Component Model.** Figure 7.2(b) depicts the number of the different classes




(a) GMF Generator Model (b) Palladio Component Model

**Figure 7.2:** Classification of the language changes

of metamodel adaptations that were used to model the coupled evolution. Here, the metamodel extensions account for only 25% of the metamodel adaptations, whereas reusable coupled operations make up 74%. Again, Table 7.1 provides more detailed results. The reusable coupled operations that were implemented while conducting the case study cover 1 out of 76 occurrences (1% of the applications of reusable coupled operations). The remaining 1% of the metamodel adaptations consist of 1 custom coupled operation for which the model migration had to be implemented manually. The model migration code handcrafted for this custom coupled operation amounts to only 10 lines of code.

It seems as if the PCM engineers have not taken the impact on the models into account, as they have all the models under their control. Consequently, there were a lot of destructive changes between the intermediate versions that were reversed at a later instant. Therefore, the obtained language history comprises only the release versions.

As the language engineers have not yet provided tool support for model migration, our approach helped by providing an automatic migrator. However, they provided us with test models and helped to validate the obtained model migration.

### 7.1.5 Study Discussion

**RQ1.** *Which fraction of the changes are metamodel extensions that do trivially not require a migration of models?* The fraction of metamodel extensions is very large (64%) for the GMF metamodel, whereas it is rather small (25%) for the PCM metamodel. A possible interpretation is that the GMF engineers as far as possible avoided metamodel adaptations that required to enhance the migrator. The reason for the metamodel extensions could as well be the nature of the evolution: they were adding new generator features to the language which are orthogonal to existing ones, i.e. cannot expressed with existing features.

**Table 7.1:** Detailed results

| | GMF | | | PCM | | | Overall |
|---|---|---|---|---|---|---|---|
| **Operation Name** | 1.0 - 2.0 | 2.0 - 2.1 | Overall | 2.0 - 3.0 | 3.0 - 4.0 | Overall | |
| **Metamodel Extension** | **136** | **12** | **148** | **9** | **16** | **25** | **173** |
| Add Supertype | | | | 1 | 2 | 3 | 3 |
| Create Attribute | 63 | 6 | 69 | | 1 | 1 | 70 |
| Create Class | 36 | 1 | 37 | 3 | 4 | 7 | 44 |
| Create Enumeration | 12 | 1 | 13 | | 4 | 4 | 17 |
| Create Package | | | | | 4 | 4 | 4 |
| Create Reference | 25 | 4 | 29 | 5 | 1 | 6 | 35 |
| **Reusable Coupled Operation** | **76** | **3** | **79** | **44** | **32** | **76** | **155** |
| Change Attribute Type | | | | 2 | 1 | 3 | 3 |
| Change Package | | | | 1 | 4 | 5 | 5 |
| Collect Feature over References | 4 | | 4 | | | | 4 |
| Delete Class | | | | 1 | | 1 | 1 |
| Delete Feature | 14 | | 14 | 4 | 2 | 6 | 20 |
| Extract Class | 1 | | 1 | | | | 1 |
| Extract Superclass | 5 | | 5 | | | | 5 |
| Fold Class | | | | 2 | | 2 | 2 |
| Generalize Reference | 5 | | 5 | 2 | 2 | 4 | 9 |
| Generalize Supertype | | | | 1 | 2 | 3 | 3 |
| Inheritance to Delegation | 1 | 1 | 2 | 4 | | 4 | 6 |
| Inline Superclass | 2 | | 2 | | | | 2 |
| Merge Classes | 2 | | 2 | 7 | | 7 | 9 |
| Merge Enumerations | 2 | | 2 | | | | 2 |
| Merge Literal | 1 | | 1 | | | | 1 |
| Move Feature over Reference | 2 | 1 | 3 | | | | 3 |
| Pull up Feature | 3 | | 3 | | | | 3 |
| Push down Feature | 1 | | 1 | | | | 1 |
| Remove Supertype | 1 | | 1 | | 1 | 1 | 2 |
| Rename | 27 | 1 | 28 | 16 | 18 | 34 | 62 |
| Specialize Supertype | | | | 3 | 1 | 4 | 4 |
| Specialize Composite Reference | | | | 1 | | 1 | 1 |
| Copy Feature | 1 | | 1 | | | | 1 |
| Extract and Group Attribute | 1 | | 1 | | | | 1 |
| Flatten Composition Hierarchy | 1 | | 1 | | | | 1 |
| Remove Superfluous Super Type | | | | | 1 | 1 | 1 |
| Pull Feature over References | 1 | | 1 | | | | 1 |
| Push Feature over References | 1 | | 1 | | | | 1 |
| **Custom Coupled Operation** | **2** | **2** | **4** | **1** | | **1** | **5** |

**RQ2.** *Which fraction of the changes can be reused by means of reusable coupled operations?* For both metamodels, a large (34% and 74%) fraction of changes can be dealt with by reusable coupled operations—aside from the metamodel extensions. This result strengthens the findings from the previous study presented in Chapter 3 (*State of the Practice: Automatability of Model Migration*) that a lot of migration effort can be saved by reuse in practice. Besides the reusable coupled operations presented in Section 5.2 (*Library of Reusable Coupled Operations*), we have also identified a small number of new reusable coupled operations.

**RQ3.** *Which fraction of the changes have to be implemented by means of custom coupled operations?* For both metamodels, a very small (1% and 2%) fraction of changes were so specific that they had to be modeled as custom coupled operations. To manually implement these custom coupled operations, the language was sufficiently expressive and the effort was feasible (103 and 10 lines of code). This result also strengthens

the findings from the previous study that a non-negligible number of changes are specific to the metamodel.

**RQ4.** *Can COPE be applied to specify the complete coupled evolution of real-world metamodels?* The case studies showed that COPE can be applied to specify the coupled evolution of real-world metamodels.

In case of the GMF metamodel, we would even have been able to directly use COPE for its maintenance. As the GMF developers do not control the numerous existing models, they took also the impact on the models into account while adapting the metamodel. COPE can help here to perform more profound metamodel adaptations.

In case of the PCM metamodel, we would not have been able to directly use COPE for its maintenance, as there were a lot of destructive changes that were reversed at a later instant. For metamodel adaptation, the PCM engineers preferred flexibility over preservation of existing models, as they have the few existing models under control. COPE can help here to perform the metamodel adaptations in a more systematic way by using reusable coupled operations.

Summing up, COPE provides a compromise between the two studied types of metamodel histories: it provides more flexibility for carrying out metamodel adaptations and offers at the same time a more systematic approach for metamodel adaptation.

**Lessons Learned.** The destructive changes in the PCM evolution that were reversed at a later instant motivated the implementation of the function to undo changes as presented in Section 6.2.2 (*Refactoring the Coupled Evolution*). This function allows language engineers to undo these changes and thus the required destructive migration, and thereby allows them to deal with the destructive changes in a clean manner.

The coupled evolution was reverse engineered on the level of the coupled evolution language as presented in Chapter 5 (*COPE – Coupled Evolution of Metamodels and Models*). Changing the coupled evolution on this level always bears the risk that the metamodel adaptation does no longer comprise the intermediate metamodel versions. Consequently, we spent a lot of effort to ensure that the history comprised the intermediate metamodel versions. We implemented the convergence view—as presented in Section 6.2.3 (*Recovering the Coupled Evolution*)—to avoid this effort for recovering the coupled evolution in future cases. The convergence view constructively ensures that the recorded history comprises the intermediate metamodel versions.

### 7.1.6 Threats to Validity

We present the threats according to the steps of the method to which they apply together with the measures that we took to mitigate them:

1. *Extraction of metamodel versions:* We assumed that a commit indicates a new version of the metamodel. Therefore, we considered to group only primitive changes from one commit to the next. However, metamodels were sometimes committed in a premature version, and hence we might miss complex changes that span several commits. Whereas we have not identified changes that span several commits in case of GMF, the PCM engineers often committed the meta-

model in a premature version. To mitigate the risk of missing those changes, we only considered the release versions of the PCM metamodel.

2. *Comparison of consecutive metamodel versions:* In the absence of unique and persistent element identifiers, the comparison cannot always be performed unambiguously [Robbes and Lanza, 2007]. Moreover, the comparison cannot recover changes that have been overwritten by others. Whereas this threat is mitigated by the low number of changes in commits to the GMF metamodel, the number is rather large in case of PCM due to choosing the release versions. To also reduce the risk of ambiguity errors for PCM, the reverse engineered history was reviewed by the language engineers.

3. *Generation of metamodel adaptation:* The difference model—which is an unordered set of changes—was transformed to the metamodel adaptation—which is an ordered sequence of changes. From the possible orders of this set, the transformation thus had to choose one. However, the chosen order might not reflect how the changes have been carried out by the language engineers. To mitigate this threat, the changes were reordered to be able to group related changes to coupled operations, and the coupled operations were validated.

4. *Detection of coupled operations:* It may seem odd that new reusable coupled operations could be used for one metamodel. Consequently, we cannot be sure that they are useful in other scenarios, threatening their classification as reusable coupled operations. However, two case studies may not suffice to show their usefulness in other scenarios. In addition, it may depend on the habits of the language engineer which reusable coupled operations are often used and which not. The extension mechanism of COPE allows the language engineer to easily register new reusable coupled operations which fit their habits.

5. *Validation of the history:* For the PCM metamodel, we had no reference migrator that we could use to validate the correctness of the migration defined by the history. As a consequence, the reverse engineered migration might not be the one intended by the PCM engineers. To mitigate this threat, the language engineers provided us with some test models and manually reviewed the reverse engineered history.

## 7.2 Graphical Modeling Framework

In the last section, we analyzed the evolution of the GMF Generator metamodel which is one of the metamodels of GMF. As the evolution is well-documented and supported by a migrator, we decided to extend it to the other metmodels of GMF. We not only examined the model migration, but also the impact on other language artifacts and the reasons for language evolution.

### 7.2.1 Study Goal

In this section, we investigate the evolution of modeling languages by reverse engineering the evolution of their metamodels and the migration of related language artifacts, like e.g. the models or the code generator. Our motivation is to identify

requirements for tools that support the (semi-)automatic coupled evolution of modeling languages and related artifacts in a way that avoids the language erosion and minimizes the handwritten code for migration. We focus on the following research questions:

- **RQ1.** *What is the impact of language changes on related language artifacts?* As the metamodel is in the center of the language definition, we are interested to understand how other language artifacts change, when the metamodel changes.
- **RQ2.** *What are the reasons for language changes?* We investigate the distribution of the maintenance activities performed to implement metamodel changes in order to examine the similarities between the evolution of programs and the evolution of languages.
- **RQ3.** *What kinds of operations capture the language changes?* We are interested to describe the metamodel changes based on a set of canonical operations, and thereby to investigate the measure in which these operations can be used to migrate the models.

### 7.2.2 Study Object

The Graphical Modeling Framework (GMF)[4] is a widely used open source framework for the model-based development of diagram editors. GMF is a prime example for a Model-Driven Architecture (MDA) [Kleppe et al., 2003], as it strictly separates platform-independent models (PIM), platform-specific models (PSM) and code. GMF is implemented on top of the Eclipse Modeling Framework (EMF)[5] and the Graphical Editing Framework (GEF)[6].

**Editor Models.** In GMF, a diagram editor is defined by models from which editor code can be generated automatically. For this purpose, GMF provides four modeling languages, a transformer that maps PIMs to PSMs, a code generator that turns PSMs into code, and a runtime platform on which the generated code relies.

The lower part of Figure 7.3 illustrates the different kinds of models of which a GMF Application consists. On the platform-independent level, a diagram editor is modeled from four different views. The domain model defines the abstract syntax of diagrams. The graphical definition model defines the graphical elements like nodes and edges in the diagram. The tool definition model defines the tools available to author a diagram. In the mapping model, the first three views are combined to an overall view which maps the graphical elements from the graphical definition model and the tools from the tool definition model onto the elements from the domain model.

The platform-independent mapping model is transformed into a platform-specific diagram generator model. This model can be altered to customize the code generation.

**Modeling Languages.** We can distinguish two kinds of languages involved in GMF.

[4]see GMF web site: `http://www.eclipse.org/modeling/gmf`
[5]see EMF web site: `http://www.eclipse.org/modeling/emf`
[6]see GEF web site: `http://www.eclipse.org/gef`

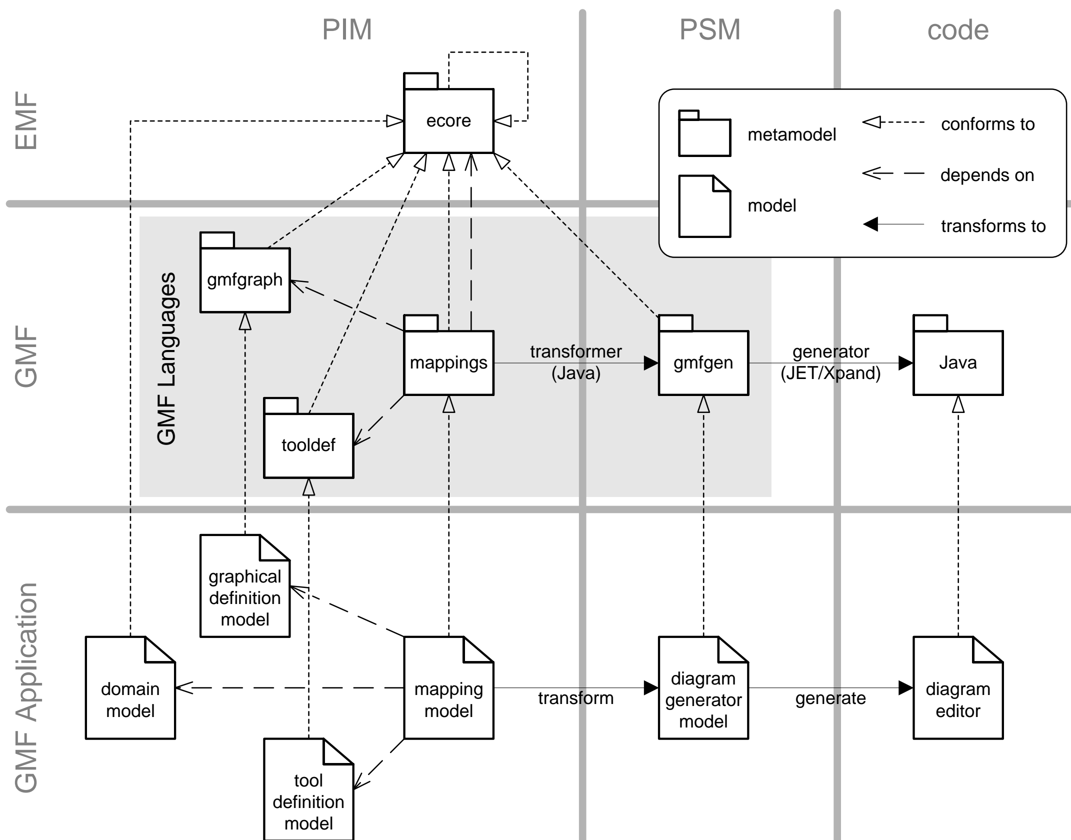



**Figure 7.3:** Languages involved in the Graphical Modeling Framework

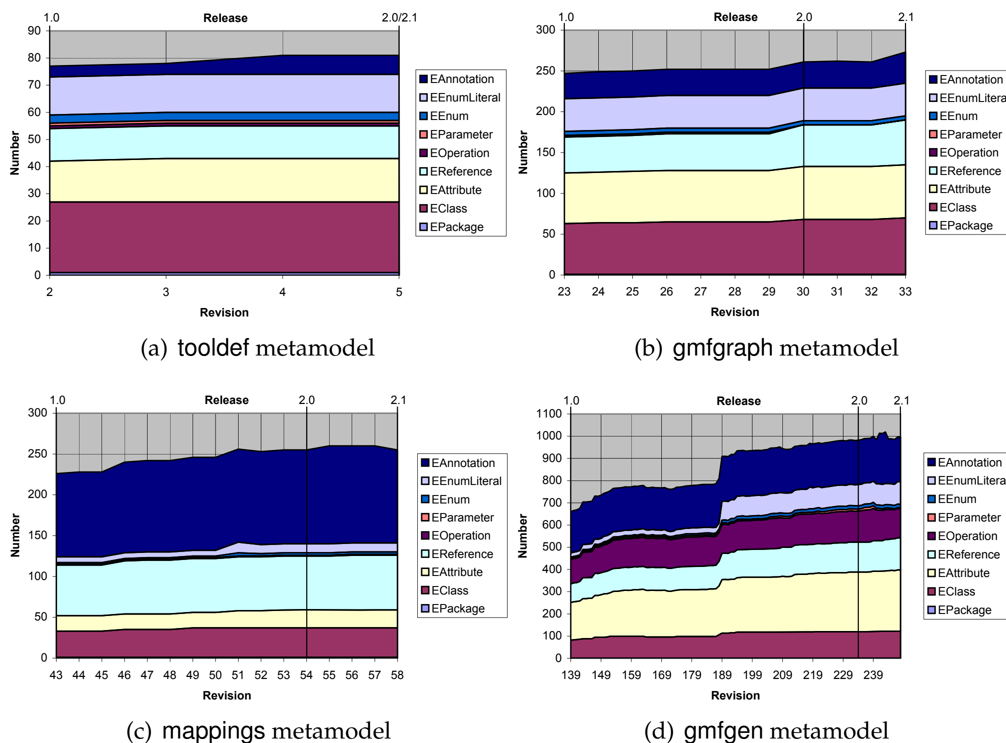
First, GMF provides domain-specific languages for the modeling of diagram editors. Each of these languages comes with a metamodel defining its abstract syntax and a simple tree-based model editor integrated in Eclipse. The upper part of Figure 7.3 shows the metamodels involved in GMF. These are ecore for domain models, gmfgraph for graphical definition models, tooldef for tool definition models, mappings for mapping models, and gmfgen for diagram generator models. The mappings metamodel refers to elements in the ecore, gmfgraph, and tooldef metamodels. This kind of dependency is typical for multi-view modeling languages. For example, there are similar dependencies between the metamodel packages defining the various sublanguages of the UML.

Second, GMF itself is implemented in various languages. All metamodels are expressed in ecore, the metamodeling language provided by EMF. Additionally, the metamodels contain context constraints which are attached as textual annotations to the metamodel elements to which they apply. These constraints are expressed in the Object Constraint Language (OCL) [Object Management Group, 2006b]. The transformer from a mapping model to a generator model is implemented in Java. For model access, it relies on the APIs generated from the metamodels of the GMF modeling languages. The generator generates code from the diagram generator model. It was formerly implemented in Java Emitter Templates (JET)[7], which was

[7]see JET web site: `http://www.eclipse.org/modeling/m2t`

later changed in favor of Xpand[8]. The generated code conforms to the Java programming language, and is based on the GMF runtime platform.

**Metamodel Evolution.** With a code base of more than 600k lines of code, GMF is a framework of large size. GMF is implemented by 13 language engineers from 3 different countries using an agile process with small development cycles. Since starting the project, the GMF engineers had to often adapt the metamodels. As many metamodel changes broke the existing models, the language engineers had to manually implement a migrator. Figure 7.4 illustrates the metamodel evolution for the two release cycles we studied, each taking one year. The figures show the number of metamodel elements for each revision of each GMF metamodel. During the evolution from release 1.0 to 2.1, the number of classes defined by all metamodels e.g. increased from 201 to 252.




(a) tooldef metamodel (b) gmfgraph metamodel

(c) mappings metamodel (d) gmfgen metamodel

**Figure 7.4:** Statistics of GMF metamodel evolution

We chose GMF as a case study, because the evolution is extensive, publicly available, and well documented by means of commit comments and change requests. However, the evolution is only available in the form of revisions from the version control system, and its documentation is only informal.

### 7.2.3 Study Execution

Due to the large size of the GMF metamodels, we developed a systematic approach to investigate its evolution as presented in the following.

[8]see Xpand web site: `http://www.openarchitectureware.org`

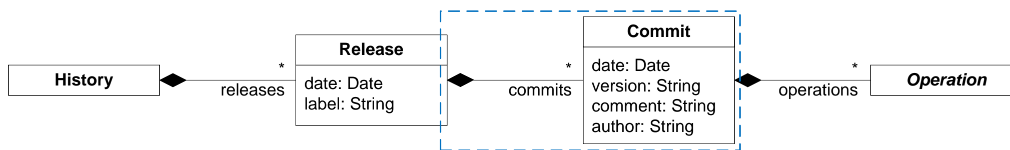



**Figure 7.5:** Modeling language history

### Modeling the History

To investigate the evolution of modeling languages, we model the history of their metamodels using the versioning metamodel presented in Section 6.3.1 (*History Metamodel*). Figure 7.5 shows how we extended this versioning metamodel for the case study. A Release is now further subdivided into a number of commits. A Commit denotes a version of the modeling language which has been committed to the version control system. Modeling languages are committed at a date, by an author, with a comment, and are tagged with a version number. A Commit comprises the sequence of operations which have been performed since the last Commit.

### Classification of Operations

We model the evolution of the metamodel by operations for stepwise metamodel adaptation. Figure 7.6 shows the classification of metamodel adaptation operations according to the following four different criteria:

**Granularity.** Similar to [Lämmel, 2001] and Section 5.2 (*Library of Reusable Coupled Operations*), we distinguish primitive and composite operations. A PrimitiveOperation supports a metamodel adaptation that can not be decomposed into smaller operations. In contrast, a CompositeOperation can be decomposed into a sequence of PrimitiveOperations. The required kinds of PrimitiveOperations can be derived from the metametamodel. There are two basic kinds of primitive changes: StructuralPrimitives and NonStructuralPrimitives. A StructuralPrimitive modifies the structure of a metamodel, i.e. creates or deletes a metamodel element. A NonStructuralPrimitive modifies an existing metamodel element, i.e. changes a feature of a metamodel element. The set of primitive operations are already complete in the sense that every metamodel adaptation can be described by composing them.

**Metamodel Aspects.** We classify an operation according to the metamodel aspect which it addresses. The different classes can be derived from the constructs provided by the metametamodel. An operation concerns either the structure of models, constraints on models, the API to access models, or the documentation of metamodel elements. A SyntaxOperation like Extract Superclass affects the abstract syntax defined by the metamodel. A ConstraintOperation adds, deletes, moves, or changes constraints in the metamodel. An APIOperation concerns the additional access methods defined in the metamodel. This includes derived features and operations. A DocumentationOperation adds, deletes, moves, or changes documentation annotations to metamodel elements.

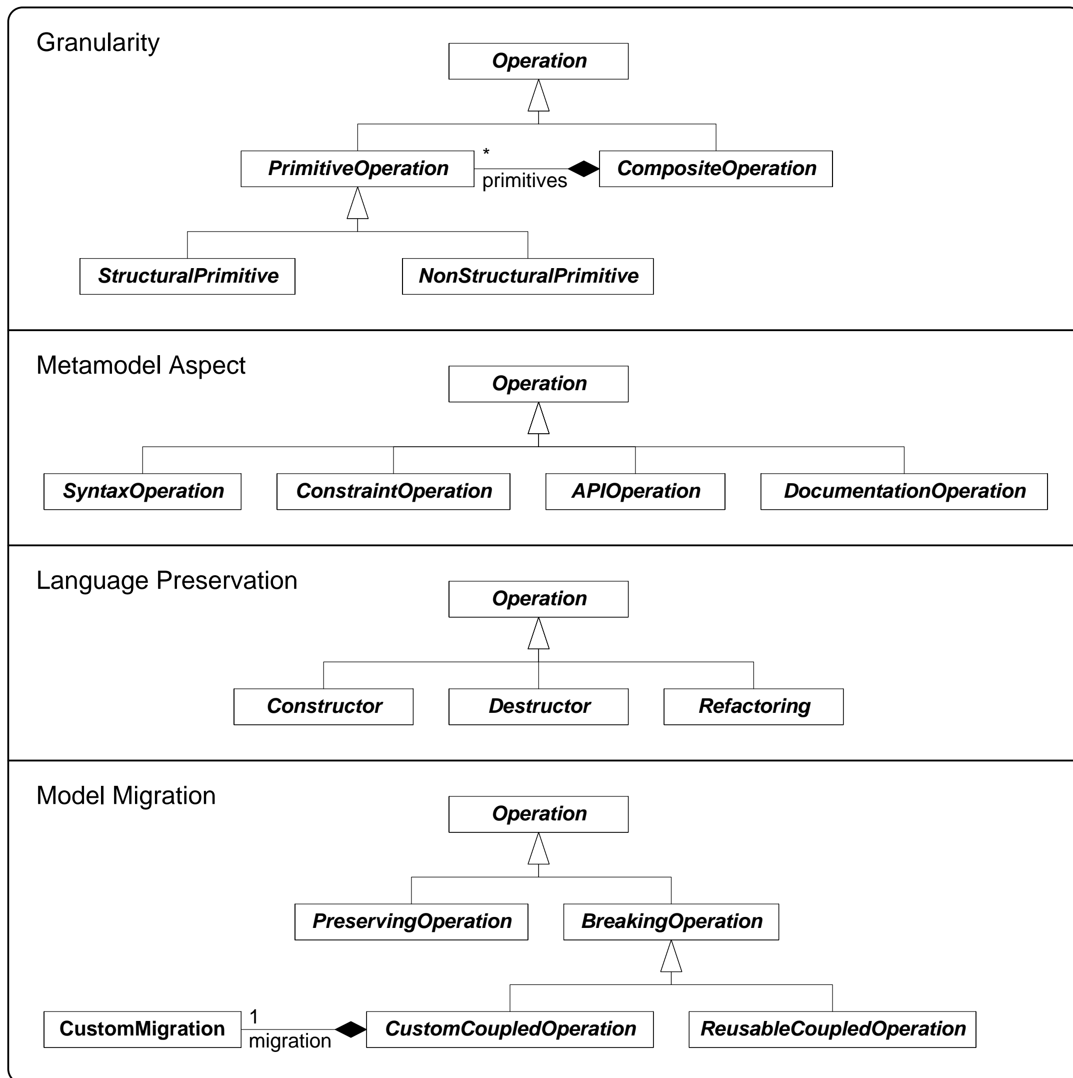



**Figure 7.6:** Classification of operations for metamodel adaptation

**Language Preservation.** According to [Wachsmuth, 2007], we can distinguish three kinds of operations with respect to preservation of the modeling language's expressiveness. By expressiveness of a modeling language, we refer to the set of syntactically valid models we can express in the modeling language. Constructors increase this set, i.e. in the new version of the language we can express new models. In contrast, Destructors decrease the set, i.e. in the old version we could express models which we cannot express in the new version of the language. Finally, Refactorings preserve the set of valid models, i.e. we can express all models in the old and the new version of the language.

**Model Migration.** As demonstrated in Chapter 3 (*State of the Practice: Automatability of Model Migration*), we can determine for each operation to what extent model migration can be automated. PreservingOperations do not require the migration of models. BreakingOperations break the instance relationship between models and the adapted metamodel. In this case, we need to provide a migration for existing

models. For a ReusableCoupledOperation, the migration does not depend on a specific metamodel. Thus it can be specified as a generic couple of metamodel adaptation and model migration. In contrast, a CustomCoupledOperation is so specific to a certain metamodel that it cannot be composed of reusable coupled operations. Consequently, it can only be covered by a sequence of adaptation steps and a reconciling CustomMigration.

The presented criteria are orthogonal to each other to a large extent. Granularity is orthogonal to all other criteria and vice versa, as we can think of example operations from each granularity for all these criteria. Additionally, language expressiveness and model migration are orthogonal to each other: the first concerns the difference in cardinality between the sets of valid models before and after adaptation, whereas the second concerns the correct migration of a model from one set to the other. However, language preservation and model migration both focus on the impact on models, and are thus only orthogonal to the metamodel aspects StructuralAdaptation and ConstraintAdaptation. This is because operations concerning APIOperation and DocumentationOperation do not affect models. Consequently, these operations are always Refactorings and PreservingOperations.

The set of operations necessary for our case study is depicted in Table 7.4. We classify each operation according to the categories presented before. For example, the operation Extract Superclass creates a new common superclass for a number of classes. This operation is a CompositeOperation, since we can express the same metamodel adaptation by the primitive operations Create Class, Add Superclass and Move Feature. The operation is a SyntaxOperation, since it affects the abstract syntax defined by the metamodel. It is a Constructor, because we can instantiate the introduced superclass in the new language version. Finally, it is a PreservingOperation, since no migration of old models to the new language version is required.

### Reverse Engineering the GMF History

We reverse engineered the GMF history following a defined procedure and using a number of tools.

**Procedure.** We applied the following steps to reconstruct a history model for GMF based on the available information:

1. *Extracting the log:* We extracted the log for the whole GMF repository. The log lists the revisions of each file maintained in the repository.
2. *Detecting the commits:* Since the versioning system used to develop GMF does not store which files were committed together, we grouped revisions to commits using a sliding window approach [Zimmermann et al., 2005]. Two revisions of different files were grouped, in case they were committed within the same time interval and with the same commit comment.
3. *Filtering the commits:* We filtered out all commits which do not include a revision of one of the metamodels.
4. *Clustering the revisions:* We clustered the files which were committed together

into more abstract language artifacts like metamodels, transformer, code generator, and migrator. This step was performed to reduce the information, as the implementation of each of the language artifacts may be modularized into several files. The commits to the language artifacts are used to answer *RQ1*.

5. *Classifying the commits:* We classified the commits according to the software maintenance categories (i.e. perfective, adaptive, preventive, and corrective) [Lientz and Swanson, 1980] based on the commit comments and change requests. The classification of the commits is used to answer *RQ2*.
6. *Extracting the metamodel revisions:* We extracted the metamodel revisions of the commits from the GMF repository.
7. *Comparing the metamodel revisions:* We compared consecutive metamodel revisions with each other resulting in a difference model for each pair of consecutive metamodel revisions. The difference model consists of a number of primitive changes between consecutive metamodel revisions.
8. *Detecting the operation sequence:* We detected the operations necessary to bridge the difference between the metamodel revisions. In contrast to the difference model, the operations also compose related primitive changes and are ordered as a sequence. To find the most plausible operations, we also analyzed commit comments, change requests, and the adaptation of the other language artifacts. The detected operation sequence is used to answer *RQ3*.
9. *Validating the operation sequence:* We validated the resulting operation sequence by applying it to migrate the existing models for testing the handcrafted migrator. We set up a number of test cases, each of which consists of a model before migration and the expected model after migration.

**Tool Support.** We employed a number of tools to perform the study. We employed statCVS[9] to parse the log into a model which is processed further by a handcrafted model transformation (steps 1-4). With the help of EMF Compare[10], we generated the difference models between two consecutive metamodel revisions (step 7). To bridge the difference between consecutive metamodel revisions, we employed the function of COPE to recover the coupled evolution that is explained in Section 6.2.3 (*Recovering the Coupled Evolution*) (step 8). From the recorded history model, we generated a migrator that was employed for validating the operation sequence (step 9). To generate the expected models for validation, we used the handcrafted migrator that comes with GMF.

### 7.2.4 Study Result

In this subsection, we present the aggregated results of our case study. The complete history can be obtained from our web site[11].

**RQ1.** *What is the impact of language changes on related language artifacts?* To answer

[9]see statCVS web site: `http://statcvs.sourceforge.net`
[10]see EMF Compare web site: `http://www.eclipse.org/emft/projects/compare`
[11]see COPE web site: `http://cope.in.tum.de/pmwiki.php?n=Documentation.GMF`

this question, we determined for each commit which other language artifacts were committed together with the metamodels. Table 7.2 shows how many of the overall 124 commits had an impact on certain language artifacts.

**Table 7.2:** Correlation between commits of metamodels and related language artifacts

| # | Metamodels | | | | | Transfor-mator | Genera-tor | Migrator |
|---|---|---|---|---|---|---|---|---|
| | gmfgraph | tooldef | mappings | gmfgen | commits | | | |
| 1 | ■ | | | | 7 | 3 | 3 | |
| 2 | | ■ | | | 2 | 1 | | |
| 3 | | | ■ | | 5 | 3 | | 2 |
| 4 | | | | ■ | 100 | 23 | 67 | 9 |
| 5 | | ■ | ■ | | 1 | | | 1 |
| 6 | | | ■ | ■ | 6 | 4 | 2 | 1 |
| 7 | ■ | | ■ | ■ | 3 | 1 | 1 | |
| | **10** | **3** | **15** | **109** | **124** | **35** | **73** | **13** |

The first four columns denote the metamodels that were changed in a commit, and the fifth column denotes the number of commits. For instance, row 6 means that the metamodels mappings and gmfgen changed together in 6 commits. The last three columns denote the number of commits in which other language artifacts, like transformer, code generator and migrator, were changed. For instance, in row 6, the transformer was changed 4 times, the generator 2 times, and the migrator had to be changed once.

In a nutshell, metamodel changes are very likely to impact language artifacts which are directly related to them, as can be seen in Table 7.2. For instance, the changes to mappings and gmfgen propagated to the transformer from mappings to gmfgen, and to the generator from gmfgen to code. Additionally, metamodel changes are not always carried out on a single metamodel, but are sometimes related to other metamodels.

**RQ2.** *What are the reasons for language changes?* To answer this question, we classified the commits into the categories of maintenance activities as explained in Section 2.5.1 (*Reasons for Language Evolution*) and investigated their distribution over these categories. Figure 7.7 illustrates the number of commits for each category, and Table 7.3 shows a more detailed classification of the commits. Note that several commits could not be uniquely associated to one category and thus had to be assigned to several categories. However, all commits could be classified into at least one of the four categories.

**Table 7.3:** Classification of metamodel commits according to maintenance categories

| **Perfective** | **45** | **Adaptive** | **33** | **Preventive** | **36** | **Corrective** | **16** |
|---|---|---|---|---|---|---|---|
| Model navigator | 13 | Transition to Xpand | 25 | Separation | 16 | Bug report | 7 |
| Rich client platform | 6 | Ecore constraints | 5 | Simplification | 10 | Rename | 3 |
| Diagram preferences | 4 | Namespace URI | 2 | Unused elements | 8 | Revert changes | 3 |
| Diagram partitioning | 2 | OCL parser | 1 | Documentation | 2 | Wrong constraint | 3 |
| Element properties | 2 | | | | | | |
| Individual features | 18 | | | | | | |

We classified 45 of the commits as perfective maintenance, i.e. add new features to enhance GMF. Besides a number of individual commits, there are a few features whose

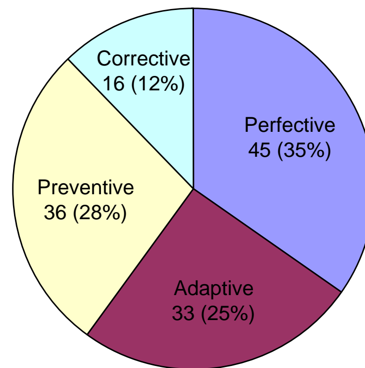



**Figure 7.7:** Share of maintenance categories

introduction spanned several commits. The generated diagram editor was extended with a model navigator, to run as a rich client, to set preferences for diagrams, to partition diagrams, and to set properties of diagram elements. We classified 33 of the commits as adaptive maintenance, i.e. adapt GMF to a changing environment. These commits were either due to the transition from JET to Xpand, adapted to changes to the constraints of ecore, were due to releasing GMF, or adapted the constraints to changes of the OCL parser. We classified 36 of the commits as preventive maintenance, i.e. refactor GMF to prevent faults in the future. These commits either separated concerns to better modularize the generated code, simplified the metamodels to make the transformations more straightforward, removed metamodel elements no longer used by transformations, or added documentation to make the metamodel more understandable. We classified 16 of the commits as corrective maintenance, i.e. correct faults discovered in GMF. These commits either fixed bugs reported by GMF users, corrected incorrectly spelled element names, reverted changes carried out earlier, or corrected invalid OCL constraints.

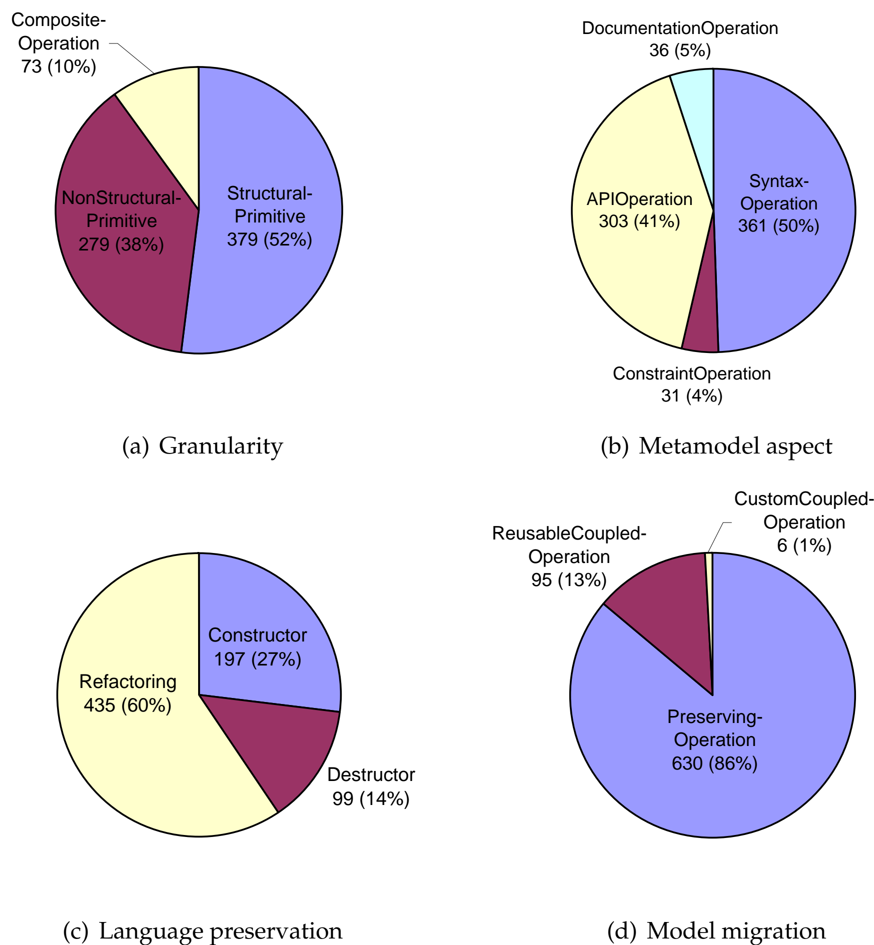
In a nutshell, the typical activities known from software maintenance also apply to metamodel maintenance [Lientz and Swanson, 1980]. Furthermore, similar to the development of software, the number of perfective activities (35%) outranges the preventive (28%) and adaptive (25%) activities which are double the number of corrective activities (12%).

**RQ3.** *What kinds of operations capture the language changes?* To answer this question, we classified the operations which describe the metamodel evolution. Figure 7.8 illustrates the classification of the operations along the different criteria. Table 7.4 shows the number and classification of each operation occurred during the evolution of each metamodel. The operations are grouped by their granularity and the metamodel aspects to which they apply.

Most of the changes could be covered by PrimitiveOperations: we found 379 (52%) StructuralPrimitives, 279 (38%) NonStructuralPrimitives and 73 (10%) CompositeOperations. Only half of the operations affected the structure defined by a metamodel: we identified 361 (50%) SyntaxOperations, 303 (41%) APIOperations, 36 (5%) DocumentationOperations, and 31 (4%) ConstraintOperations. Most of the changes are refactorings which do not change the expressiveness of the modeling language: we found 435 (60%) Refactorings, 197 (27%) Constructors, and 99 (14%) Destructors. Only very

**Table 7.4:** Classification of operations occurred during metamodel adaptation

| Classification: Granularity | Classification: Metamodel Aspect | Operation | Classification: Language Preservation | Classification: Model Migration | Number of Applications: gmfgraph | tooldef | gmfmap | gmfgen | all |
|---|---|---|---|---|---|---|---|---|---|
| Structural-Primitive | Syntax | Create Class | Constructor | Preserving | 4 | | 2 | 37 | 43 |
| | | Create Enumeration | Constructor | Preserving | | | 3 | 13 | 16 |
| | | Create Opposite Reference | Destructor | Preserving | | | | 14 | 14 |
| | | Create Optional Attribute | Constructor | Preserving | 1 | 1 | 3 | 63 | 68 |
| | | Create Optional Reference | Constructor | Preserving | 2 | | 3 | 14 | 19 |
| | | Create Required Attribute | Constructor | Custom | | | | 1 | 1 |
| | | Create Required Reference | Constructor | Custom | | | | 1 | 1 |
| | | Delete Feature | Destructor | Reusable | 4 | | | 14 | 18 |
| | | Merge Literal | Destructor | Reusable | | | | 1 | 1 |
| | Constraint | Create Constraint Annotation | Destructor | Preserving | | | 5 | 3 | 8 |
| | | Delete Constraint Annotation | Constructor | Preserving | | | 2 | 1 | 3 |
| | API | Create Deprecated Annotation | Refactoring | Preserving | | | | 3 | 3 |
| | | Create Setter Visibility Annotation | Refactoring | Preserving | 1 | | 5 | 30 | 36 |
| | | Create Volatile Attribute | Refactoring | Preserving | 1 | | | 4 | 5 |
| | | Create Operation | Refactoring | Preserving | | | | 53 | 53 |
| | | Create Volatile Reference | Refactoring | Preserving | | | | 1 | 1 |
| | | Delete Setter Visibility Annotation | Refactoring | Preserving | 1 | | 5 | 31 | 37 |
| | | Delete Operation | Refactoring | Preserving | 1 | | | 34 | 35 |
| | Documentation | Create Documentation Annotation | Refactoring | Preserving | 3 | 3 | 2 | 8 | 16 |
| | | Delete Documentation Annotation | Refactoring | Preserving | | | | 2 | 2 |
| NonStructural-Primitive | Syntax | Abstract Class to Interface | Refactoring | Preserving | | | 1 | 2 | 3 |
| | | Add Supertype | Constructor | Preserving | 10 | | | | 10 |
| | | Change Attribute Type | Refactoring | Reusable | | 1 | | | 1 |
| | | Drop Attribute Identifier | Constructor | Preserving | 1 | | | | 1 |
| | | Drop Class Abstract | Constructor | Preserving | | | | 1 | 1 |
| | | Drop Class Interface | Constructor | Preserving | | | 1 | 2 | 3 |
| | | Drop Reference Opposite | Constructor | Preserving | | | | 1 | 1 |
| | | Make Class Abstract when Interface | Refactoring | Preserving | 14 | 4 | 9 | 22 | 49 |
| | | Make Class Interface when Abstract | Refactoring | Preserving | | | | 2 | 2 |
| | | Make Feature Required | Destructor | Custom | 2 | | | | 2 |
| | | Make Feature Volatile | Destructor | Reusable | | | 1 | 5 | 6 |
| | | Make Reference Composite | Constructor | Reusable | 1 | | | | 1 |
| | | Remove Supertype | Destructor | Reusable | 10 | | | 1 | 11 |
| | | Rename Class | Refactoring | Reusable | | | | 1 | 1 |
| | | Rename Feature | Refactoring | Reusable | | | | 10 | 10 |
| | | Rename Literal | Refactoring | Reusable | | | | 7 | 7 |
| | | Set Package Namespace URI | Refactoring | Reusable | 1 | | 3 | 3 | 7 |
| | | Specialize Reference Type | Refactoring | Preserving | 4 | | | | 4 |
| | Constraint | Modify Constraint Annotation | Destructor | Preserving | | | 8 | 9 | 17 |
| | API | Rename Volatile Feature | Refactoring | Preserving | | | | 2 | 2 |
| | | Rename Operation | Refactoring | Preserving | | | | 9 | 9 |
| | | Set Feature Changeable | Refactoring | Preserving | 2 | | 11 | 63 | 76 |
| | | Set Reference Resolve Proxies | Refactoring | Preserving | 1 | | 5 | 31 | 37 |
| | Documentation | Modify Documentation Annotation | Refactoring | Preserving | 2 | | 7 | 9 | 18 |
| Composite-Operation | Syntax | Collect Feature over References | Destructor | Reusable | | | | 4 | 4 |
| | | Complex Restructuring | Refactoring | Custom | 1 | | | 1 | 2 |
| | | Extract and Group Attribute | Refactoring | Reusable | | | | 1 | 1 |
| | | Extract Class | Refactoring | Reusable | | | | 1 | 1 |
| | | Extract Subclass | Constructor | Reusable | | | 1 | | 1 |
| | | Extract Superclass | Constructor | Preserving | 3 | | 1 | 5 | 9 |
| | | Flatten Composite Hierarchy | Destructor | Reusable | | | | 1 | 1 |
| | | Generalize Attribute | Constructor | Preserving | 1 | | | | 1 |
| | | Generalize Reference | Constructor | Preserving | 1 | | | 5 | 6 |
| | | Inheritance to Delegation | Refactoring | Reusable | | | 1 | 2 | 3 |
| | | Inline Superclass | Destructor | Reusable | 1 | | | 2 | 3 |
| | | Merge Classes | Destructor | Reusable | | | | 2 | 2 |
| | | Merge Enumerations | Destructor | Reusable | | | 2 | 2 | 4 |
| | | Move Feature over Reference | Constructor | Reusable | | | | 3 | 3 |
| | | Pull Feature over References | Refactoring | Reusable | | | | 1 | 1 |
| | | Pull up Feature | Constructor | Preserving | | | | 3 | 3 |
| | | Push down Feature | Destructor | Reusable | 4 | | 2 | 1 | 7 |
| | | Push Feature over References | Refactoring | Reusable | | | | 1 | 1 |
| | | Specialize Supertype | Constructor | Preserving | 6 | | | | 6 |
| | | Unfold Superclass | Destructor | Preserving | 1 | | | | 1 |
| | Constraint | Move Constraint Annotation | Refactoring | Preserving | | | 1 | 2 | 3 |
| | API | Move Operation | Refactoring | Preserving | | | | 1 | 1 |
| | | Operation to Volatile Feature | Refactoring | Preserving | | | | 3 | 3 |
| | | Pull up Operation | Refactoring | Preserving | | | | 3 | 3 |
| | | Push down Operation | Refactoring | Preserving | | | | 1 | 1 |
| | | Volatile to Opposite Reference | Refactoring | Preserving | | | | 1 | 1 |
| | | | | | **84** | **9** | **84** | **554** | **731** |




(a) Granularity (b) Metamodel aspect

(c) Language preservation (d) Model migration

**Figure 7.8:** Classification of operations along the different criteria

few changes cannot be covered by reusable coupled operations which are able to automatically migrate models: we identified 630 (86%) PreservingOperations, 95 (13%) ReusableCoupledOperations, and 6 (1%) CustomCoupledOperations. As can be seen in Table 7.4, a custom migration was necessary 4 times to initialize a new mandatory feature or a feature that was made mandatory. In these cases, the migration is associated to one PrimitiveOperation, and consists of 10 to 20 lines of handwritten code. Additionally, 2 custom migrations were necessary to perform a complex restructuring of the model. In these cases, the migration is associated to a sequence of 11 and 13 PrimitiveOperations, and consists of 60 and 70 lines of handwritten code, respectively.

In a nutshell, a large fraction of changes can be captured by primitive changes or operations which are independent of the metamodel. A significant number of operations are known from object-oriented refactoring. Only very few changes were specific to the metamodel, denoting more complex evolution.

### 7.2.5 Study Discussion

Based on the results of our case study, we learned a number of lessons about the evolution of modeling languages in practice.

**Other Language Artifacts need to be Migrated (RQ1).** The migration is not restricted to models, but also concerns other language artifacts, e.g. transformers and code generators. During the evolution of GMF, these artifacts needed to be migrated manually. In contrast to models, these artifacts are mostly under control of the language engineers, and thereby their migration is not necessarily required to be automated. However, automating the migration of these artifacts would further reduce the effort involved in language evolution. The model-based development of metamodels with EMF facilitated the identification of changes between two different versions of the metamodel. In contrast, the specification of transformers and code generators as Java code made it hard to trace the evolution in a model-based manner. We thus need a more structured and appropriate means to describe the other language artifacts depending on the metamodels. Language engineering could benefit from the same advantages as model-based software development.

**Metamodels Evolve due to both User Requests and Technological Change (RQ2).** On the one hand, a metamodel defines the abstract syntax of a language, and thereby metamodels evolve when the requirements of the language change. In GMF, user requests for new features imposed many of such changes to the GMF modeling languages. On the other hand, an API for model access is intimately related to a metamodel, and thereby metamodels evolve when requirements for model access change. In GMF, particularly the shift from JET to XPand as the generator implementation language imposed many of such changes in the gmfgen metamodel. Since a metamodel captures the abstract syntax as well as the API for model access, language and API evolution interact. Changes in the abstract syntax clearly lead to changes in the API. But API changes can also require to change the abstract syntax of the underlying language: in GMF, we found several cases where the abstract syntax was changed to simplify model access.

**Language Evolution is Similar to Software Evolution (RQ2, RQ3).** This hypothesis was postulated by Favre in [Favre, 2005]. The answers to *RQ2* and *RQ3* provide evidence that the hypothesis holds. First, the distribution of activities performed by the engineers of GMF to implement language changes mirrors the distribution of classical software maintenance activities (i.e. perfective and adaptive maintenance activities being the most frequent) [Lientz and Swanson, 1980]. Second, many operations to adapt the metamodels (see Table 7.4) are similar to operations known from object-oriented refactoring [Fowler, 1999] (e.g. Extract Superclass). Like software evolution, the time scale for language evolution can be quite small. In the first year of the investigated evolution of GMF, the metamodels were changed 107 times, i.e. on average every four days. However, in the second year the number of metamodel changes decreased to 17, i.e. the stability of GMF increased over time. It thus seems that the time scale in which the changes happen increases with the language's maturity. The same phenomenon applies to the relation between the metamodels and the metametamodel, as the evolution of ecore required the migration of the GMF

metamodels. However, the more abstract the level, the less frequent the changes: we identified two changes in the metametamodel of the investigated evolution of GMF.

**Operation-based Coupled Evolution of Metamodels and Models is Feasible (RQ3).** The engineers of GMF provided a migrator to automatically migrate the already existing models. This migrator allows the GMF engineers to make changes that are not backwards-compatible, and are essential as the kinds and number of built models are not under control of the language engineers. We reverse engineered the evolution of the GMF metamodels by sequencing operations. Most of the metamodel evolution can be covered by operations which are independent of the specific metamodel. Only a few custom operations were required to capture the remaining changes. The employed operations can be used to migrate the models as well. In addition, the case study provides evidence for the suitability of operation-based metamodel evolution in forward engineering as proposed in Chapter 5 (*COPE – Coupled Evolution of Metamodels and Models*). Operation-based forward engineering of modeling languages documents changes on a high level of abstraction which allows for a better understanding of language evolution.

### 7.2.6 Threats to Validity

We discuss our results with respect to their construct, internal and external validity.

**Construct Validity.** The results might be influenced by the measurement we used for our case study. We assumed that a commit represents exactly one language change. However, a commit might encapsulate several language changes, and one language change might be distributed over several commits. This interpretation is a threat to the results for both *RQ1* and *RQ2*. Other case studies might be required to investigate these research questions in more detail, and to increase the confidence and generality of our results. However, our results are consistent with the view that languages evolve like software, which was postulated and tacitly accepted as a fact [Favre, 2005].

**Internal Validity.** The results might be influenced by the method applied for investigating the evolution. The algorithm used to detect the commits (step 2) might miss language artifacts which were also committed together. To mitigate this threat, we have manually validated the commits by checking their temporal neighborhood. By filtering out the commits which did not change the metamodel (step 3), we might miss language changes not affecting the metamodel. Such changes might be changes to the language semantics defined by code generators and transformers. However, the model migration defined by the handcrafted migrator could be fully assigned to metamodel adaptations. We might have misclassified some commits, when classifying the commits according to the maintenance categories (step 5). However, the results are in line with existing evidence on software evolution [Lientz and Swanson, 1980]. When detecting the operation sequence (step 8), the picked operations might have a different intention than the engineers had when performing the changes. To mitigate this threat, we have automatically validated the model migration by means of test cases. Furthermore, we have manually validated

the migration of all language artifacts by taking their co-evolution into account.

**External Validity.** The results might be influenced by the fact that we investigated only a single system. The modeling languages provided by GMF are among the many modeling languages that are developed using EMF. The relevance of our results obtained by analyzing GMF can be affected when analyzing languages developed with other technologies. Our results are however in line with existing evidence on grammar evolution [Lämmel and Verhoef, 2001, Lämmel and Zaytsev, 2009b], and this increases our confidence that the defined operations are valid for many other languages. Furthermore, our previous studies on the evolution of metamodels—see Chapter 3 (*State of the Practice: Automatability of Model Migration*) and Section 7.1 (*GMF Generator Model and Palladio Component Model*)—revealed similar results.

## 7.3 Quamoco Quality Metamodel

Quamoco[12] is a research project whose goal is to develop a language for modeling the product quality of software. Currently, we are applying COPE for the evolutionary development of the metamodel on which the modeling language is based. In this case study, we analyzed the differences between a forward and reverse engineering use case of our approach.

### 7.3.1 Study Goal

We performed the study to test the applicability of COPE to real-world coupled evolution. In contrast to the previous studies, we applied COPE to forward engineer the coupled evolution. More specifically, we performed the study to answer the following research questions:

- **RQ1.** *What are the reasons for the metamodel changes?* We classify the metamodel changes according to the maintenance activities in order to determine the reasons for carrying them out.
- **RQ2.** *To what extent can the coupled evolution of metamodel and models be automated?* To measure automatability, we determine the fraction of simple metamodel extensions, reusable and custom coupled operations.
- **RQ3.** *Is the forward engineered coupled evolution different from the reverse engineered coupled evolution?* We compare the results of this forward engineering case study to the results of the reverse engineering case studies presented in Section 7.1 (*GMF Generator Model and Palladio Component Model*) and Section 7.2 (*Graphical Modeling Framework*).

### 7.3.2 Study Object

This section describes the quality metamodel whose evolution we supported with COPE. The quality metamodel was developed in the Quamoco project and defines

[12] see Quamoco web site: `http://www.quamoco.de`

a structure to which a quality model needs to conform. We use quality models to specify and evaluate the quality of software products [Kläs et al., 2010]. Figure 7.9 shows a simplified version of the quality metamodel.

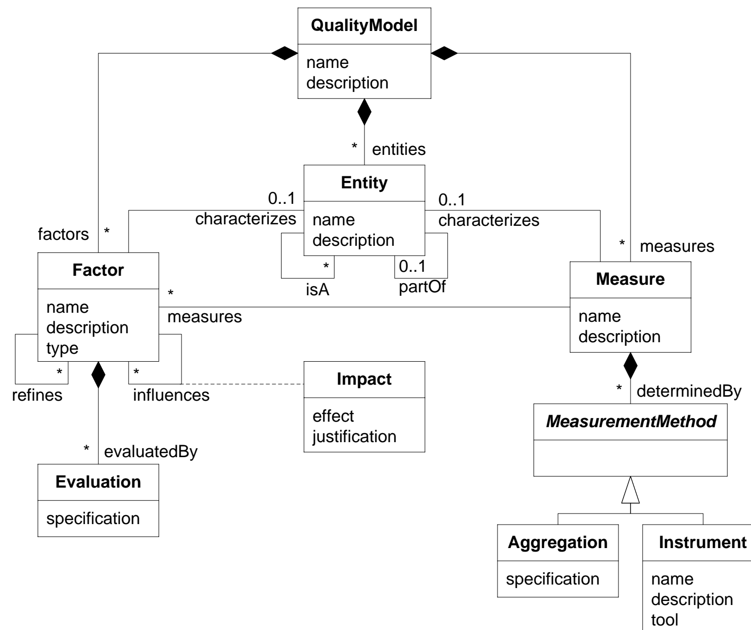



**Figure 7.9:** Quamoco metamodel (simplified version)

A QualityModel consists of entities, factors and measures. An Entity is used to model a part of the software product or its environment. Entities can form a hierarchy which is specified by isA and partOf relationships. A Factor defines a property of a software product and thus characterizes the corresponding entity. Factors can form a hierarchy which is specified by refines relationships. Impacts define influences with positive or negative effect between different factor hierarchies. A Measure measures factors, i.e. provides a means to assess the factors, and thus also characterizes an entity. Measures can be determined by different MeasurementMethods: An Aggregation combines other measures, and an Instrument employs a tool or review technique. An Evaluation evaluates a factor based on its measures as well as the factors that refine or influence the factor. Most classes define attributes to specify a name and description.

### 7.3.3 Study Execution

We started with a metamodel that has already been developed in our group before the Quamoco project [Deissenboeck et al., 2007]. However, in Quamoco, we also had to consider the requirements of the other project partners: Capgemini, itestra, SAP and Siemens. Therefore, we decided to use an evolutionary method to adapt the quality metamodel developed in our group to these requirements. This method consisted of the following steps:

1. *Elicitation of changes:* The Quamoco project is organized in a number of iterations. In each iteration, a version of the quality metamodel is developed. There have been a number of workshops with all partners in each iteration

in which changes to the metamodel were discussed and decided. After iteration 2, we decided to perform sprints—known from the agile development method Scrum [Schwaber and Beedle, 2002]—to build a generically applicable quality model with the metamodel. These sprints also helped to elicit further metamodel adaptations.

2. *Implementation of changes:* We implemented the changes on the quality metamodel with COPE in order to be able to migrate existing models. We tried to use as much reusable coupled operations from the library as possible to perform the changes. If a migration recurred or we had the same migration in an earlier case study, we implemented a new reusable coupled operation. If reuse of the migration made no sense, we implemented it as a custom coupled operation. We released the metamodel, whenever it was required to build models with it.
3. *Validation of changes:* The project partners reviewed the changes to validate them. Moreover, based on the metamodel, we implemented an editor using EMF which has been used throughout the project by different project partners to build models for their particular domain. Thereby, the users got a feeling about whether the adaptations improved the metamodel. The reviews of the metamodel and the usage of the editor often have lead to new metamodel changes.
4. *Validation of migration:* Since we participated in the project, we knew the intention behind the changes and thus how to implement the changes by coupled operations. However, the implicit semantics in the domain of software quality often gave rise to a number of possible model migrations. To validate the specified migration, we discussed the migration with model developers and let the result of the migration on a certain model be reviewed by the developers of the model.

We performed a number of iterations using these steps. COPE has been used to support the coupled evolution of the Quamoco metamodel for 1.5 years now, and is still being used.

### 7.3.4 Study Result

We first illustrate the resulting metamodel evolution, before we discuss the reasons for language changes as well as the degree of possible automation.

**Evolution in Numbers.** Figure 7.10 illustrates the evolution of the Quamoco metamodel in numbers of metamodel elements. The figure also indicates the different phases: When importing the existing metamodel, we reduced the number by removing elements that were not important for iteration 1. In iteration 1 and 2, we gradually increased the size of the metamodel due to 4 releases in each iteration. During each of the 3 sprints, we further adapted the metamodel. Compared to the earlier case studies, the Quamoco metamodel is subject to constant growth. Therefore, we consider the metamodel as being still in its initial development phase.

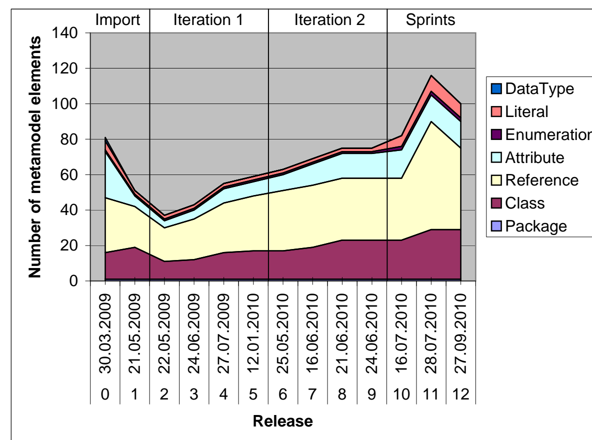



**Figure 7.10:** Evolution of the Quamoco metamodel in numbers

**Reasons for Language Changes.** To determine the reasons for language changes, we have classified the overall 50 commits that changed the metamodel according to the maintenance categories. Figure 7.11 and Table 7.5 show the number of commits for each category, similarly to the foregoing study.

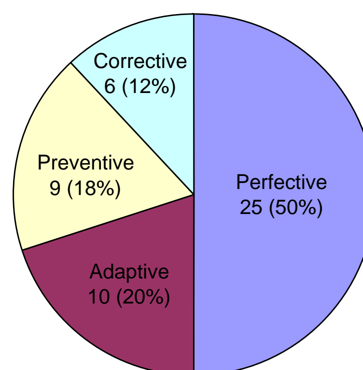



**Figure 7.11:** Share of maintenance categories

We classified half of the commits as *perfective* maintenance, i.e. add new features to enhance the Quamoco metamodel. In Table 7.5, we group these commits according to the constructs that they add or refine. We classified 20% of the commits as *adaptive* maintenance, i.e. adapt the Quamoco metamodel to a changing environment. These commits were either due to changing the generated structural editor, releasing the metamodel, the import of the XML schema to EMF, or to prepare it for a graphical editor implemented in GMF. We classified 18% of the commits as *preventive* maintenance, i.e. refactor the Quamoco metamodel to prevent faults in the future. These commits either introduced better names, simplified the metamodels to make the evaluation more straightforward, or added documentation to make the metamodel more understandable. We classified 12% of the commits as *corrective* maintenance, i.e. correct faults discovered in the Quamoco metamodel. These commits either fixed the metamodel to avoid dangling references in models, corrected multiplicities of features or the abstractness of a class, or fixed the order of super type declarations for code generation.

**Table 7.5:** Classification of metamodel commits according to the maintenance categories

| Perfective | 25 | Adaptive | 10 | Preventive | 9 | Corrective | 6 |
|---|---|---|---|---|---|---|---|
| Type system | 5 | Structural editor | 5 | Better names | 4 | Dangling references | 2 |
| Measure | 4 | Release | 2 | Simplification | 3 | Multiplicities | 2 |
| Warning Constraints | 3 | XML Schema -> Ecore | 2 | Documentation | 2 | Abstract class | 1 |
| Annotation and Tag | 2 | Graphical editor | 1 | | | Super type order | 1 |
| Evaluation | 2 | | | | | | |
| Instrument | 2 | | | | | | |
| Modularization | 2 | | | | | | |
| Entity | 1 | | | | | | |
| Factor | 1 | | | | | | |
| Quality Requirement | 1 | | | | | | |
| Source | 1 | | | | | | |
| Tool | 1 | | | | | | |

**Automatability of the Coupled Evolution.** Figure 7.12 depicts the number of the different classes of metamodel adaptations that were used to model the coupled evolution. Metamodel extensions account for 43% of the metamodel adaptations, whereas reusable coupled operations account for 54%. Table 7.6 refines the classification by listing the names and number of occurrences of the different metamodel adaptations. As metamodel extensions, we see simple model-preserving adaptations as well as primitive changes that do not affect the syntax of the modeling language. Both categories are separated by a dashed line. Indicated in the same way, we distinguish reusable coupled operations known from the library as well as new reusable coupled operations. The low number of new operations shows that most of the coupled evolution can be covered by operations already in the library. Moreover, the 3 operations Change Namespace Prefix, Change Namespace URI and Change Attribute Default Value are trivially supported by COPE's generic instance structure.

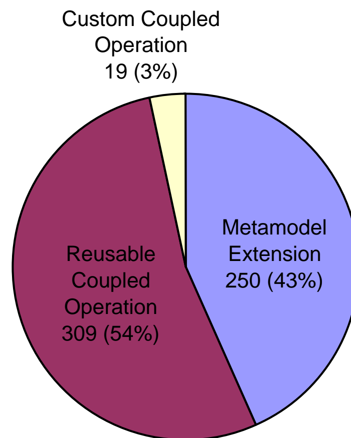



**Figure 7.12:** Classification of the Quamoco language changes

The remaining 3% of the metamodel adaptations consist of 19 custom coupled operations. Table 7.7 lists the different custom coupled operations together with the release in which they occurred and the lines of handwritten code. In average, each custom coupled operation required to write 27 lines of code. Many of the custom coupled operations are metamodel-specific variations of reusable coupled operations. In the table, we indicate this by highlighting the name of the corresponding reusable cou-

**Table 7.6:** Coupled operations required for the Quamoco case study

| **Phase** | **Imp.** | **Iteration 1** | | | | **Iteration 2** | | | | **Sprints** | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Release** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **all** |
| **Coupled Operation** | **199** | **32** | **21** | **97** | **26** | **101** | **11** | **8** | **4** | **5** | **50** | **24** | **578** |
| **Metamodel Extension** | **25** | **3** | **5** | **15** | **18** | **21** | **3** | **5** | **2** | **1** | **19** | **2** | **250** |
| Add Super Type | | | | 5 | | 2 | | 2 | | | 4 | | **13** |
| Create Attribute | 3 | | | 2 | | 2 | | | | | | | **7** |
| Create Class | 6 | | | 3 | 1 | 3 | | 3 | | | 1 | | **17** |
| Create Enumeration | 1 | | | | | | | | | 1 | | | **2** |
| Create Literal | | | | | | | | | | | 3 | | **3** |
| Create Reference | 10 | 2 | 1 | 3 | 2 | 10 | | | | | 3 | 1 | **32** |
| Create Volatile Attribute | | | | | | | 3 | | | | | | **3** |
| Create Volatile Reference | | | | | | | | | | | 8 | | **8** |
| Drop Class Abstract | | 1 | | | | | | | | | | | **1** |
| Create Warning Constraint | | | | 1 | 7 | | | | | | | 1 | **9** |
| Delete Warning Constraint | | | | | | 3 | | | | | | | **3** |
| Document Metamodel Element | 78 | | | 53 | | | | | | | | | **131** |
| Drop Feature Changeable | 5 | | | | | | | | | | | | **5** |
| Drop Feature Ordered | | | | | 8 | | | | | | | | **8** |
| Drop Reference Resolve Proxies | | | | 1 | | | | | | | | | **1** |
| Make Feature Changeable | | | 4 | | | | | | | | | | **4** |
| Make Feature Unsettable | | | | | | | | | 1 | | | | **1** |
| Make Reference Resolve Proxies | | | | | | 1 | | | 1 | | | | **2** |
| **Reusable Coupled Operation** | **170** | **29** | **16** | **82** | **8** | **70** | **8** | **2** | **2** | **1** | **30** | **22** | **309** |
| Association to Class | | | | | | 1 | | | | | 4 | | **5** |
| Create Opposite Reference | | 1 | 4 | 1 | 1 | 11 | 1 | | | | 5 | | **24** |
| Delete Class | 8 | 2 | | | | | | | | | | | **10** |
| Delete Data Type | 1 | | | | | | | | | | | | **1** |
| Delete Enumeration | 1 | | | | | | | | | | | | **1** |
| Delete Feature | 13 | 5 | 1 | | | 7 | | | | | | 6 | **32** |
| Delete Opposite Reference | | | | | | | | | | | | 10 | **10** |
| Drop Reference Opposite | | | | | | 6 | | | | | | | **6** |
| Extract Class | 6 | | | 2 | | 2 | | | | | | | **10** |
| Extract Subclass | | | | | | | | 1 | | | | | **1** |
| Extract Superclass | 1 | | | | | 3 | 2 | | | | | | **6** |
| Fold Super Class | 2 | | | | | | | | | | 1 | | **3** |
| Generalize Attribute | 2 | | | | | | | | | | | | **2** |
| Generalize Reference | | 1 | | 3 | 2 | 12 | | | | | | | **18** |
| Identifier to Reference | 1 | | | | | | | | | | | | **1** |
| Inline Class | | 4 | | | | | | | | | | | **4** |
| Inline Subclass | | 2 | | | | 2 | | | | | | | **4** |
| Inline Superclass | 1 | | 1 | | | 1 | | | | | | | **3** |
| Make Class Abstract | 2 | | 1 | | | | | | 1 | | | | **4** |
| Make Reference Composite | | 2 | | | | | | | | | | | **2** |
| Merge Classes | | | | | | 3 | | | | | | | **3** |
| Merge Features | | | | | | 6 | | | | | | | **6** |
| Pull up Feature | | | | | | 4 | | | | | | | **4** |
| Remove Superfluous Super Type | | | | 5 | | | | | | | | | **5** |
| Rename | 42 | 10 | 5 | 12 | 4 | 8 | 3 | | | | 16 | | **100** |
| Specialize Attribute | | | | 1 | | | | | | | | | **1** |
| Specialize Composite Reference | | | 1 | 1 | | | | | | | | | **2** |
| Specialize Reference | 2 | | 2 | 1 | | | | | | | 1 | | **6** |
| Specialize Supertype | | | | | | 1 | 1 | | | | 1 | | **3** |
| Switch Composite | | 1 | | 1 | | 1 | | | | | | 5 | **8** |
| Unfold Superclass | | | | 1 | | | | | | | | | **1** |
| Change Namespace Prefix | 1 | | | | | | | | | | | | **1** |
| Change Namespace URI | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | **12** |
| Extract and Group Attribute | | | | | | | | | | | 1 | | **1** |
| Merge Data Types | 6 | | | | | | | | | | | | **6** |
| Set Attribute Default Value | 2 | | | | | | | | | | | | **2** |
| Specialize Reference Type | | | | | | 1 | | | | | | | **1** |
| **Custom Coupled Operation** | 4 | | | | | 10 | | 1 | | 3 | 1 | | **19** |

pled operation.

**Table 7.7:** Custom coupled operations performed in Quamoco

| Release | Custom Coupled Operation | LoC |
|---|---|---|
| 1 | Complex Identifier to Reference | 17 |
| | Replace attribute by reference | 14 |
| | Replace boolean by enumeration | 12 |
| | Replace structure by attribute | 18 |
| 6 | Remove root class and instance | 10 |
| | Specialize Super Type and initialize an attribute's value | 6 |
| | Complex Merge Classes | 77 |
| | Specialize Reference but preserve links | 39 |
| | Complex Switch Composite | 57 |
| | Complex Move Feature | 86 |
| | Tag instances of a certain class | 18 |
| | Complex Merge Classes | 42 |
| | Replace reference by textual specification | 18 |
| | Automatically fix attribute values | 11 |
| 8 | Lift annotations to first class construct | 32 |
| 10 | Create Attribute and initialize its value | 12 |
| | Create Attribute and initialize its value | 12 |
| | Automatically fix attribute values | 11 |
| 11 | Automatically set tag descriptions | 29 |
| **19** | **altogether** | **521** |

### 7.3.5 Study Discussion

We discuss the results according to the research questions.

**RQ1.** *What are the reasons for the metamodel changes?* Similar to the GMF case study, the typical activities known from software maintenance also apply to metamodel maintenance [Lientz and Swanson, 1980]. Morever, similar to the development of software, the number of perfective activities outranges the preventive, adaptive and corrective activities. Compared to the GMF case study, more commits can be classified as perfective, since the Quamoco metamodel is still in its initial development phase. However, these commits not just added new classes and features like in GMF, but remodeled the existing classes and features to support the new features, leading to more complex model migration.

**RQ2.** *To what extent can the coupled evolution of metamodel and models be automated?* Similar to the reverse engineering case studies, the coupled evolution of metamodel and models can be automated to a large extent: Most of the coupled evolution could be covered by reusable coupled operations, and only very few custom coupled operations had to be implemented. Due to the available tool support for model migration, the share of metamodel extensions is much lower than in case of GMF—similar to PCM, where the language engineers did not care about model migration at all. As a consequence, COPE allows the language engineers to perform more involved metamodel changes without losing the models built with the metamodel.

**RQ3.** *Is the forward engineered coupled evolution different from the reverse engineered coupled evolution?* Concerning the number of coupled operations, the forward engineered coupled evolution is much more extensive than the reverse engineered coupled evolution: The number of coupled operations applied to the Quamoco metamodel is nearly as high as for the much larger GMF metamodels. Concerning the automatibility, the coupled evolution forward engineered by this case study is not much different from the reverse engineered metamodel histories: Most of the coupled evolution can still be covered by reusable coupled operations. Many of the reusable coupled operations identified in the previous case studies proved to be useful for the adaptation of the Quamoco metamodel. Moreover, forward engineering requires less effort than reverse engineering, since for the latter, a lot of effort is necessary for understanding the intended migration.

### 7.3.6 Threats to Validity

We are aware that our result can be influenced by threats to construct, internal and external validity.

**Construct Validity.** The result might be influenced by the measurement we used for our case study.

To determine the automatability of the coupled evolution, we counted the number of operations. The result might be distorted if custom coupled operations are more complex than reusable coupled operations. However, the custom coupled operations in average required to write 27 lines of code which is not too different from the average size of the implementation of a reusable coupled operation.

To determine the reasons for language changes, we grouped the operations according to the commits—a commit possibly consisting of multiple changes. However, the result is similar to the result of the GMF case study, and we thus are confident that it truthfully represents the distribution of the effort for the different maintenance activities.

**Internal Validity.** The results might be influenced by the method that we chose for forward engineering the coupled evolution. In step 2, we might have changed the metamodel in a way that was not demanded by the project partners. To mitigate this threat, we validated the changes by means of step 3. In step 3, certain issues in the metamodel implementation might be hidden by how the editor represents the metamodel. However, the implementation was also validated by the project participants in our group that implemented the evaluation based on it. The threat that the chosen model migration is incorrect was mitigated by step 4. Finally, we might have misclassified the commits with respect to the maintenance categories. However, we are confident that the result is correct, since we performed the changes ourselves and thereby did not falsify the results by having to understand the changes.

**External Validity.** The results might be influenced by the fact that we forward engineered the coupled evolution for a single metamodel. However, the results obtained by the Quamoco case study are similar to the results of the case studies in which we

reverse engineered the coupled evolution of PCM and GMF (see Section 7.1 (*GMF Generator Model and Palladio Component Model*) and Section 7.2 (*Graphical Modeling Framework*)). Therefore, we are confident that the result can be generalized to most coupled evolutions—at least in the modelware space. Moreover, other case studies in other technical spaces—e.g. dataware [Curino et al., 2008a] and grammarware [Lämmel and Zaytsev, 2009b]—lead to similar results.

Moreover, the results might be influenced by the fact that we participated in the Quamoco project. We could have changed the metamodel in a way that was best supported by reusable coupled operations in the library. To mitigate this threat, the changed metamodel and the model migration was validated by the other project partners.

## 7.4 Unicase Unified Model

COPE has been applied to migrate models built with the CASE tool Unicase[13] in response to metamodel evolution. Besides models conforming to the metamodel, Unicase also records the histories of the models. To not lose these histories due to metamodel evolution, they also need to be migrated. This case study evaluates whether COPE's language can also be used to specify the migration of the histories.

### 7.4.1 Study Goal

The goal was to build a solution based on COPE that is able to migrate the models as well as their histories. More specifically, the study was performed to answer the following research questions:

- **RQ1.** *Can COPE's language be used to specify the migration of artifacts other than models?* We apply COPE's language to specify the migration of histories recorded with Unicase, which also need to be migrated in response to metamodel adaptation.
- **RQ2.** *Can the migration of the other artifacts be reused in the same way the model migration can be reused?* We implement a mechanism to extend reusable coupled operations with a specification for the migration of Unicase histories.

### 7.4.2 Study Object

Unicase is a CASE tool based on EMFStore[14] which is a version control system suitable for models.

**Unicase** is an EMF-based tool that implements a modeling language based on a unified metamodel [Bruegge et al., 2008]. It consists of a set of editors to manipulate models conforming to the unified metamodel, and a repository to persist and version the models as well as to collaborate on the models. The unified metamodel covers

[13]see Unicase web site: `http://www.unicase.org`
[14]see EMFStore web site: `http://www.emfstore.org`

the whole development process from requirements over design to deployment, including project management artifacts. System model elements such as requirements or UML elements, are part of the same model and stored in the same repository as development process model elements such as tasks or users. To give an overview over the unified metamodel, Table 7.8 shows the current number of classes and enumerations that are contained in the different packages defined by the metamodel.

**Table 7.8:** Number of elements in the Unicase metamodel

| **Packages** | **activity** | **attachment** | **bug** | **change** | **classes** | **common** | **component** | **diagram** | **document** | **meeting** | **organization** | **profile** | **rationale** | **requirement** | **state** | **task** | **util** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **# Classes** | 7 | 2 | 1 | 3 | 1 | 4 | 3 | 7 | 3 | 5 | 3 | 7 | 7 | 1 | 5 | 5 | 1 |
| **# Enumerations** | | 1 | 2 | | 6 | | | | | | | | | | | 1 | |

**EMFStore** provides Unicase with a repository to persist and version EMF models as well as to collaborate on the models. EMFStore employs operation-based change tracking [Koegel et al., 2009b]—which records changes on models as a sequence of performed operations—and operation-based merging [Koegel et al., 2009a]. Operation-based change tracking is more accurate than state-based change tracking—which derives changes from model states by differencing—since it has exact information about the order of the changes as well as composite changes [Koegel et al., 2009b, Koegel et al., 2010a]. Operation-based change tracking is similar to how COPE records changes on metamodels.

The version model of EMFStore—which is shown in Figure 7.13—is a sequence of versions with revision links [Koegel, 2008]. Every Version contains a ChangePackage and may contain a full ModelState representation. A ChangePackage contains all Operations that transformed the previous version into this version along with administrative information such as the modifying user, a time stamp and a log message. In the same way that COPE's history model needs to be consistent with the current version of the metamodel, the ChangePackages of EMFStore need to be consistent with the ModelStates.

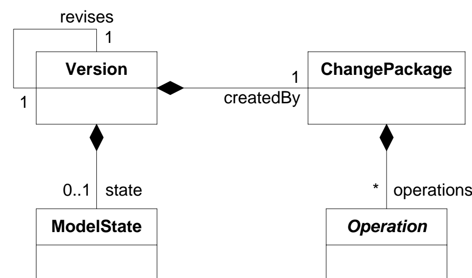



**Figure 7.13:** Version metamodel of EMFStore

Figure 7.14 shows the simplified metamodel for operations stored by the EMFStore. To be able to send the operations to the repository in a self-contained manner, an Operation is independent of the metamodel and model: the Operation refers to the ModelElement that it changes via the element's unique identifier, and a Feature-

Operation refers to the changed feature by the featureName. An AttributeOperation changes the value of an attribute of a model element. A ReferenceOperation creates or removes one or several links between model elements. A CreateDeleteOperation creates or deletes a model element, preserving a copy of the created or deleted model element. A CompositeOperation can group several related operations—e.g. to represent a refactoring. As the operation metamodel of COPE and EMFStore are very similar, we are currently developing a generic operation recorder that can be instantiated in both tools [Herrmannsdoerfer and Koegel, 2010a].

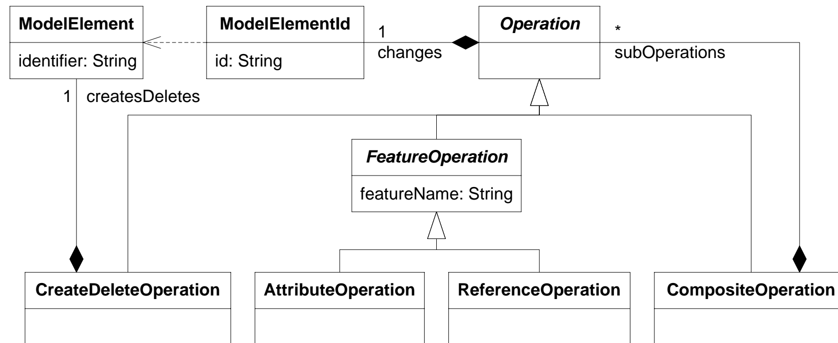



**Figure 7.14:** Operation metamodel of EMFStore

### 7.4.3 Study Execution

We employed a forward engineering approach in which we recorded the coupled operations whenever the metamodel needed to be adapted. The coupled operations do not only need to capture model migration, but also the migration of the recorded operations. To extend COPE with a means to also migrate the operations, we used the following method:

1. *Coupled evolution:* When Unicase was maintained by its language engineers, the need arose to adapt the metamodel. The changes were recorded by using the coupled operations provided by COPE. At the beginning, we performed the changes with COPE, and later, the Unicase language engineers used COPE themselves to record the changes.
2. *Specification of change migration:* For each coupled operation, we analyzed whether the encapsulated metamodel adaptation also impacts the recorded operations. If yes, we specified an appropriate change migration in the language provided by COPE. Since the operations refer to the model elements by identifier, the operations need to be loaded together with the corresponding model state during migration so the change migration can resolve the identifiers to the model elements. After this step, we can answer *RQ1*, i.e. whether we are able to specify the change migration with COPE's language.
3. *Validation of change migration:* The consistency between the recorded operations and the model states is crucial for the functions provided by EMFStore. Consequently, we need to validate that the combination of model and change migration preserves the consistency. EMFStore has been extended with checks to automatically verify the consistency during startup. We rely on these checks,

when testing the change migration.

4. *Reuse of change migration:* Like the model migration, we might want to reuse the change migration. When a reusable coupled operation was used for the first time, we specified the change migration by attaching a custom migration. When a reusable coupled operation recurred, we tried to specify the change migration in a metamodel-independent manner. In order to do so, we need a means to extend reusable coupled operations with specifications of how to migrate other artifacts. After this step, we are able to answer *RQ2*, i.e. whether we can also reuse the change migration similarly to the model migration.

We performed all the steps in close cooperation with the language engineers of Unicase. COPE has been used to migrate Unicase models for a period of more than a year, and is still being used.

### 7.4.4 Study Result

We first explain how we attach a custom change migration to a coupled operation using COPE's language. Then, we introduce a mechanism to extend a reusable coupled operation with a reusable change migration. Finally, we give an overview over the coupled operations that have been performed during the Unicase evolution.

**Custom Change Migration (RQ1).** When performing a coupled operation, the recorded changes might no longer be consistent with the model states. To preserve the consistency between changes and states, we may also need to migrate the changes. Consequently, for each coupled operation, we need to determine whether it impacts the changes, and if it does, how they need to be migrated. At first, we were defining the change migration as a custom migration, even in the case of reusable coupled operations.

Listing 7.1 shows the custom change migration for renaming the attribute Status of the class BugReport. The model migration is already provided by the reusable coupled operation Rename (line 4). Since the changes refer to the attribute by name, we need to change the featureName accordingly. More specifically, we iterate over all FeatureOperations, check whether they change the feature, and if they do, change the featureName (lines 22-26). To check whether a FeatureOperation changes a feature, it is not enough to compare the featureName, since there might be multiple features with the same name in independent classes. Therefore, we have to check whether the changed ModelElement is an instance of the class in which the feature can be used (helper method isFeatureChange, lines 11-20). In order to do so, we need to resolve the ModelElement based on the ModelElementId by which an Operation refers to the ModelElement (helper method getElementById, lines 7-9).

**Reusable Change Migration (RQ2).** When there was a second application of Rename, we tried to also reuse the change migration. However, we do not want to overwrite the reusable coupled operation in the library, but rather specify the Unicase-specific change migration separately. In order to do so, we have employed a technique known from aspect-orientation to extend an operation without changing it: We can specify operations that are automatically injected before or after an exist-

**Listing 7.1:** Custom change migration for renaming an exemplar attribute

```
statusAttribute = model.bug.BugReport.Status

// metamodel adaptation + model migration
rename(statusAttribute, "done")

// change migration
getElementById = { id ->
  return model.UnicaseModelElement.allInstances.find{e -> id.equals(e.identifier)}
}

isFeatureChange = { operation, EStructuralFeature feature ->
  if(feature.name.equals(operation.featureName)) {
    def id = operation.modelElementId.id
    def element = getElementById(id)
    if(element == null || element.instanceOf(feature.eContainingClass)) {
      return true
    }
  }
  return false
}

for(operation in esmodel.versioning.operations.FeatureOperation.allInstances) {
  if(isFeatureChange(operation, statusAttribute)) {
    operation.featureName = name
  }
}
```

ing reusable coupled operation during the migrator generation. We have introduced annotations to specify that an operation extends another operation. Of course, the extending operation needs to have the same signature as the extended operation.

Listing 7.2 shows the use of the before annotation to extend the operation Rename with a change migration (line 19). Even though Rename can be applied to all metamodel elements, we only need to extend it for features (lines 20-24), since the renaming of other metamodel elements does not impact the changes. Since a feature can also be a reference, the change migration needs to be more general as the custom change migration shown in Listing 7.1. Besides the featureName of FeatureOperations (lines 2-6), we also need to migrate the oppositeFeatureName of ReferenceOperations (lines 7-16).

**Performed Coupled Operations (RQ1, RQ2).** Table 7.9 gives an overview over the coupled operations that we have performed over a time interval of more than one year. The table provides columns to show the name of the coupled operation, to indicate the change migration and to depict the number of its applications. We distinguish metamodel extensions, reusable and custom coupled operations.

*Metamodel extensions.* Few of the coupled operations are metamodel extensions which do not require a model migration and thus neither a change migration. This indicates that the Unicase modeling language had already passed the initial development phase, when we started to use COPE to migrate models. Note that the unicase annotations are only used for customizing the representation of the metamodel elements in the Unicase editor.

*Reusable coupled operations.* Most of the coupled operations are reusable coupled operations. However, most of them only affect the serialization of models, thus are triv-

**Listing 7.2:** Reusable change migration for reusable coupled operation Rename

```
1  unicaseRenameFeature = { EStructuralFeature feature, String name ->
2    for(operation in esmodel.versioning.operations.FeatureOperation.allInstances) {
3      if(isFeatureChange(operation, feature)) {
4        operation.featureName = name
5      }
6    }
7    if(feature instanceof EReference) {
8      def opposite = feature.eOpposite
9      if(opposite != null) {
10       for(operation in esmodel.versioning.operations.ReferenceOperation.allInstances
            ) {
11         if(isFeatureChange(operation, opposite)) {
12           operation.oppositeFeatureName = name
13         }
14       }
15     }
16   }
17 }
18
19 @before("rename")
20 unicaseRename = {ENamedElement element, String name ->
21   if(element instanceof EStructuralFeature) {
22     unicaseRenameFeature((EStructuralFeature) element, name)
23   }
24 }
```

ially supported by COPE's generic instance structure, and do not affect the changes. Therefore, we have not mentioned these operations in the library and show them below the dashed line in the table. From the other reusable coupled operations, we only needed to extend two operations with a change migration.

*Custom coupled operations.* Only very few coupled operations are custom coupled operations, most of which also require a custom model migration. The only exception is Enumeration to Subclasses which provides a reusable model migration. A change to the value of the enumeration attribute needs to be migrated to a change of the type of the model element. However, type changes during model evolution are not supported by EMF. We could only perform the change due to the specific usage of the attribute in Unicase: Unicase ensured that the value of the enumeration attribute is never changed, after the model element has been created.

### 7.4.5 Study Discussion

We applied COPE to forward engineer the coupled evolution of the Unicase metamodel and its models. To support model evolution, Unicase records the changes performed on the models and stores them in an EMFStore repository. When the metamodel is adapted, we thus do not only have to migrate the models, but also the recorded changes. While COPE's migration language showed to be expressive enough to specify the required change migrations, we needed to extend COPE to be able to refine existing reusable coupled operations with the appropriate change migration. Change migrations that can be easily expressed only modify or delete existing changes and do not require the state of the model before or after a change.

However, certain coupled operations cannot be easily performed, if the consistency between models and changes needs to be preserved. First, change migrations which

**Table 7.9:** Coupled operations performed during the Unicase case study

| Coupled Operation | Change Migration | Number |
|---|---|---|
| **Metamodel Extension** | | **39** |
| Add Super Class | No migration necessary | 6 |
| Create Annotation (unicase) | No migration necessary | 7 |
| Create Attribute | No migration necessary | 9 |
| Create Class | No migration necessary | 4 |
| Create Enumeration | No migration necessary | 3 |
| Create Reference | No migration necessary | 9 |
| Create Volatile Reference | No migration necessary | 1 |
| **Reusable Coupled Operation** | | **177** |
| Delete Enumeration | No migration necessary | 1 |
| Delete Feature | Delete FeatureOperations | 5 |
| Generalize Reference | No migration necessary | 2 |
| Inline Superclass | No migration necessary | 2 |
| Remove Superfluous Super Class | No migration necessary | 2 |
| Rename | Rename FeatureOperations | 8 |
| Add Reference Key | No migration necessary | 1 |
| Change Namespace Prefix | No migration necessary | 1 |
| Remove Reference Key | No migration necessary | 153 |
| Set Attribute Default Value | No migration necessary | 2 |
| **Custom Coupled Operation** | | **7** |
| Convert Rich Text Strings | Migrate AttributeOperations | 2 |
| Delete Class and Objects (moving its contents) | Delete Operations on objects | 1 |
| Enumeration to Subclasses | Delete AttributeOperations | 1 |
| Make Attribute Identifier | No migration necessary | 1 |
| Replace Enumeration by Boolean as Attribute Type | Migrate AttributeOperations | 1 |
| Replace Identifier Attribute by UUID | No migration necessary | 1 |

require to create changes in the history are difficult to implement. For instance, Extract Class requires to add changes to create the extracted instance as well as to attach it to the context instance. We believe that such change migrations could in principal be handled by COPE, but would require implementation of more complicated algorithms. Second, change migrations which require the state of the model before or after a change require additional techniques. For instance, Move Feature over Reference requires to know the value of the reference before a change that needs to be migrated. Currently, EMFStore only stores the state after a number of changes in order to save space. To be able to implement such change migrations in all generality, we would need to reconstruct these intermediate model states during migration. However, this would strongly reduce the performance of the coupled operations.

### 7.4.6 Threats to Validity

We discuss our results with respect to their internal and external validity.

**Internal Validity.** The results might be influenced by the method we chose for the case study. We only implemented the change migration for coupled operations required in the case study, and thus we cannot be sure whether the technique works for other coupled operations (step 2). However, we were interested in supporting

only the changes that really were performed in practice. During validation (step 3), the change migration might work fine on the chosen change histories and thereby hide errors that might happen when migrating other changes. However, the model states and change histories were quite large, having more than thousand model elements and several hundreds of versions. Moreover, when building reusable change migrations (step 4), we might miss certain cases that need to be considered to make them generally applicable. To mitigate this threat, we only built them when they recurred, and let the resulting change migration be reviewed by the language engineers of Unicase.

**External Validity.** The results might be influenced by the fact that we restricted ourselves to a certain kind of artifact. Consequently, we can only be sure that COPE's language can be applied to specify certain kinds of change migrations. However, the migration of changes is probably not the most easiest task, as we have seen. Therefore, we believe that the language can be used to migrate other artifacts in response to metamodel adaptation. Future case studies are necessary to demonstrate the ability to express the migration of other artifacts.

## 7.5 Transformation Tool Contest

We have participated in the Transformation Tool Contest (TTC)[15] in order to compare COPE to different model transformation tools. We concentrated on the model migration case, as the primary scope of COPE is model migration and not model transformation in general. The model migration case demands the implementation of a model migration for activity diagrams from UML 1.4 to 2.2.

### 7.5.1 Study Goal

The aim of TTC is to compare the expressiveness, usability and performance of model transformation tools along a number of selected case studies. The approach of the contest is that different persons apply their favorite model transformation tool to one of the proposed cases. We applied COPE to the model migration case to compare it to the other model transformation tools. The contest specifically targets the following research questions:

- **RQ1.** *What are the pros and cons of different model transformation tools considering a certain application?* By comparing the solutions for a single case, we are able to identify the pros and cons of the different tools concerning the case.
- **RQ2.** *What are the merits of different tool features that help to improve model transformation tools and to indicate open problems?* Through the comparison of the solutions, tool developers might learn how to improve their model transformation tool by detecting issues in their tool and by learning from the features of other tools.

[15]see TTC web site: `http://planet-research20.org/ttc2010`

### 7.5.2 Study Object

The contest provided 3 cases of which we selected the model migration case [Rose et al., 2010c]. The other cases targeted other applications of model transformation tools which are not the primary scope of COPE.

**Core Task.** As the core task, the model migration case requires the specification of a model migration for activity diagrams from UML 1.4 to 2.2. Figure 7.15 depicts both versions of the corresponding metamodel. The metamodels shown in the figure are pruned to the minimally necessary classes. Both figures show classes that more or less correspond to each other in about the same positions.

In UML 1.4 [Object Management Group, 2001], activity models are a sub type of state machines, as shown in Figure 7.15(a). A StateMachine defines a root (top) state and a number of transitions. There are different kinds of states (StateVertex): simple States, Pseudostates, FinalStates, ActionStates, ObjectFlowStates and CompositeStates. A Transition has a source and target state as well as a Guard which is defined by a BooleanExpression. ActivityGraph refines StateMachine by a number of Partitions to group the contents of the activity model.

In UML 2.2 [Object Management Group, 2009], activity models are defined by separate classes, as shown in Figure 7.15(b). An Activity model defines a number of nodes, edges and groups (ActivityPartition). There are different kinds of nodes: ActivityNodes, InitialNodes, JoinNodes, ForkNodes, DecisionNodes, ActivityFinalStates, OpaqueActions and ObjectNodes. An ActivityEdge has a source and target node as well as a BooleanExpression as guard. An ActivityPartition groups a number of nodes and edges.

The case defines the criteria shown in Table 7.10 with respect to which the solutions and thus the tools should be evaluated and compared.

**Extensions.** Besides the core task to specify a model migration between the two metamodel versions, the case defines three extensions.

*Alternative object flow state migration semantics.* ObjectFlowStates can be migrated in a different way: Instead of migrating an ObjectFlowState to ObjectNode, it should be migrated to ObjectFlow, thereby replacing an object by a link.

*Concrete syntax.* The concrete syntax of activity models built with ArgoUML[16] should also be migrated.

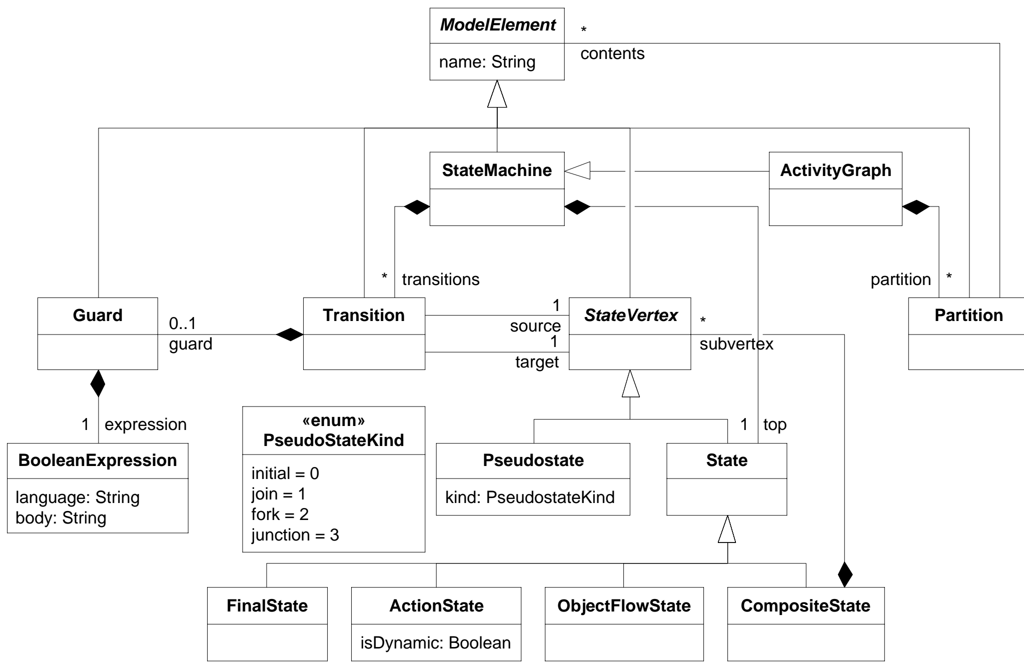
*XMI.* Models that are stored using an older version of the XMI standard [Object Management Group, 2007] should also be migrated.

### 7.5.3 Study Execution

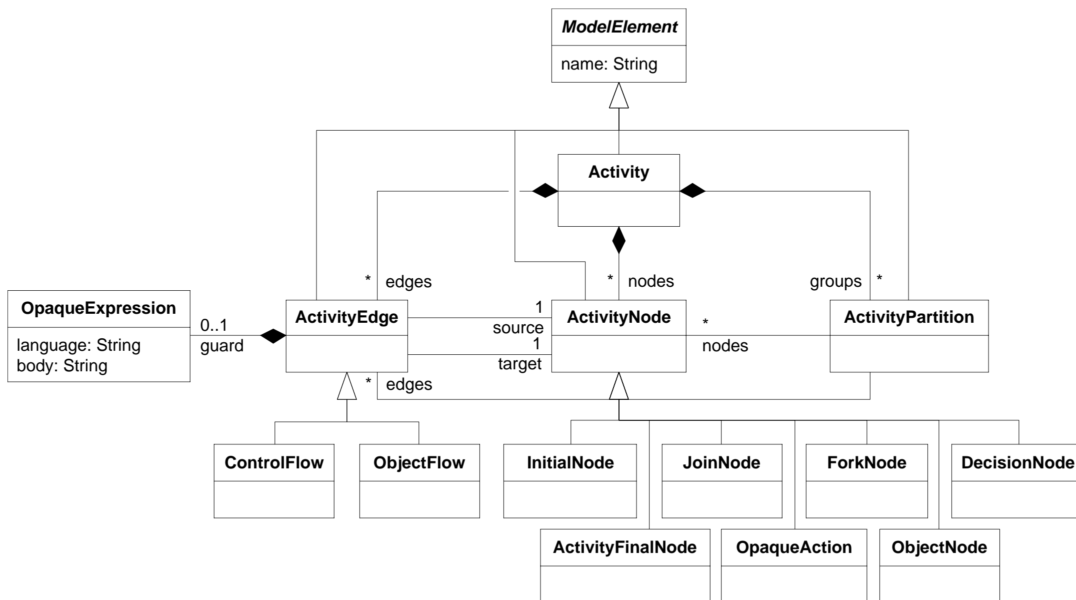
We participated with COPE in the contest. The organizers of the contest devised the following steps to carry out the contest:

1. *Submission of solution:* Potential participants submitted solutions for the cases

[16]see ArgoUML web site: `http://argouml.tigris.org/`




(a) UML 1.4




(b) UML 2.2

**Figure 7.15:** Metamodel for UML activity diagrams

**Table 7.10:** Criteria for solutions to the migration case, taken from [Rose et al., 2010c]

| Name | Description |
|---|---|
| Correctness | Does the transformation produce a model equivalent to the migrated UML 2.2 model included in the case resources? |
| Conciseness | How much code is required to specify the transformation? |
| Clarity | How easy is it to read and understand the transformation? |

to the contest. A valid submission consists of an installation in a remote virtual machine to make the solution reproducible, an accompanying document which describes the solution, and an appendix with the full listing of the solution. The participants could also discuss the case in a forum that was set up by the organizers.

2. *Review of solution:* The submitted solutions were reviewed by experts in the domain of model transformation to evaluate them with respect to validity for the contest. As a result, a number of submitted solutions were accepted for the contest.
3. *Presentation of solution:* The participants presented their solution at the workshop in front of all other participants. The presentation was restricted to 10 minutes. The other participants of the contest could use the installation in the remote virtual machine to play with the solution and tool.
4. *Presentation of opponents:* Each submission needed to have two opponents that critically analyzed the solution before the workshop. Both opponents presented the found issues and open questions together in a 5 minute slot.
5. *Evaluation of solution:* All the contest participants were requested to fill out an evaluation sheet for each solution in parallel to the presentation. The evaluation sheet considered the case criteria as well as the criteria shown in Table 7.11 and required to fill in points for each criterion. Based on the obtained points, the organizers awarded the best solutions for each case.

**Table 7.11:** Additional criteria for evaluation by participants

| Name | Description |
|---|---|
| Appropriateness | How suitable is the tool for the specific application defined by the case? |
| Extensions | To what extent have the extensions defined by the case been solved? |
| Tool Maturity | How mature is the tool? |

While steps 1 and 2 were conducted before the workshop took place, steps 3 to 5 were conducted at the workshop. We only explained the method for what the contest organizers call *offline* cases. Additionally, there were *online* cases for which also the implementation of the solution was performed at the workshop. Even though we also participated with COPE in the online cases, the result is not interesting for the evaluation, since the online cases did not target model migration.

### 7.5.4 Study Result

Altogether nine solutions for the migration case were presented at the workshop, including the one built with COPE [Herrmannsdoerfer, 2010]. The other solutions employed the following model transformation tools: ATL (Atlas Transformation Language) and Java [Cicchetti et al., 2010], Epsilon Flock [Rose et al., 2010b], Fujaba [Koch et al., 2010], GReTL (Graph Repository Transformation Language) [Horn, 2010], GrGen.NET (Graph Rewrite Generator) [Buchwald and Jakumeit, 2010], MOLA (MOdel transformation LAnguage) [Kalnina et al., 2010], PETE (Prolog EMF Transformation Engine) [Schätz, 2010] as well as UML-RSDS [Lano and Rahimi, 2010]. Since we focus on evaluating COPE, we only present the solution of COPE as well as the feedback we got for COPE. We first present the submitted solution, before we present the findings of the reviewers and opponents as well as the result of the evaluation.

**Solution for Core Task.** For the core task, we used the *Metamodel Convergence* function (see Section 6.2.3 (*Recovering the Coupled Evolution*)) to reverse engineer the coupled evolution between both metamodel versions. Listing 7.3 shows the result as a migrator script that is generated from the reverse engineered history model. Table 7.12 provides an overview over the coupled operations, their number of occurrences, effect and type. The coupled operations are classified according to their effect on existing models (see Section 5.1.6 (*Classification of Coupled Operations*)), thereby allowing to reason about the coupled evolution.

**Table 7.12:** Coupled operations required for the migration case

<table>
<tr><th>Coupled Operation</th><th>#</th><th>Effect</th><th>Type</th></tr>
<tr><td>Rename</td><td>21</td><td rowspan="5">Refactoring</td><td rowspan="8">Reusable Coupled Operation</td></tr>
<tr><td>Inline Superclass</td><td>3</td></tr>
<tr><td>Unfold Class</td><td>2</td></tr>
<tr><td>Enumeration to Subclasses</td><td>1</td></tr>
<tr><td>Split Reference by Type</td><td>1</td></tr>
<tr><td>Delete Feature</td><td>8</td><td rowspan="3">Destructor</td></tr>
<tr><td>Make Class Abstract</td><td>2</td></tr>
<tr><td>Specialize Reference</td><td>1</td></tr>
<tr><td>Split Class</td><td>1</td><td>Constructor</td><td>Custom Coupled Operation</td></tr>
</table>

*Refactorings* preserve the set of models that can be built. Consequently, the information contained in models is preserved in response to a refactoring. Rename is used to map the state machine classes and features to activity model classes and features, respectively. Inline Superclass is applied to remove classes for which there is no corresponding class in the new metamodel—e.g. State machine (line 3 in Listing 7.3) and Pseudostate (line 31). Unfold Class is e.g. used to replace the containment reference top in State machine with the features of the sub class CompositeState of its target class (line 10). Enumeration to Subclasses is applied to split PseudoState into a subclass for each literal of the enumeration PseusoStateKind (line 25). Partition Reference is used to partition the reference contents of Partition into references for each subclass of ModelElement (line 34).

**Listing 7.3:** Migrator code generated from the reverse engineered history model

```
// reusable coupled operations
makeAbstract(minuml1.StateMachine, minuml1.ActivityGraph)
inlineSuperClass(minuml1.StateMachine)
rename(minuml1.ActivityGraph, "Activity")
rename(minuml1.Activity.partition, "group")
rename(minuml1.StateVertex, "ActivityNode")
makeAbstract(minuml1.State, minuml1.ActionState)
specializeReference(minuml1.Activity.top, minuml1.CompositeState, 1, 1)
inlineSuperClass(minuml1.State)
unfoldClass(minuml1.Activity.top)
deleteFeature(minuml1.Activity.name_CompositeState)
rename(minuml1.Activity.transitions, "edge")
deleteFeature(minuml1.Activity.incoming)
deleteFeature(minuml1.Activity.outgoing)
rename(minuml1.Activity.subvertex, "node")
rename(minuml1.Partition, "ActivityPartition")
rename(minuml1.CompositeState, "StructuredActivityNode")
rename(minuml1.StructuredActivityNode.subvertex, "node")
rename(minuml1.Transition, "ActivityEdge")
unfoldClass(minuml1.ActivityEdge.guard)
deleteFeature(minuml1.ActivityEdge.name_Guard)
rename(minuml1.BooleanExpression, "OpaqueExpression")
rename(minuml1.ActionState, "OpaqueAction")
deleteFeature(minuml1.OpaqueAction.isDynamic)
enumerationToSubClasses(minuml1.Pseudostate.kind, minuml1)
rename(minuml1.initial, "InitialNode")
rename(minuml1.join, "JoinNode")
rename(minuml1.fork, "ForkNode")
rename(minuml1.junction, "DecisionNode")
rename(minuml1.ActivityEdge.expression, "guard")
inlineSuperClass(minuml1.Pseudostate)
rename(minuml1.ObjectFlowState, "ObjectNode")
rename(minuml1.FinalState, "ActivityFinalNode")
partitionReference(minuml1.ActivityPartition.contents)
rename(minuml1.ActivityPartition.activityEdge, "edges")
rename(minuml1.ActivityPartition.activityNode, "nodes")
deleteFeature(minuml1.ActivityPartition.guard)
deleteFeature(minuml1.ActivityPartition.activity)
deleteFeature(minuml1.ActivityPartition.activityPartition)

// custom coupled operation: Split Class
// metamodel adaptation
activityEdgeClass = minuml1.ActivityEdge
activityEdgeClass.'abstract' = true
newClass(minuml1, "ControlFlow", [minuml1.ActivityEdge], false)
newClass(minuml1, "ObjectFlow", [minuml1.ActivityEdge], false)

// model migration
for(edge in activityEdgeClass.allInstances) {
  if(edge.source.instanceOf(minuml1.ObjectNode) ||
      edge.target.instanceOf(minuml1.ObjectNode)) {
    edge.migrate(minuml1.ObjectFlow)
  } else {
    edge.migrate(minuml1.ControlFlow)
  }
}

// reusable coupled operations
rename(minuml1Package, "minuml2")
minuml1Package = minuml1
minuml1Package.nsPrefix = "minuml2"
minuml1Package.nsURI = "minuml2"
```

*Destructors* decrease the set of models that can be built. Consequently, information may be lost in models in response to a destructor. Delete Feature is used to delete features that are no longer available in the target metamodel, e.g. isDynamic of ActionState. Delete Feature is also applied to delete features that are created by earlier applications of other operations. For instance, the application of Unfold Class leads to more features than are actually part of the new metamodel (lines 11, 13 and 14). Make Class Abstract is used to make classes abstract to be able to apply Inline Superclass. For instance, State is made abstract, and its instances are migrated to ActionState which is renamed to OpaqueAction (line 7). Specialize Reference is applied to specialize the type of top to CompositeState so that we are able to unfold its features into StateMachine (line 8).

*Constructors* increase the set of models that can be built. Consequently, no information is lost in models due to a constructor, but new information can be added to models. Split Class is used to split Transition—which is renamed to ActivityEdge—into ControlFlow and ObjectFlow (lines 41 to 56). If the source or target of an ActivityEdge is an ObjectNode, we migrate to ObjectFlow, otherwise to ControlFlow.

**Solution for Extensions.** While we could implement the first extension, we could not implement the remaining two extensions, since COPE is not intended to bridge several technical spaces, but to provide support for incrementally adapting metamodels within the EMF technical space.

*Alternative object flow state migration semantics.* To implement this extension, we modified the history model for the core task using COPE's refactoring functions (see Section 6.2.2 (*Refactoring the Coupled Evolution*)). Listing 7.4 shows the replacement for the custom coupled operation in Listing 7.3 (lines 41 to 56). Using the function for *Flattening Operations*, we replaced the custom migration to split the class ActivityEdge by a custom migration to build instances for the subclass ObjectFlow (lines 2 to 19 in Listing 7.4). In this custom migration, edges incoming to and outgoing from ObjectNodes are replaced by direct ObjectFlows. After attaching the new custom migration, two primitive operations (make class ActivityEdge abstract, create subclass ControlFlow) remain that have no longer a custom migration attached. Making the class ActivityEdge abstract is breaking and thus requires a migration. Using the function for *Reordering Operations*, we moved the operations to make class ActivityEdge abstract and to create its concrete subclass ControlFlow to after the new custom coupled operation. Using the function for *Replacing Operations*, we replaced the primitive operations to make class ActivityEdge abstract with the reusable coupled operation Make Class Abstract. However, the semantics extension leads to more loss of information than the core migration, as instances of ObjectNode are deleted by the custom coupled operation.

*Concrete syntax.* We would need a concrete syntax built with GMF and not with ArgoUML which is implemented in a different technical space.

*XMI.* We would only need a bridge that generically migrates metamodels and models from XMI 1.2 to EMF.

**Own Evaluation.** We evaluate the solution according to the criteria defined for the case in Table 7.10.

**Listing 7.4:** Update for the extension of the migration semantics

```
...
// custom coupled operation
// metamodel adaptation
newClass(minuml1, "ObjectFlow", [minuml1.ActivityEdge], false)

//model migration
for(on in minuml1.ObjectNode.allInstances) {
  for(i in on.incoming) {
    for(o in on.outgoing) {
      def of = minuml1.ObjectFlow.newInstance()
      on.container.edge.add(of)
      of.source = i.source
      of.target = o.target
    }
  }
  for(i in on.incoming) i.delete()
  for(o in on.outgoing) o.delete()
  on.delete()
}

// reusable coupled operations
newClass(minuml1, "ControlFlow", [minuml1.ActivityEdge], false)
makeAbstract(minuml1.ActivityEdge, minuml1.ControlFlow)
...
```

*Correctness.* The reverse engineered coupled evolution is correct in the sense that it produces the same model as the one provided with the case. Additionally, by classifying coupled operations, one can reason about their effect on existing models. To adapt the provided minimal original metamodel to the evolved metamodel, a number of coupled operations were necessary that may lead to information loss in existing models. This is however not the case for the provided original model, but may be the case for other models conforming to the original metamodel.

*Conciseness.* We showed that most of the coupled evolution can be automated by reusable coupled operations, while only a very small amount (9 lines) of migration code needs to be handcoded. Therefore, the solution can be considered as very concise.

*Clarity.* For users familiar with typical model transformation languages, the coupled evolution may be difficult to understand. The reason is probably that the model migration is modularized along the metamodel adaptation. However, the modularization allows language engineers to understand and reason about each coupled operation separately.

**Evaluation by Reviewers.** Two reviewers had to quantitatively evaluate the submitted solutions according to a number of criteria. Table 7.13 shows the scores that our solution obtained from both reviewers. Our solution obtained good grades for being able to reproduce the solution and to understand the description as well as for the core task. However, we received low marks for not providing a solution to all extensions and for forgetting to discuss the disadvantages of our solution in the description.

The reviewers also qualitatively evaluated the solution with respect to the case criteria. We present the points of each reviewer separately.

**Table 7.13:** Evaluation by reviewers

| Criterion | Scale | Reviewer 1 | Reviewer 2 |
|---|---|---|---|
| Overall rating | -3..3 | 3 (strong accept) | |
| Reviewer's confidence | 1..3 | 3 (high) | 2 (medium) |
| Solution reproducibility | 1..5 | 4 (installation + document + appendix) | |
| Document understandability | 1..5 | 4 (good) | |
| Case specific score for core task | 1..5 | 5 (excellent) | 4 (good) |
| Case specific score for extensions | 1..5 | 3 (fair) | 1 (very poor) |
| Fair discussion of disadvantages of solution | 1..5 | 2 (poor) | 3 (fair) |

*Reviewer 1* approves correctness, as the generated model seems to be correct. In the reviewer's opinion, the approach can also be considered concise, since it describes the coupled evolution of metamodels and models. However, reviewer 1 is surprised by the large number of operation applications, which makes the solution rather complex. In the reviewer's opinion, the migration is basically rather clear, as it is based on coupled operations, which, however, require sufficient familiarity with the library of operations.

*Reviewer 2* thinks that conciseness and clarity are quite good. In the reviewer's opinion, the only point where the solution is not concise is the one operation which requires a custom migration. However, reviewer 2 is more concerned with the correctness of the migration. The reviewer defines two aspects of correctness: First, the approach needs to guarantee that the composition of the operations yields a model transformation from the source to the target metamodel. Second, we need to be able to ensure that the composition of the operations defines the intended model transformation.

**Evaluation by Opponents.** We mention the arguments of both opponents separately.

*Opponent 1* thinks that COPE is well integrated into Eclipse, thus benefiting optimally from Eclipse features such as browse trees, separate views, etc. Even though almost every aspect of the language changes in the case, COPE scales very well to such revolutions in the opponent's opinion, requiring only one custom coupled operation. Opponent 1 believes that this is because the approach is very modular, treating the model migration of different metamodel changes separately from each other. However, the migration of reusable coupled operations is fixed, requiring the specification of a custom coupled operation, in case a different migration semantics is required. Moreover, opponent 1 thinks that additional effort is necessary to learn a new language to specify custom coupled operations.

*Opponent 2* suspects that the performance for model migration is not very good, due to having an interpreted language as well as the sequential execution of the operations. The opponent also is not sure about how to determine the effort to specify the model migration for this kind of approach. While the migration only needs to be specified manually for custom coupled operations, effort is also necessary to apply reusable coupled operations.

**Evaluation by Participants.** From the 9 model transformation tools applied to the

model migration case, COPE made the second place after Epsilon Flock and before GrGen.NET. Besides the overall ranks, Table 7.14 also shows the specific ranks for the different criteria as calculated by the workshop organizers from the 12 received evaluation sheets. COPE obtained high ranks in all case criteria, i.e. also correctness which seems to be the weak point of Flock. Moreover, we received a high rank for the criterion appropriateness, and a medium rank for the criteria extensions and tool maturity.

**Table 7.14:** Evaluation by participants

| Ranks | Flock | COPE | GrGen.NET | Fujaba | Mola | PETE | ATL/ Java | GReTL | RSDS |
|---|---|---|---|---|---|---|---|---|---|
| **Correctness** | 7 | 2 | 2 | 2 | 6 | 1 | 5 | 8 | 9 |
| **Conciseness** | 1 | 2 | 2 | 4 | 5 | 7 | 6 | 8 | 9 |
| **Clarity** | 1 | 2 | 3 | 5 | 4 | 8 | 6 | 7 | 9 |
| **Appropriateness** | 1 | 2 | 3 | 5 | 4 | 7 | 8 | 6 | 9 |
| **Extensions** | 1 | 4 | 4 | 2 | 8 | 7 | 3 | 4 | 9 |
| **Tool Maturity** | 3 | 5 | 2 | 1 | 4 | 6 | 8 | 7 | 9 |
| **Overall** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

### 7.5.5 Study Discussion

All in all, we got positive feedback for COPE in the tool transformation contest. The expert reviewers highlighted the conciseness and clarity of the history model. The opponents pointed out the modularity of the coupled evolution as well as the seamless integration into Eclipse. We also conclude that all the workshop participants were convinced about COPE, as it made the second place due to their evaluations. COPE received a high rank in all the case criteria correctness, conciseness and clarity. Particularly, the high ranks in correctness and conciseness confirm that the requirements of semantics preservation and automation are fulfilled by COPE.

**Correctness.** One reviewer was concerned about the syntactic and semantic correctness of the model migration. The syntactic correctness is ensured by COPE by simultaneously recording the coupled operation and applying it to the metamodel. Even though COPE allows the user to refactor the history model, it provides checks to ensure that the metamodel adaptation is preserved. The semantic correctness is supported by recording the model migration at the same time, when the metamodel is adapted. We believe that the language engineer is most aware about the intention behind the metamodel adaptation, when changing the metamodel.

**Clarity of the Library.** According to one reviewer, the user needs to have knowledge about the operations in the library to profit from reusable coupled operations. However, the barrier of entrance for understanding the library can be lowered by the following techniques: First, the operations in the library need to have consistent names which are similar to refactorings known from object-oriented software

evolution. Second, COPE supports the user to select an operation by only showing those operations in the operation browser that are applicable based on the currently selected element in the metamodel editor. Third, COPE provides a view that interactively shows the documentation for each reusable coupled operation.

**Performance.** One opponent was concerned about the performance of the model migration. The performance is certainly not as good as in the case of the most advanced model transformation tools. However, we believe that performance is not the most important aspect of a tool for model migration. It is more important to reduce the effort for specifying a correct model migration that can be used to automatically migrate models.

### 7.5.6 Threats to Validity

We are aware that our results can be influenced by threats to construct, internal and external validity.

**Construct Validity.** The results might be influenced by the comparison criteria that are defined for the case study. The comparison criteria explicitly defined by the case are rather restricted: correctness, conciseness and clarity. However, due to the high number of submitted solutions, it is quite difficult to deeply evaluate more comparison criteria. Moreover, the correctness criterion is rather weak, since it only requires to correctly migrate a single model. To derive the correct migration from the semantics of both versions of UML is difficult, since UML in general lacks a clear semantics. Consequently, the unclear UML semantics has lead to a number of discussions about the correct model migration in the forum. However, it would have been possible to define multiple test models for the migration.

**Internal Validity.** The results might be influenced by the method we chose for the case study. If the submitters of the migration case also participate in this case, they may have an advantage over the other participants, since they can tailor the migration case to their tool. To mitigate this threat, the cases were reviewed before being accepted and they could be discussed by the participants. Additionally, there is no way to ensure that the participants try out the solution before evaluating it. Consequently, it might be possible that the participants do not evaluate the solution, but rather how the solution is presented by its submitter. To mitigate this threat, reviewers and opponents have to provide an explicit evaluation of the solution: the reviewers by writing a review, and the opponents by giving a statement after the solution is presented.

**External Validity.** The results might be influenced by the fact that we restricted ourselves to a number of model transformation tools and only 1 migration case. Of course, the contest can only take into account model transformation tools for which a solution is submitted. Nevertheless, there have been quite a number of submissions, resulting in 9 accepted solutions. From the 9 used model transformation tools, only 2 are really tailored for model migration: Flock and COPE. However, the chosen migration case also was not typical for model migration, as the pruned metamodel was

more or less completely changed. Consequently, the model migration tools could not really profit from requiring to specify only the difference between both metamodel versions. Nevertheless, they outperformed the model transformation tools in the evaluation by the workshop participants.

## 7.6 Comparison of Model Migration Tools

Several tools have been proposed to build a migration strategy that automates the migration of existing models. However, only few of them participated in the TTC presented in the previous section. Hence, little is still known about the advantages and disadvantages of the tools in different situations. In this section, we thus compare a representative sample of migration tools—AML, COPE, Ecore2Ecore and Epsilon Flock—using common migration examples. The criteria used in the comparison aim to support users in selecting the most appropriate tool for their situation.

### 7.6.1 Study Goal

The study was performed to compare existing model migration tools for the Eclipse Modeling Framework. More specifically, the study was performed to answer the following research questions:

- **RQ1.** *What are the strengths and weaknesses of the different model migration tools?* We apply the tools to common migration examples in order to learn about their strengths and weaknesses.
- **RQ2.** *What is the most appropriate model migration tool for a certain situation?* From the tools' strengths and weaknesses, we synthesize recommendations for choosing a migration tool in a certain situation.

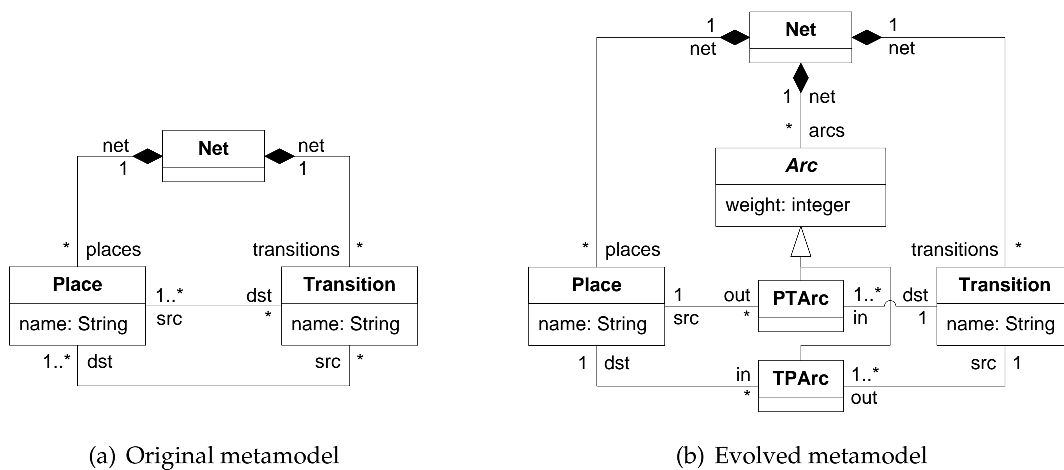
### 7.6.2 Study Object

The comparison is based on the practical application of selected tools to the coupled evolution examples. We first present the examples and then list the selection of tools.

#### Coupled Evolution Examples

To compare migration tools, two examples of coupled evolution were used. The first is a well-known toy example in the model migration literature and was used to test the comparison process, as discussed in Section 7.6.3 (*Study Execution*). The second is a larger example taken from a real-world model-based development project, and was identified as a potentially useful example for coupled evolution case studies in [Herrmannsdoerfer et al., 2009c].

**Petri Nets.** The first example is an evolution of a Petri net metamodel, previously used in [Cicchetti et al., 2008, Garcés et al., 2009, Rose et al., 2010d, Wachsmuth, 2007] to discuss coupled evolution and model migration.




(a) Original metamodel

(b) Evolved metamodel

**Figure 7.16:** Petri nets metamodel evolution

In Figure 7.16(a), a Petri Net comprises Places and Transitions. A Place has any number of source (src) or destination (dst) Transitions. Similarly, a Transition has at least one src and dst Place. In this example, the metamodel in Figure 7.16(a) is to be evolved to support weighted connections between Places and Transitions and between Transitions and Places.

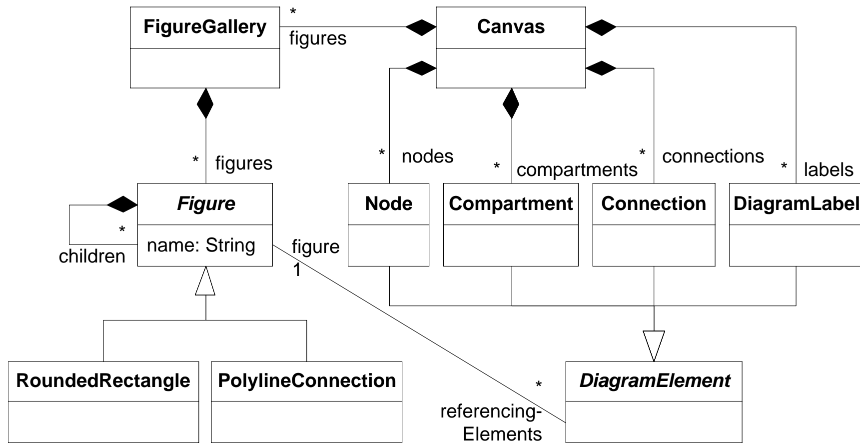
The evolved metamodel is shown in Figure 7.16(b). Places are connected to Transitions via instances of PTArc. Likewise, Transitions are connected to Places via TPArc. Both PTArc and TPArc inherit from Arc, and therefore can be used to specify a weight.

**GMF.** The second example is taken from the Graphical Modeling Framework (GMF) [Gronback, 2009], an Eclipse project for generating graphical editors for models. The development of GMF is model-based and utilizes four domain-specific metamodels. Here, we consider one of those metamodels, gmfgraph, and its evolution between GMF versions 1.0 and 2.0. This metamodel has already been used in the GMF case study in Section 7.2 (*Graphical Modeling Framework*) where we also present its role within GMF.

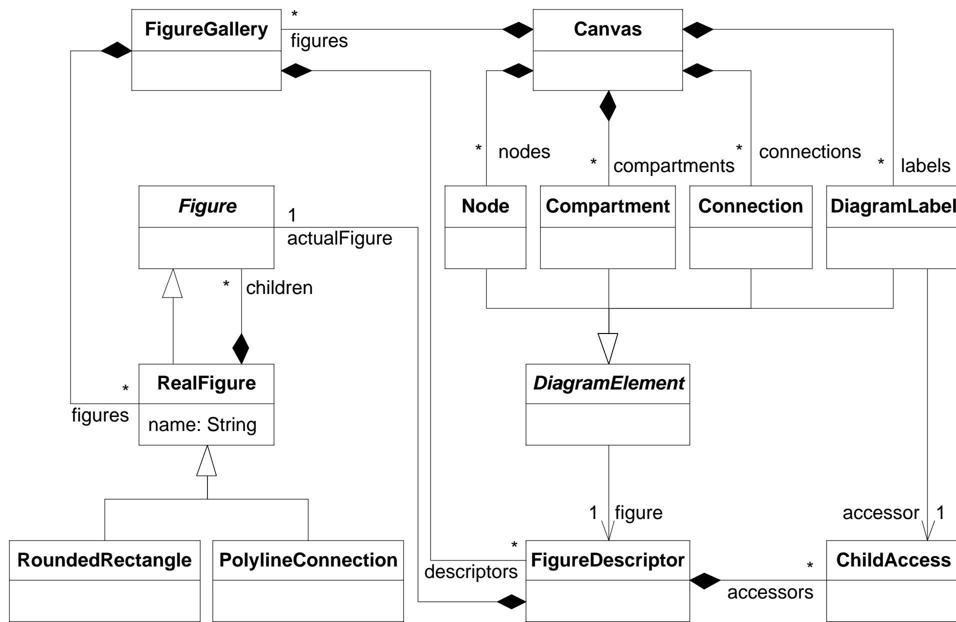
The gmfgraph metamodel (see Figure 7.17(a)) describes the appearance of the generated graphical model editor. The classes Canvas, Figure, Node, DiagramLabel, Connection, and Compartment are used to represent components of the graphical model editor to be generated. The evolution in the gmfgraph metamodel was driven by analyzing the usage of the Figure.referencingElements reference, which relates Figures to the DiagramElements that use them. As described in the gmfgraph documentation[17], the referencingElements reference increased the effort required to reuse figures, a common activity for users of GMF. Furthermore, referencingElements was used only by the GMF code generator to determine whether an accessor should be generated for nested Figures.

In GMF 2.0 (see Figure 7.17(b)), the gmfgraph metamodel was evolved to make reusing figures more straightforward by introducing a proxy [Gamma et al., 1995]

[17]see gmfgraph documentation: `http://wiki.eclipse.org/GMFGraph_Hints`




(a) GMF 1.0




(b) GMF 2.0

**Figure 7.17:** Metamodel gmfgraph

for Figure, termed FigureDescriptor. The original referencingElements reference was removed, and an extra class, ChildAccess, was added to make more explicit the original purpose of referencingElements (accessing nested Figures).

GMF provides a migrating algorithm that produces a model conforming to the evolved gmfgraph metamodel from a model conforming to the original gmfgraph metamodel. In GMF, the migration is implemented using Java. The GMF source code includes two example editors, for which the source code management system contains versions conforming to GMF 1.0 and GMF 2.0. For the comparison of migration tools described in this section, the migrating algorithm and example editors provided by GMF were used to determine the correctness of the migration strategies produced by using each model migration tool.

#### Compared Tools

For the comparison in this section, we selected one tool from each of the three categories—*manual specification*, *operation-based* and *metamodel matching* approaches—described in Section 4.7 (*Modelware*). We included a further tool from the manual specification category, Ecore2Ecore, as it is distributed with EMF. With the exception of COPE, each of these tools is discussed briefly below. Section 7.6.4 (*Study Result*) describes the experiences with using each tool in more detail.

**AtlanMod Matching Language (AML)** [Garcés et al., 2009, Garcés et al., 2009] is a model matching tool, which can be used as a *metamodel matching* migration tool. AML provides heuristics that the user combines to specify a metamodel matching strategy. A migrating ATL transformation is automatically generated by matching original and evolved metamodels.

**Ecore2Ecore** [Hussey and Paternostro, 2006] is a *manual specification* migration tool that is part of EMF. The migration is specified with a mapping model and hand-written Java code. Ecore2Ecore has been used in real-world projects, such as the Eclipse MDT UML2 project[18], to manage coupled evolution.

**Epsilon Flock** [Rose et al., 2010d] (subsequently referred to as Flock) is a *manual specification* migration tool. Flock is a domain-specific transformation language tailored for model migration. In particular, Flock automatically copies from original to migrated model all model elements that have not been affected by metamodel evolution. Flock is built atop Epsilon[19] [Kolovos, 2009], an extensible platform providing inter-operable programming languages for model-based development.

### 7.6.3 Study Execution

The comparison of migration tools was conducted by applying each of the four tools (Ecore2Ecore, AML, COPE and Flock) to the two examples of coupled evolution (Petri nets and GMF). The developers of each tool were invited to participate in the comparison. The authors of COPE and Flock were able to participate fully, while the

[18]see MDT UML2 web site: `http://www.eclipse.org/modeling/mdt/uml2`
[19]see Epsilon web site: `http://www.eclipse.org/gmt/epsilon`

authors of Ecore2Ecore and AML were available for guidance, advice, and to comment on preliminary results. The comparison was conducted in five steps:

1. *Setup:* We identified comparison criteria and assigned the tools to developers. We allocated responsibility for using each tool on the examples to a different person. Because the authors of Ecore2Ecore and AML were not able to fully participate in the comparison, two colleagues experienced in model transformation and migration stood in. To improve the validity of the comparison, each tool was used by someone other than its developer. Other than this restriction, the tools were allocated arbitrarily.
2. *Familiarization:* We used the first example of coupled evolution (Petri nets) to familiarize ourselves with the migration tools and to assess the suitability of the comparison criteria. Table 7.15 summarizes the used comparison criteria. They have been derived from comparisons of languages for model transformation like [Taentzer et al., 2005], [Mens and Van Gorp, 2006], [Czarnecki and Helsen, 2006] and [Grønmo et al., 2009].
3. *Analysis:* The tools were applied to the larger example of coupled evolution (GMF), and experiences were recorded along the application.
4. *Results:* We compiled the experiences by criteria and noted similarities and differences between the tools. The result of this step answers *RQ1*.
5. *Synthesis:* We synthesized, by consensus, guidance for selecting a tool for a certain situation. The result of this step answers *RQ2*.

**Table 7.15:** Summary of comparison criteria

| Name | Description |
|---|---|
| Construction | Ways in which tool supports the development of migration strategies |
| Change | Ways in which tool supports change to migration strategies |
| Extensibility | Extent to which user-defined extensions are supported |
| Reuse | Mechanisms for reusing migration patterns and logic |
| Conciseness | Size of migration strategies produced with tool |
| Clarity | Understandability of migration strategies produced with tool |
| Expressiveness | Extent to which migration problems can be codified with tool |
| Interoperability | Technical dependencies and procedural assumptions of tool |
| Performance | Time taken to execute migration |

### 7.6.4 Study Result

By applying the method described in Section 7.6.3 (*Study Execution*), four model migration tools were compared. This subsection reports similarities and differences of each tool, using nine criteria. Each subsubsection considers one criterion and contains the experiences as reported by the developer using the tool. The complete solutions for the two examples are available online[20].

[20] see GIT repository: `http://github.com/louismrose/migration_comparison`

### Constructing the Migration Strategy

Facilitating the specification and execution of migration strategies is the primary function of model migration tools. In the following, we report the process for and challenges faced in constructing migration strategies with each tool.

**AML.** An AML user specifies a combination of match heuristics from which AML infers a migrating transformation by comparing original and evolved metamodels. Matching strategies are written in a textual syntax, which AML compiles to produce an executable workflow. The workflow is invoked to generate the migrating transformation, codified in the Atlas Transformation Language (ATL) [Jouault and Kurtev, 2006]. Devising correct matching strategies was difficult, as AML lacks documentation that describes the input, output and effects of each heuristic. Papers describing AML—such as [Garcés et al., 2009, Garcés et al., 2009]—discuss each heuristic, but mostly in a high-level manner. A semantically invalid combination of heuristics can cause a runtime error, while an incorrect combination results in the generation of an incorrect migration transformation. However, once a matching strategy is specified, it can be reused for similar cases of metamodel evolution. To devise the matching strategies used for the examples, AML's author provided considerable guidance.

**COPE.** A COPE user applies coupled operations to the original metamodel to form the evolved metamodel. Each coupled operation specifies a metamodel evolution along with a corresponding fragment of the model migration. A history of applied operations is later used to generate a complete migration strategy. As COPE is meant for coupled evolution of models and metamodels, reverse engineering a large metamodel can be difficult. Determining which sequence of operations will produce a correct migration is not always straightforward. To aid the user, COPE allows operations to be undone. To help with the migration process, COPE offers the *Metamodel Convergence* function which utilizes EMF Compare to display the differences between two metamodels. While this was useful, it can, understandably, only provide a list of explicit differences and not the semantics of a metamodel change. Consequently, reverse engineering a large and unfamiliar metamodel is challenging, and migration for the gmfgraph example could only be completed with considerable guidance from the author of COPE.

**Ecore2Ecore.** In Ecore2Ecore, model migration is specified in two steps. In the first step, a graphical mapping editor is used to construct a model that declares basic migrations. In this step, only very simple migrations such as class and feature renaming can be declared. In the next step, the developer needs to use Java to specify a customized parser (resource handler, in EMF terminology) that can parse models that conform to the original metamodel and migrate them so that they conform to the new metamodel. This customized parser exploits the basic migration information specified in the first step and delegates any changes that it cannot recognize to a particular Java method in the parser for the developer to handle. Handling such changes is tedious, as the developer is only provided with the string contents of the unrecognized features and then needs to use low-level techniques—such as data-type checking and conversion, string splitting and concatenation—to address them. Here, it is worth mentioning that Ecore2Ecore cannot handle all migration scenarios and is limited to cases where only a certain degree of structural change has

been introduced between the original and the evolved metamodel. For cases which Ecore2Ecore cannot handle, developers need to specify a custom parser without any support for automated element copying.

**Flock.** In Flock, model migration is specified manually. Flock automatically copies only those model elements which still conform to the evolved metamodel. Hence, the user specifies migration only for model elements which no longer conform to the evolved metamodel. Due to the automatic copying algorithm, an empty Flock migration strategy always yields a model conforming to the evolved metamodel. Consequently, a user typically starts with an empty migration specification and iteratively refines it to migrate non-conforming elements. However, there is no support to ensure that all non-conforming elements are migrated. In the gmfgraph example, completeness could only be ensured by testing with numerous models. Using this method, a model migration can be easily encoded for the Petri net example. For the gmfgraph example whose metamodels are larger, it was more difficult, since there is no tool support for analyzing the changes between original and evolved metamodel.

### Changing the Migration Strategy

Migration strategies can change in at least two ways. Firstly, as a migration strategy is developed, testing might reveal errors which need to be corrected. Secondly, further metamodel changes might require changes to an existing migration strategy.

**AML.** Because AML automatically generates migrating transformations, changing the transformation, for example after discovering an error in the matching strategy, is trivial. To migrate models over several versions of a metamodel at once, the migrating transformations generated by AML can be composed by the user. AML provides no tool support for composing transformations.

**COPE.** As mentioned previously, COPE provides an undo feature, meaning that any incorrect migrations can be easily fixed. COPE stores a history of *releases*—a set of operations that has been applied between versions of the metamodel. Because the migration code generated from the release history can migrate models conforming to any previous metamodel release, COPE provides a comprehensive means for chaining migration strategies.

**Ecore2Ecore.** Migrations specified using Ecore2Ecore can be modified via the graphical mapping editor and the Java code in the custom model parser. Therefore, developers can use the features of the Eclipse Java IDE to modify and debug migrations. Ecore2Ecore provides no tool support for composing migrations, but composition can be achieved by modifying the resource handler.

**Flock.** There is comprehensive support for fixing errors. A migration strategy can easily be re-executed using a launch configuration, and migration errors are linked to the line in the migration strategy that caused the error to occur. If the metamodel is further evolved, the original migration strategy has to be extended, since there is no explicit support to chain migration strategies. The full migration strategy may need to be read to know where to extend it.

### Extensibility

The fundamental constructs used for specifying migration in COPE and AML (operations and matching heuristics, respectively) are extensible. Flock and Ecore2Ecore use a more imperative (rather than declarative) approach, and as such do not provide extensible constructs.

**AML.** An AML user can specify additional matching heuristics. This requires understanding of AML's domain-specific language for manipulating the data structures from which migrating transformations are generated.

**COPE** provides the user with a large number of operations. If there is no applicable operation, a COPE user can write their own operations using an in-place transformation language embedded into Groovy[21].

### Reuse

Each migration tool captures patterns that commonly occur in model migration. In the following, we consider the extent to which the patterns captured by each tool facilitate reuse between migration strategies.

**AML.** Once a matching strategy is specified, it can potentially be reused for further cases of metamodel evolution. Matching heuristics provide a reusable and extensible mechanism for capturing metamodel change and model migration patterns.

**COPE.** An operation in COPE represents a commonly occurring pattern in coupled evolution. Each operation captures the metamodel evolution and model migration steps. Custom operations can be written and reused.

**Ecore2Ecore.** Mapping models cannot be reused or extended in Ecore2Ecore, but as the custom model parser is specified in Java, developers can decompose it into reusable parts, some of which can potentially be reused in other migrations.

**Flock.** A migration strategy encoded in Flock is modularized according to the classes whose instances need migration. There is support to reuse code within a strategy by means of operations with parameters and across strategies by means of imports. Reuse in Flock captures only migration patterns, and not the higher level coupled evolution patterns captured in COPE or AML.

### Conciseness

A concise migration strategy is arguably more readable and requires less effort to write than a verbose migration strategy. In the following, we comment on the conciseness of migration strategies produced with each tool, and report the lines of code (without comments and blank lines) used.

**AML.** 117 lines were automatically generated for the Petri nets example. 563 lines were automatically generated for the gmfgraph example, and a further 63 lines of code were added by hand to complete the transformation. Approximately 10 lines of

[21]see Groovy web site: `http://groovy.codehaus.org/`

the user-defined code could be removed by restructuring the generated transformation.

**COPE** requires the user to apply operations. Each operation application generates one line of code. The user may also write additional migration code. For the Petri net example, 11 operations were required to create the migrator and no additional code. We migrated the gmfgraph example using 76 operations and 73 lines of additional code.

**Ecore2Ecore.** As discussed above, handling changes that cannot be declared in the mapping model is a tedious task and involves a significant amount of low level code. For the Petri nets example, the Ecore2Ecore solution involved a mapping model containing 57 lines of (automatically generated) XMI and a custom hand-written resource handler containing 78 lines of Java code.

**Flock.** 16 lines of code were necessary to encode the Petri nets example, and 140 lines of code were necessary to encode the gmfgraph example. In the gmfgraph example, approximately 60 lines of code implement missing built-in support for rule inheritance, even after duplication was removed by extracting and reusing a subroutine.

### Clarity

Because migration strategies can change and might serve as documentation for the history of a metamodel, their clarity is important. In the following, we report on aspects of each tool that might affect the clarity of migration strategies.

**AML.** The AML code generator takes a conservative approach to naming variables, to minimize the chances of duplicate variable names. Hence, some of the generated code can be difficult to read and hard to reuse if the generated transformation has to be completed by hand. When a complete transformation can be generated by AML, clarity is not as important.

**COPE.** Migration strategies in COPE are defined as a sequence of operations. The release history stores the set of operations that have been applied, so the user is clearly able to see the changes they have made, and find where any issues may have been introduced.

**Ecore2Ecore.** The graphical mapping editor provided by Ecore2Ecore allows developers to have a high-level visual overview of the simple mappings involved in the migration. However, migrations expressed in the Java part of the solution can be far more obscure and difficult to understand as they mix high-level intention with low-level string management operations.

**Flock** clearly states the migration strategy from the source to the target metamodel. However, the boilerplate code necessary to implement rule inheritance slightly obfuscates the real migration code.

### Expressiveness

Migration strategies are easier to infer for some categories of metamodel change than others [Gruschko et al., 2007]. In the following, we report on the ability of each tool

to migrate the examples considered in this comparison.

**AML.** A complete migrating transformation could be generated for the Petri nets example, but not for the gmfgraph example. The latter contains examples of two complex changes that AML does not currently support[22]. Successfully expressing the gmfgraph example in AML would require changes to at least one of AML's heuristics. However, AML provided an initial migration transformation that was completed by hand. In general, AML cannot be used to generate complete migration strategies for coupled evolution examples that contain *metamodel-specific coupled changes*, according to the classification introduced in Section 3.2 (*Classification of Metamodel Changes*).

**COPE.** The expressiveness of COPE is defined by the set of operations available. The Petri net example was migrated using only built-in operations. The gmfgraph example was migrated using 76 built-in operations and 2 user-defined migration actions. Custom migration actions allow users to specify any migration strategy.

**Ecore2Ecore.** A complete migration strategy could be specified for the Petri nets example, but not for the gmfgraph example. The developers of Ecore2Ecore have advised that the latter involves significant structural changes between the two versions and recommended implementing a custom model parser from scratch.

**Flock.** Since Flock extends EOL, it is expressive enough to encode both examples. However, Flock does not provide an explicit construct to copy model elements and thus it was necessary to call Java code from within Flock for the gmfgraph example.

#### Interoperability

Migration occurs in a variety of settings with differing requirements. In the following, we consider the technical dependencies and procedural assumptions of each tool, and seek to answer questions such as: "Which modeling technologies can be used?" and "What assumptions does the tool make on the migration process?"

**AML** depends only on ATL, while its development tools also require Eclipse. AML assumes that the original and target metamodels are available for comparison, and does not require a record of metamodel changes. AML can be used with either Ecore (EMF) or KM3 metamodels.

**COPE** depends on EMF and Groovy, while its development tools also require Eclipse and EMF Compare. COPE does not require both the original and target metamodels to be available. When COPE is used to create a migration strategy after metamodel evolution has already occurred, the metamodel changes must be reverse engineered. To facilitate this, the target metamodel can be used with the convergence function. COPE targets EMF and does not support other modeling technologies.

**Ecore2Ecore** depends only on EMF. Both the original and the evolved versions of the metamodel are required to specify the mapping model with the Ecore2Ecore development tools. Alternatively, the Ecore2Ecore mapping model can be constructed programmatically and without using the original metamodel[23]. Unlike the other tools considered, Ecore2Ecore does not require the original metamodel to be available in

[22] see public communication with the author of AML: `http://www.eclipse.org/forums/index.php?t=rview&goto=526894#msg_526894If`

[23] private communication with Marcelo Paternostro, an Ecore2Ecore developer

the workspace of the metamodel user.

**Flock** depends on Epsilon and its development tools also require Eclipse. Flock assumes that the original and target metamodels are available for encoding the migration strategy, and does not require a record of metamodel changes. Flock can be used to migrate models represented in EMF, MDR, XML and Z (CZT), although we only encoded a migration strategy for EMF metamodels in the presented examples.

### Performance

The time taken to execute model migration is important, particularly once a migration strategy has been distributed to metamodel users. Ideally, migration tools will produce migration strategies whose execution time is quick and scales well with large models.

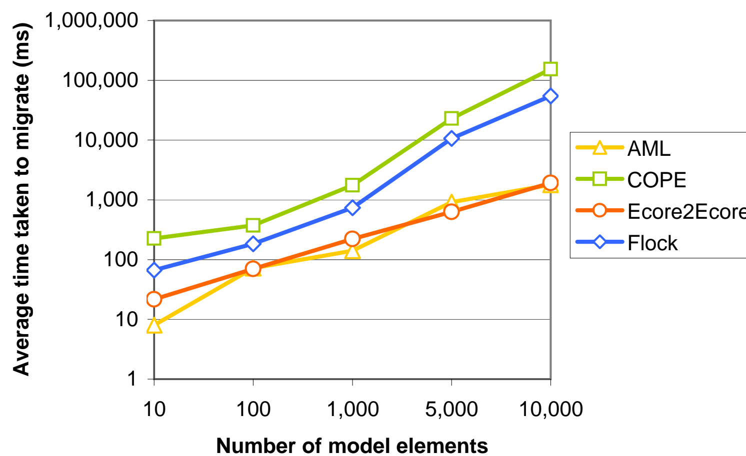



**Figure 7.18:** Performance comparison of the migration tools

To measure performance, we produced Petri net models with a random generator, varying their size. Figure 7.18 shows the average time taken by each tool to execute migration across 10 repetitions for models of different sizes. Note that the Y axis has a logarithmic scale. The results indicate that, for the Petri nets coupled evolution example, AML and Ecore2Ecore execute migration significantly more quickly than COPE and Flock, particularly when the model to be migrated contains more than thousand model elements. Figure 7.18 indicates that, for the Petri nets coupled evolution example, Flock executes migration between two and three times faster than COPE, although we found out that turning off validation causes COPE to perform similarly to Flock.

## 7.6.5 Study Discussion

**RQ1.** *What are the strengths and weaknesses of the different model migration tools?* The comparison results highlight the similarities and differences between a representative sample of model migration approaches. In this subsection, the differences are

used to consider which tools are better suited to particular model migration situations.

COPE captures coupled evolution patterns (which apply to both model and metamodel), while Ecore2Ecore, AML and Flock capture only model migration patterns (which apply just to models). Because of this, COPE facilitates a greater degree of reuse in model migration than other approaches. However, the order in which the user applies patterns with COPE impacts on both metamodel evolution and model migration, which can complicate pattern selection particularly when a large amount of evolution occurs at once. The reusable coupled evolution patterns in COPE make it well suited to migration problems, in which metamodel evolution is frequent and in small steps.

Flock, AML and Ecore2Ecore are preferable to COPE when metamodel evolution has occurred before the selection of a migration approach. Because of its use of coupled evolution patterns, we conclude that COPE is better suited to forward rather than reverse engineering.

Through its convergence function and integration with the EMF metamodel editor, COPE facilitates metamodel analysis that is not possible with the other approaches considered in this section. COPE is well-suited to situations in which measuring and reasoning about coupled evolution is important.

In situations where migration involves modeling technologies other than EMF, AML and Flock are preferable to COPE and Ecore2Ecore. AML can be used with models represented in KM3, while Flock can be used with models represented in MDR, XML and CZT. Via the connectivity layer of Epsilon, Flock can be extended to support further modeling technologies.

There are situations in which Ecore2Ecore or AML might be preferable to Flock and COPE. For large models, Ecore2Ecore and AML might execute migration significantly more quickly than Flock and COPE. Ecore2Ecore is the only tool that has no technical dependencies (other than a modeling framework). In situations where migration must be embedded in another tool, Ecore2Ecore offers a smaller footprint than other migration approaches. Compared to the other considered approaches, AML automatically generates migration strategies with the least guidance from the user.

Despite these advantages, Ecore2Ecore and AML are unsuitable for some types of migration problem, because they are less expressive than Flock and COPE. Specifically, changes to the containment of model elements typically cannot be expressed with Ecore2Ecore and changes that are classified as *metamodel-specific* in Section 3.2 (*Classification of Metamodel Changes*) cannot be expressed with AML. Because of this, it is important to investigate metamodel changes before selecting a migration tool. Furthermore, it might be necessary to anticipate which types of metamodel change are likely to arise before selecting a migration tool. Investing in one tool to discover later that it is no longer suitable causes wasted effort.

**RQ2.** *What is the most appropriate model migration tool for a certain situation?* Some preliminary recommendations and guidelines in choosing a migration tool were synthesized from the presented results and are summarized in Table 7.16.

**Table 7.16:** Summary of tool selection advice (tools are ordered alphabetically)

| Requirement | Recommended Tools |
|---|---|
| Frequent, incremental coupled evolution | COPE |
| Reverse engineering | AML, Ecore2Ecore, Flock |
| Modeling technology diversity | Flock |
| Quicker migration for larger models | AML, Ecore2Ecore |
| Minimal dependencies | Ecore2Ecore |
| Minimal hand-written code | AML, COPE |
| Minimal guidance from user | AML |
| Support for metamodel-specific migrations | COPE, Flock |

### 7.6.6 Threats to Validity

We discuss our results with respect to construct, internal and external validity.

**Construct Validity.** The results might be influenced by the comparison criteria we chose for the case study. To compare the model migration tools, we considered a number of criteria that we deem important for comparing these kinds of tools and that can be evaluated with the presented method. We are aware that there are other criteria—like productivity, usability or learnability—which might be important for comparing model migration tools. However, drawing conclusions about productivity, usability and learnability is challenging with the employed method due to the subjective nature of these characteristics. We envisage that a comprehensive user study (with hundreds of users) would likely provide better results than could be achieved using the presented method.

**Internal Validity.** The results might be influenced by the method we chose for the case study. For each of the five steps explained in Section 7.6.3 (*Study Execution*), we mention possible threats to internal validity. In step 1, we might have chosen criteria that cannot be applied to reasonably compare model migration tools. To mitigate this threat, we first applied the criteria to the Petri net example in step 2 to assess their suitability. The Petri net example used in step 2 might not provide enough variation to completely familiarize the users with the tools. However, we allowed the tool users to ask the tool developers for assistance when applying the tools to the gmfgraph example. In step 3, the users might have understood the criteria differently, when assessing the model migration tools. To mitigate this threat, we compared and consolidated the different assessments in step 4 and 5.

**External Validity.** The results might be influenced by the fact that we restricted ourselves to one test user per tool and to only 2 migration scenarios. Thereby, the results might not be transferable to other users or to other migration scenarios. The transferability to other users may be affected by the knowledge of the single user about the model migration tool. To mitigate this threat, we chose for each tool a user that was not familiar with the tool before. The transferability to other migration scenarios may be affected by the special properties of the chosen scenarios. However, the scenarios

are a well-known research example (petri nets) and a scenario taken from a real-life metamodel evolution (gmfgraph).

## 7.7 Summary

In this chapter, we have presented six case studies that applied COPE to automate the real-world coupled evolution of metamodels and models. These case studies helped to evaluate, refine and improve the approach that we have chosen for COPE. We subsequently summarize the three categories reverse engineering, forward engineering and comparison case studies. For each of these categories, we have performed two case studies.

**Reverse Engineering** case studies recover the coupled evolution from existing metamodel versions, after the metamodel has been adapted. First, we have reverse engineered the coupled evolution of the GMF Generator and PCM metamodels (Section 7.1 (*GMF Generator Model and Palladio Component Model*)). Through this case study, we have confirmed that most of the coupled evolution can be covered by reusable coupled operations and that the complete metamodel adaptation can be captured by a sequence of coupled operations. Second, we have recovered and analyzed the coupled evolution of all four GMF metamodels (Section 7.2 (*Graphical Modeling Framework*)). This case study also showed that metamodels evolve not only due to user requests, but also due to technological changes, that other artifacts need to be migrated in response to metamodel evolution, and that language evolution is similar to software evolution. While COPE targets forward engineering the coupled evolution, the reverse engineering case studies have been performed to demonstrate the applicability of COPE in practice. These case studies helped to build up and test the library of reusable coupled operations, and gave rise to a set of functions that support reverse engineering the coupled evolution from a sequence of metamodel versions.

**Forward Engineering** case studies use COPE to record the coupled evolution, while the metamodel needs to be adapted. First, we have used COPE to forward engineer the coupled evolution of the Quamoco metamodel (Section 7.3 (*Quamoco Quality Metamodel*)). This case study revealed that, while more changes occur with appropriate tool support for model migration, coupled evolution is not much different between forward and reverse engineering case studies—in terms of both automatability of model migration and reasons for language changes. Second, we have applied COPE to migrate Unicase models and their changes in response to metamodel evolution (Section 7.4 (*Unicase Unified Model*)). Through this case study, we showed that COPE's language to specify migration can also be used to encode the migration of other artifacts like the changes recorded by the Unicase tool. Consequently, these case studies showed that COPE effectively supports the development of a correct model migration by incrementally applying coupled operations. Moreover, the existing library of reusable coupled operations proved to be complete enough to cover most of the coupled evolution, thereby significantly reducing the development effort.

**Comparison** case studies compare COPE to other model transformation or migra-

tion tools. First, we have participated with COPE in the model migration case of the Transformation Tool Contest (Section 7.5 (*Transformation Tool Contest*)). This case study confirmed that COPE is better suited to specify model migration than model transformation languages, and that it provides a perfect compromise for ensuring conciseness and correctness of a model migration. Second, we have compared a number of model migration tools for EMF, together with the developers of the other tools (Section 7.6 (*Comparison of Model Migration Tools*)). This case study shows that COPE is well suited for frequent, incremental coupled evolution, when minimal handwritten migration code is desired, and when support for metamodel-specific migrations is required. The comparisons also revealed the weaknesses of COPE: To profit from the reduction of effort, the users need to have knowledge about the library of reusable coupled operations, and the modularization of the migration into operations is not as performant as the specification of a single transformation. Consequently, the comparisons helped to identify areas for further improvement of the approach.

Chapter 8

# Beyond Model Migration: Evolutionary Metamodeling

In two case studies, we were using our approach to support the evolutionary development of modeling languages. Until now, our approach focused on reducing the effort for model migration in response to metamodel adaptation. However, we have already seen through a case study that the operation-based approach can be extended to also migrate other artifacts like e.g. model changes. In this chapter, we want to show that the operation-based approach can support a number of other tasks that are important when evolving a modeling language. More specifically, we developed methods to support the following two tasks:

1. Before adapting the metamodel, we need to identify changes that improve the modeling language. We can identify possible changes by analyzing the models built with a metamodel, and recommend operations to perform these changes.
2. After adapting the metamodel, we may also need to adapt the simulator or code generator of the modeling language that implements the language semantics. To support semantics-preserving model migration, we can extend the operations to also adapt the interpreters of the modeling language.

The two methods have been published in [Herrmannsdoerfer et al., 2010a] and [Herrmannsdoerfer and Koegel, 2010b].

**Contents**


**8.1 The Process of Evolutionary Metamodeling . . . . . . . . . . . . . 232**
**8.2 Metamodel Usage Analysis for Identifying Metamodel Improvements . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 236**
**8.3 Towards Semantics-Preserving Model Migration . . . . . . . . . . 254**
**8.4 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 263**


Section 8.1 (*The Process of Evolutionary Metamodeling*) gives an overview over the tasks for the evolutionary development of modeling languages. In Section 8.2 (*Metamodel Usage Analysis for Identifying Metamodel Improvements*), we propose an approach to mine metamodel improvements from the models built with the meta-

model, and demonstrates the usefulness of the approach by means of an extensive case study. Section 8.3 (*Towards Semantics-Preserving Model Migration*) enables semantics-preserving model migration by extending the operations to also adapt the interpreter of the modeling language. Finally, we conclude this chapter with Section 8.4 (*Summary*).

## 8.1 The Process of Evolutionary Metamodeling

The results of the case studies indicate that COPE is especially suited for the evolutionary development and maintenance of modeling languages. We claim that a good modeling language is hard to obtain by an upfront design, but rather has to be developed by an evolutionary process. A version of a modeling language is defined and deployed to obtain feedback from the language users, which again may lead to a new version.

In this section, we define a systematic process in order to support the evolutionary development of modeling languages. The process is based both on our experiences from the case studies as well as on the process of database and software refactoring—defined in [Ambler and Sadalage, 2006] and [Mens and Tourwé, 2004], respectively. Figure 8.1 illustrates the activities of the process and their interplay as a UML activity diagram. We focus on the iterations from one language version to the next and not on the development of the initial language version. In the following, we explain the different activities in more detail.

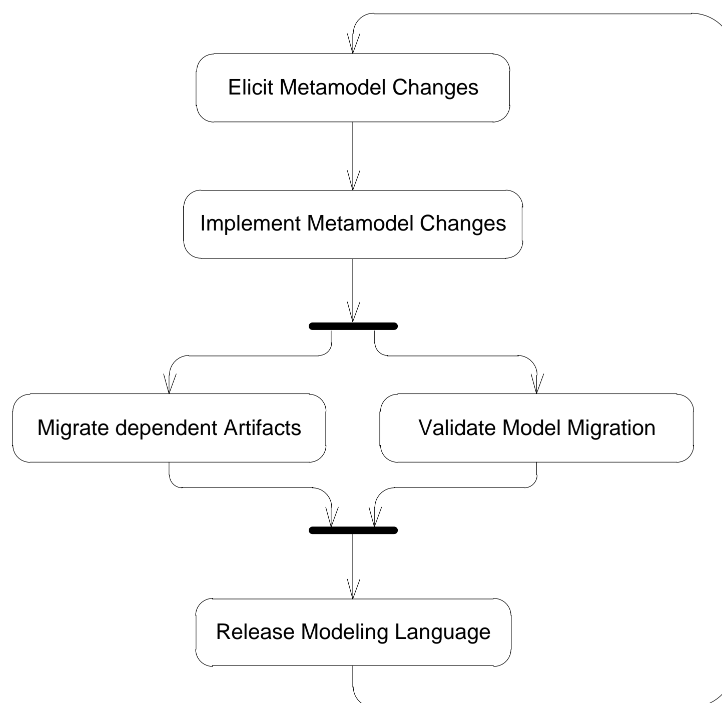



**Figure 8.1:** Evolutionary metamodeling process

### 8.1.1 Elicit Metamodel Changes

Before a new iteration of the process can start, the language engineer needs to identify changes that need to be performed to the metamodel. These changes either introduce new language features or improve the maintainability or usability of the modeling language. To identify metamodel changes, the language engineer can use the following techniques: he or she can improve the maintainability by analyzing the metamodel, improve the usability by analyzing the models built with the metamodel, or introduce new language features by getting feedback from the language users.

**Metamodel Analysis.** The idea behind many of the coupled operations is to improve the maintainability of the metamodel. As a consequence, we could identify metamodel elements to which these coupled operations can be applied. For instance, we could search for redundant features in sibling classes, as they can be pulled up into their common super class. However, when analyzing only the metamodel, we can only identify refactorings, i.e. changes that preserve the expressiveness of the modeling language.

**Model Analysis.** Often, modeling languages are not completely used by the language users and thus could be restricted to improve its usability. To identify restrictions of the metamodel, we could analyze the models built with the metamodel. For instance, we could search for features that are not used in the models and remove them from the metamodel. In Section 8.2 (*Metamodel Usage Analysis for Identifying Metamodel Improvements*), we refine this technique by defining and evaluating a number of such usage patterns. However, when analyzing only the model, we can only identify destructors, i.e. changes that reduce the expressiveness of the modeling language.

**Feedback from Language Users.** To also identify extensions of the modeling language, we need to get feedback from the language users about which new features they need. Feedback from the language users can for instance be obtained by providing a forum, by performing a workshop or by conducting a survey. If the modeling language has an extension mechanism, we could also get the feedback by analyzing models using this extension mechanism. By getting feedback from the language users, we can also identify constructors, i.e. changes that increase the expressiveness of the modeling language.

### 8.1.2 Implement Metamodel Changes

After identifying the metamodel changes, the language engineer has to choose the coupled operations to implement them. To choose the suitable coupled operations, the language engineer has to consider the impact on the modeling language and on existing models (see Section 5.1.6 (*Classification of Coupled Operations*)) as well as the automatability of the reconciling model migration (see Section 5.1 (*COPE in a Nutshell*)).

**Language Preservation.** The language engineer can choose the operations based on

how the changes impact the modeling language. If the changes require to preserve the modeling language, the language engineer should choose refactorings. If they require to extend or restrict the modeling language, he or she should choose constructors or destructors, respectively.

**Model Preservation.** The language engineer can also choose the operations based on how the changes should impact existing models. If the changes should not impact the existing models, the language engineer can only use model-preserving operations. However, this often restricts the way in which the languages can be changed, thus maybe leading to overly complex metamodels. If the changes can impact the existing models, but no information should be lost in these models, the language engineer should only use safely model-migrating operations. If the language engineer intends to delete information, he or she can also use unsafely model migrating operations.

**Automatability.** When choosing the operations to implement the language changes, the language engineer should also consider the automatability of the model migration. The language engineer should first try to find appropriate reusable coupled operations in the library (see Section 5.2 (*Library of Reusable Coupled Operations*)) in order to reduce the effort for model migration. The criteria mentioned above can be used to reduce the number of possible operations. If the changes cannot be implemented by reusable coupled operations, the language engineer has to implement custom coupled operations. If information is required from the language user during migration, the custom coupled operation can be made interactive (see Section 5.4.3 (*Coping with Model-Specific Migration*)).

### 8.1.3 Migrate dependent Artifacts

In response to adaptation of the metamodel, we do not only need to migrate models, but we also may need to adapt the definitions of the other constituents of a modeling language, as they depend on the metamodel. Besides the abstract syntax, the language engineer may need to adapt the definition of the concrete syntax and semantics.

**Abstract Syntax.** From the metamodel, an API is often generated for accessing models built with the metamodel. The language engineer can refine the API by implementing additional constraints for validation as well as derived features that facilitate model access. When the metamodel is adapted, the API can be adapted by simply regenerating it. However, the constraints and derived features may need to be adapted to correctly use the adapted API.

**Concrete Syntax.** The textual, diagrammatic or tabular concrete syntax is defined based on the metamodel defining the abstract syntax. From the concrete syntax definition, the language engineer can ideally generate an editor supporting the concrete syntax. When the metamodel is adapted, the concrete syntax definition may also need to be adapted. Finally, the language engineer needs to regenerate the editor, maybe needing to adapt customizations of the generated code.

**Semantics.** The semantics of a modeling language is usually defined as a code generator or simulator based on the abstract syntax. To implement a code generator or simulator, languages for model transformation are often applied. Since these model transformations are defined based on the metamodel, they may need to be adapted, when the metamodel changes. The model migration and the adaptation of the semantics definition need to be consistent to ensure semantics preservation.

### 8.1.4 Verify Model Migration

Before models are migrated, the language engineer needs to ensure that the defined model migration is correct. There are different levels of correctness: syntax preservation, information preservation and semantics preservation.

**Syntax Preservation.** The syntax of a model is preserved if the model migration transforms it to a model conforming to the adapted metamodel. Syntax preservation is important to be able to completely load the model with the editor, after the metamodel has been adapted. To ensure syntax preservation, the language engineer can either extensively test the model migration or prove it for the defined model migration. Due to the modularization of the model migration, he or she can prove syntax preservation separately for each coupled operation. For reusable coupled operations, we can show syntax preservation independently of the metamodel.

**Information Preservation.** Syntax preservation can be easily ensured by deleting model elements that no longer conform to the adapted metamodel. However, deleting model elements leads to loss of information in the model. The information contained in a model is preserved if the model migration has a safe inverse, which can transform the migrated model back to the original model. While the language engineer can use the inverse relationship defined by the library to show reversability for reusable coupled operations, information preservation has to be proven manually for custom coupled operations.

**Semantics Preservation.** Semantics preservation does not only require that the information contained in a model is preserved, but also that the meaning of the model is preserved. It is difficult to ensure semantics preservation without an explicit definition of the semantics. To ensure semantics preservation, the model migration and the adaptation of the semantics definition need to be consistent. Again, due to the modularization of the model migration, we can show semantics preservation for each coupled operation separately. In Section 8.3 (*Towards Semantics-Preserving Model Migration*), we examine an approach to extend reusable coupled operations with an appropriate adaptation of the semantics definition in order to constructively ensure semantics preservation.

### 8.1.5 Release Modeling Language

After the changes have been performed, the language engineer can release the new version of the modeling language. The language engineer has to make the new lan-

guage version as well as the model migration available to the language user. Then, the language users can update the modeling language and migrate their models to the new version.

**Update of Modeling Language.** Ideally, there is an update mechanism to make the new version of the modeling language available to the language user. The language user can then update the modeling language to the new version using this update mechanism. Then, the language user can use the new features that have been added to the modeling language or benefit from the language improvements that have been made.

**Model Migration.** Finally, the language users need to migrate the existing models so that they can be used with the evolved modeling language. Ideally, the migration is performed with the migrator that is generated from the history model and that is deployed with the new version of the updated editor. In case of interactive custom coupled operations, the language user is required answer the questions, before the migration can complete.

## 8.2 Metamodel Usage Analysis for Identifying Metamodel Improvements

Modeling languages raise the abstraction level at which software is built by providing a set of constructs tailored to the needs of their users [Karsai et al., 2009]. Metamodels define their constructs and thereby reflect the expectations of the language engineers about the use of the language. In practice, language users often do not use the constructs provided by a metamodel as expected by language engineers [Kelly and Pohjonen, 2009, Paige et al., 2000]. In this section, we advocate that insights about how constructs are used can offer language engineers useful information for improving the metamodel. We define a set of usage and improvement patterns to characterize the use of the metamodel by the built models. We present our experience with the analysis of the usage of seven metamodels (EMF, GMF, Unicase) and a large corpus of models. Our empirical investigation shows that we identify mismatches between the expected and actual use of a language which are useful for metamodel improvements.

### 8.2.1 Templates for defining Usage Analyses

Our ultimate goal is to recommend metamodel changes to improve the usability of the language. In case the language engineers have access to a relevant set of models built with the language, they can take advantage of this information and learn about how the language is used by investigating the built models. Once the information about the actual use of the metamodel is available, the metamodel can be improved along two main directions. First, metamodel constructs that are not used in models can be safely removed without affecting existing models. By restricting the metamodel, we can simplify the language for both language users and engineers. Second, metamodel constructs that are used to encode constructs currently not available in

the language can be lifted to first class constructs. By enriching the metamodel with the needed constructs, we support the users to use the language in a more direct manner.

**Collecting Usage Data.** Before we can identify metamodel improvements, we need to collect usage data from the models built with the metamodel. This usage data collection has to fulfill three requirements:

- *Significance:* We need to collect data from a significant number of models built with the metamodel. In the best case, we should collect usage data from every built model. When this is not possible, we should analyze a significant number of models to be sure that the results of our analyses are relevant and can be generalized for the actual use of the language. Generally, the higher the ratio of the existing models that are analyzed, the more relevant our analyses.
- *Privacy:* We need to collect the appropriate amount of data necessary for identifying metamodel improvements. If we collect too much data, we might violate the intellectual property of the owners of the analyzed models. If we collect too few data, we might not be able to extract meaningful information from the usage data.
- *Composability:* The usage data from individual models needs to be collected in a way that it can be composed without losing information.

To specify the collection of usage data, we employ the following template:

*Context:* the kind of metamodel element for which the usage data is collected. The context can be used as pattern to apply the usage data collection to metamodel elements of the kind.

*Confidence:* the number of model elements from which the usage data is collected. The higher this number, the more confidence can we have in the collected data. In the following, we say that we are not confident if this number is zero, i.e. we do not have usage data that can be analyzed.

*Specification:* a function to specify how the usage data is collected. There may be different result types for the data that is collected. In the following, we use numbers and functions that map elements to numbers.

**Analyzing Usage Data.** To identify metamodel improvements, we need to analyze the usage data collected from the models. The analysis is based on an expectation that we have for the usage data. If the expectation about the usage of a metamodel construct is not fulfilled, that construct is a candidate to be improved. To specify expectations and the identification of metamodel improvements from these expectations in the following, we employ the following template:

*Expectation:* a boolean formula to specify the expectation that the usage data needs to fulfill. If the formula evaluates to true, the expectation is fulfilled; otherwise we can propose an improvement. Certain expectations can be automatically derived from the metamodel, e.g. we expect that a non-abstract class is used in models. Other expectations can only be defined manually by the language engineer, e.g. that certain classes are more often used than other classes. In the

following, we focus mostly on expectations that can be automatically derived from the metamodel, as they can be applied to any metamodel without additional information.

*Improvement:* the metamodel changes that can be recommended if the expectation is not fulfilled. The improvement is specified as operations that can be applied on the metamodel. As described above, the improvements can consist of the restriction of the language, or the addition of new constructs. If we have collected data from all models built with a metamodel, we can also be sure that the restrictions can be safely applied, i.e. without breaking the existing models.

### 8.2.2 Towards a Catalog of Usage Analyses

In this section, we present a catalog of analyses of the usage of metamodels. Each subsection presents a category of analyses, each analysis being essentially a question about how the metamodel is actually used. We use the templates defined in Section 8.2.1 (*Templates for defining Usage Analyses*) to define the analyses in a uniform manner. This catalog of analyses is by no means complete, it rather represents a set of basic analyses. We only define analyses that are used in Section 8.2.7 (*Study Result*) as part of the study.

**Conventions.** To define our analyses, we use formula based on the E-MOF metametamodel defined in Section 2.2.4 (*Complete E-MOF Metametamodel*).

*Metamodel.* Let $mm = (N_{mm}, E_{mm}, src_{mm}, tgt_{mm}, lab_{mm})$ be a metamodel that conforms to the metametamodel. To access the metamodel, the classes defined by the metametamodel can be interpreted as sets—e.g. *Class* denotes the set of classes defined by a metamodel:

$$Class := \{x \in N_{mm} \mid isKindOf(x, \textsf{Class})\}$$

Additionally, the features can be interpreted as navigation functions on the metamodel—e.g. $c.abstract$ returns whether a class $c \in Class$ is abstract:

$$c.abstract = v \in N_m :\Leftrightarrow Edge(c, v, \textsf{abstract})$$

$PV(t)$ denotes the possible values of a primitive type $t \in PrimitiveType$.

*Model.* Let $m = (N_m, E_m, src_m, tgt_m, lab_m)$ be a model that conforms to metamodel $mm$. Objects are model nodes that are instances of classes:

$$Object := \{o \in N_m \mid \exists c \in Class : isInstanceOf(o, c)\}$$

To access models, we require an object $o \in Object$ of a class $c \in Class$ to provide two methods. The method $o.get(f)$ returns the value of a feature $f \in Feature$ for a certain object $o \in Object$:

$$o.get(f) := \{v \in N_m \mid Edge(o, v, f)\}$$

The value is returned as a set even in the case of single-valued features to simplify the formulas. The method $o.isSet(f)$ returns true if the value of a feature $f \in Feature$ is set for a certain object $o \in Object$:

$$o.isSet(f) :\Leftrightarrow o.get(f) \neq \emptyset \wedge o.get(f) \neq \{f.defaultValue\}$$

A feature is set if and only if the value of the feature is different from the empty list and different from the *default value*, in case the feature is an attribute.

### Class Usage Analysis

If metamodels are seen as basis for the definition of the syntax of languages, a non-abstract class represents a construct of the language. Thereby, the measure in which a language construct is used can be investigated by analyzing the number of objects $CU(c)$ of the non-abstract class $c$ defined by the metamodel:

*Context:* $c \in Class$, $\neg c.abstract$

*Confidence:* $\|Object\|$

*Specification:* $CU(c) := \|\{o \in Object \mid isInstanceOf(o, c)\}\|$

**Q1. Which classes are not used?** We expect that the number of objects for a non-abstract class is greater than zero. Classes with no object represent language constructs that are not needed in practice, or the fact that language users did not know or understand how to use these constructs:

*Expectation:* $CU(c) > 0$

*Improvement:* Delete Class, Make Class Abstract

Classes with no objects that have subclasses can be made abstract, otherwise, classes without subclasses might be superfluous and thereby are candidates to be deleted from the metamodel. Both metamodel changes reduce the number of constructs available to the user, making the language easier to use and learn. Furthermore, deleting a class results in a smaller metamodel implementation which is easier to maintain by the language engineers. Non-abstract classes which the language engineers forgot to make abstract can be seen as metamodel bugs.

**Q2. What are the most widely used classes?** We expect that the more central a class of the metamodel is, the higher the number of its objects. If a class is more widely used than we expect, this might hint at a misuse of the class by the language users or the need for additional constructs:

*Expectation:* the more central the construct, the higher its use frequency

*Improvement:* the unexpectedly frequently used classes are source for language extensions

This analysis can only be performed manually, since the expectation cannot be automatically derived from the metamodel.

### Feature Usage Analysis

If metamodels are seen as basis for the definition of a language, features are typically used to define how the constructs of a modeling language can be combined (refer-

ences) and parameterized (attributes). As derived features cannot be set by language users, we investigate only the use of non-derived features:

*Context:* $f \in Feature, \neg f.derived$

*Confidence:* $\|FO(f)\|$, $FO(f) := \{o \in Object \mid isKindOf(o, f.class)\}$

*Specification:* $FU(f) := \|\{o \in FO(f) \mid o.isSet(f)\}\|$

We can only be confident for the cases when there exist objects $FO(f)$ of classes in which the feature $f$ could possibly be set, i.e. in all subclasses of the class in which the feature is defined.

**Q3. Which features are not used?** We expect that the number of times a non-derived feature is set is greater than zero. Otherwise, we can make it derived or even delete it from the metamodel:

*Expectation:* $FU(f) > 0$

*Improvement:* Delete Feature, make the feature derived

If we delete a feature from the metamodel or make it derived, it can no longer be set by the language users, thus simplifying the usage of the modeling language. Features that are not derived but need to be made derived can be seen as a bug in the metamodel, since the value set by the language user is ignored by the language interpreters.

### Feature Multiplicity Analysis

Multiplicities are typically used to define how many constructs can be referred from another construct. Again, we are only interested in non-derived features, and we can only be confident for a feature, in case there are objects in which the feature could possibly be set:

*Context:* $f \in Feature, \neg f.derived$

*Confidence:* $\|FU(f)\|$

*Specification:* $FM(f) : \mathbb{N} \to \mathbb{N}$, $FM(f, n) := \|\{o \in FU(f) \mid \|o.get(f)\| = n\}\|$

**Q4. Which features are not used to their full multiplicity?** We would expect that the distribution of used multiplicities covers the possible multiplicities of a feature. More specifically, we are interested in the following two cases: First, if the lower bound of the feature is 0, there should be objects with no value for the feature—otherwise, we might be able to increase the lower bound:

*Expectation:* $f.lowerBound = 0 \Rightarrow FM(f, 0) > 0$

*Improvement:* increase the lower bound (Specialize Attribute / Reference)

A lower bound greater than 0 explicitly states that the feature should be set, thus avoiding possible errors when using the metamodel. Second, if the upper bound of the feature is greater 1, there should be objects with more than one value for the feature—otherwise, we might be able to decrease the upper bound:

*Expectation:* $f.upperBound > 1 \Rightarrow \max\{n \in \mathbb{N} \mid FM(f, n) > 0\} > 1$

*Improvement:* decrease the upper bound (Specialize Attribute / Reference)

Decreasing the upper bound reduces the number of possible combinations of constructs and thereby simplifies the usage of the language.

### Attribute Value Analysis

The type of an attribute defines the values that an object can use. The measure in which the possible values are covered can be investigated by determining how often a certain value is used. Again, we are only interested in non-derived attributes, and we can only be confident for an attribute, if there are objects in which the attribute could possibly be set:

*Context:* $a \in Attribute$, $\neg a.derived$

*Confidence:* $FO(a)$

*Specification:* $AVU(a) : PV(a.type) \to \mathbb{N}$, $AVU(a, v) := \|\{o \in FO(a) \mid v \in o.get(a)\}\|$

**Q5. Which attributes are not used in terms of their values?** We expect that all the possible values of an attribute are used. In case of attributes that have a finite number of possible values (e.g. Boolean, Enumeration), we require them to be all used. In case of attributes with a (practically) infinite domain (e.g. Integer, String), we require that more than 10 different values are used. Otherwise, we might be able to specialize the type of the attribute.

*Expectation:* $(\|PV(a.type)\| < \infty \Rightarrow \|PV(a.type)\| = \|VU(a)\|) \wedge (\|PV(a.type)\| = \infty \Rightarrow \|VU(a)\| > 10)$, where $VU(a) = \{v \in PV(a.type) \mid AVU(a, v) > 0\}$

*Improvement:* specialize the attribute type (Specialize Attribute)

More restricted attributes can give users better guidance about how to fill its values in models, thus increasing usability. Additionally, such attributes are easier to implement for language engineers, since the implementation has to cover less cases.

**Q6. Which attributes do not have the most used value as default value?** In many cases, the language engineers set as default value of attributes the values that they think are most often used. In these cases, the value that is actually most widely used should be set as default value of the attribute:

*Expectation:* $a.upperBound = 1 \Rightarrow (a.defaultValue = mvu \in PV(a.type) \Leftrightarrow \max\{AVU(a, v) \mid v \in PV(a.type)\} = AVU(a, mvu))$

*Improvement:* change the default value

If this is not the case, and language users use other values, the new ones can be set as default.

**Q7. Which attributes have often used values?** We expect that no attribute value is used too often. Otherwise, we might be able to make the value a first-class construct

of the metamodel. A value is used too often if its usage share is at least 10%:

*Expectation:* $\|PV(a.type)\| = \infty \Rightarrow \forall v \in PV(a.type) : AVU(a, v) < 10\% \cdot FU(a)$

*Improvement:* lift the value to a first-class construct

Lifting the value to an explicit construct, makes the construct easier to use for language users as well as easier to implement for language engineers.

### 8.2.3 Prototypical Implementation

We have implemented the approach based on the Eclipse Modeling Framework (EMF) [Steinberg et al., 2009]. The gathering of usage data is implemented as a batch tool that traverses all the model elements. The results are stored in a model that conforms to a simple metamodel for usage data. The batch tool could be easily integrated into the modeling tool itself and automatically send the data to a server where it can be accessed by the language engineers. Since the usage data is required to be composeable, it can be easily aggregated.

The usage data can be loaded into the metamodel editor which proposes improvements based on the usage data. The expectations are implemented as constraints that can access the usage data. Figure 8.2 shows how violations of these expectations are presented to the user in the *metamodel editor*. Overlay icons indicate the metamodel elements to which the violations apply, and a view provides a list of all *usage problems*. The constraints have been extended to be able to propose operations for metamodel improvements. The proposed operations are shown in the context menu and can be executed via COPE's *operation browser* (see also Section 6.1.2 (*User Interface*)).

### 8.2.4 Study Goal

We have performed an empirical study to validate whether we can really identify metamodel improvements using the usage analysis method. More specifically, we were interested in the following two research questions:

- **RQ1.** *Do we identify deviations between actual and expected usage?* We are interested in whether models completely use their metamodels in practice.
- **RQ2.** *Do the deviations really lead to metamodel improvements?* We try to find explanations for the deviations in order to determine whether they really lead metamodel improvements.

### 8.2.5 Study Execution

To perform our analyses, we performed the following steps:

1. *Mine models:* We obtained as many models as possible that conform to a certain metamodel. In the case of an in-house metamodel, we asked the language engineers to provide us with the models known to them. For the published metamodels, we iterated through several open repositories (CVS and SVN) and

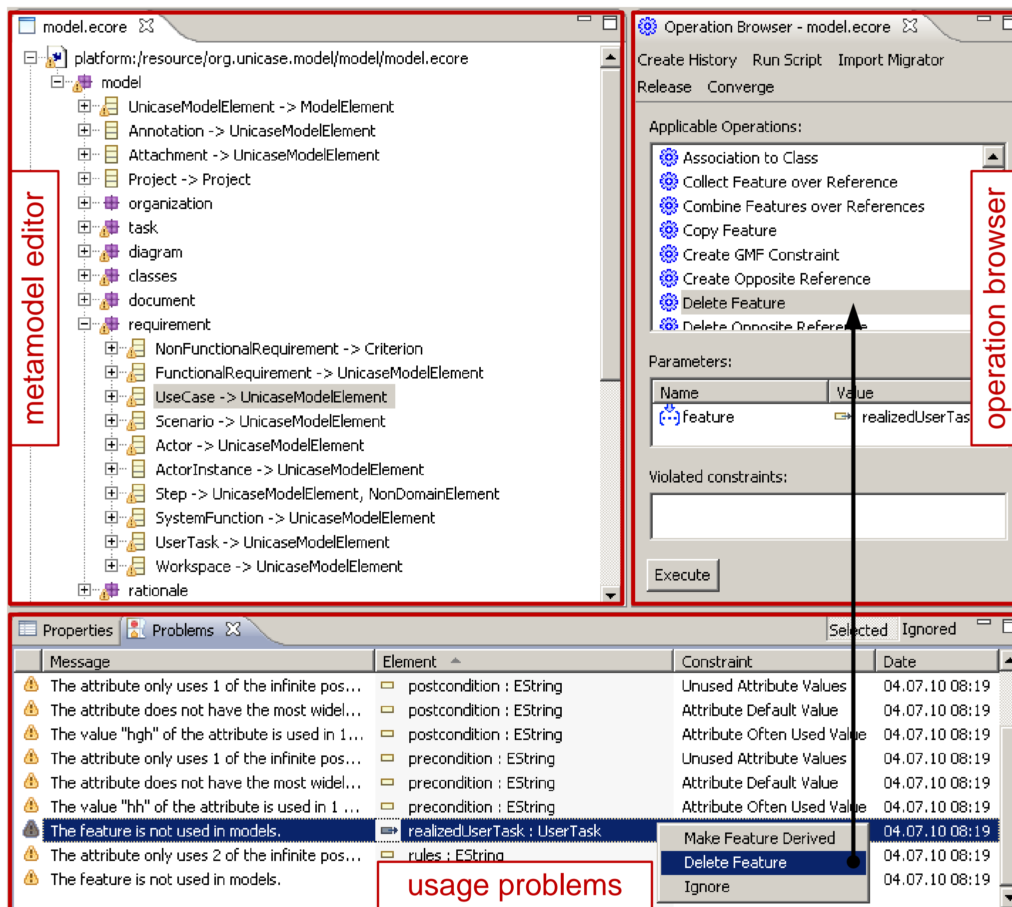



**Figure 8.2:** Proposing metamodel improvements in COPE

downloaded all models conforming to these metamodels. As far as possible, we removed the duplicate models.

2. *Perform usage analysis:* We applied the usage analyses presented in Section 8.2.2 (*Towards a Catalog of Usage Analyses*) to the mined models. For each of the analyzed metamodels, this results in a set of usage problems. The obtained usage problems can be used to answer *RQ1*.

3. *Interpret usage problems:* To determine whether the usage problems really help us to identify possible metamodel improvements, we tried to find explanations for the problems. In order to do so, we investigated the documentation of the metamodels as well as the interpreters of the modeling languages and, if possible, we interviewed the language engineers. The obtained explanations can be used to answer *RQ2*.

### 8.2.6 Study Object

**Metamodels.** To perform our experiments, we have chosen 7 metamodels whose usage we have analyzed. Table 8.1 shows the number of elements of these metamodels.

Two metamodels are part of the Eclipse Modeling Framework (EMF)[1] which is used to develop the abstract syntax of a modeling language: The ecore metamodel defines the abstract syntax from which an API for model access and a structural editor can be generated; and the genmodel allows language engineers to customize the code generation. Four metamodels are part of the Graphical Modeling Framework (GMF)[2] which can be used to develop the diagrammatic, concrete syntax of a modeling language: The graphdef model defines the graphical elements like nodes and edges in the diagram; the tooldef model defines the tools available to author a diagram; the mappings model maps the nodes and edges from the graphdef model and the tools from the tooldef model onto the metamodel elements from the ecore model; and the mappings model is transformed into a gmfgen model which can be altered to customize the generation of a diagram editor. Finally, the last metamodel (unicase) is part of the tool Unicase[3] which can be used for UML modeling, project planning and change management.

**Table 8.1:** A quantitative overview over the analyzed metamodels

| # | ecore | genmodel | graphdef | tooldef | mappings | gmfgen | unicase |
|---|---|---|---|---|---|---|---|
| Class | 20 | 14 | 72 | 26 | 36 | 137 | 77 |
| Attribute | 33 | 110 | 78 | 16 | 22 | 302 | 88 |
| Reference | 48 | 34 | 57 | 12 | 68 | 160 | 161 |

**Models.** For each metamodel, we have mined models from different repositories. Table 8.2 shows the repositories as well as the number of models and model elements which have been obtained from them. Models that conform to the first 6 metamodels have been obtained from the AUTOSAR development partnership[4], from the Eclipse[5] and GForge[6] open source repositories, by querying the Google Code Search[7] and from the Atlantic Zoo[8]. For the unicase metamodel, its language engineers provided us with 3 models consisting of 8,213 model elements.

**Table 8.2:** A quantitative overview over the analyzed models

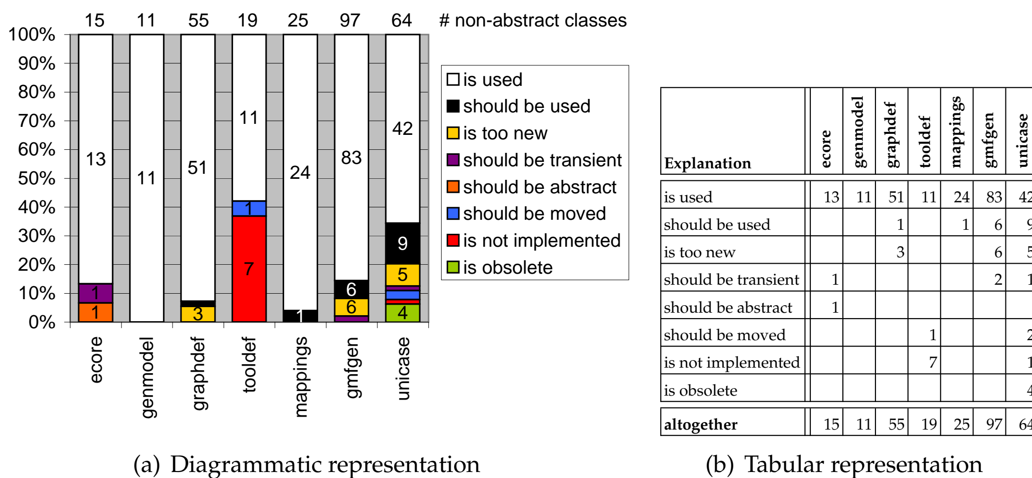
| | ecore | | genmodel | | graphdef | | tooldef | | mappings | | gmfgen | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| repository | # models | # model elements | # models | # model elements | # models | # model elements | # models | # model elements | # models | # model elements | # models | # model elements |
| AUTOSAR | 18 | 384,685 | 18 | 16,189 | 11 | 1,835 | 11 | 436 | 11 | 538 | 13 | 2,373 |
| Eclipse | 1,834 | 250,107 | 818 | 69,361 | 105 | 6,026 | 58 | 1,769 | 72 | 5,040 | 52 | 11,043 |
| GForge | 106 | 26,736 | 94 | 41,997 | 12 | 806 | 10 | 241 | 11 | 480 | 11 | 1,680 |
| Google | 50 | 9,266 | 59 | 3,786 | 69 | 7,627 | 74 | 2,421 | 76 | 4,028 | 78 | 18,710 |
| Atlantic Zoo | 278 | 68,116 | – | – | – | – | – | – | – | – | – | – |
| **altogether** | 2,286 | 738,910 | 989 | 131,333 | 197 | 16,294 | 153 | 4,867 | 170 | 10,086 | 154 | 33,806 |

[1] see EMF web site: `http://www.eclipse.org/emf`
[2] see GMF web site: `http://www.eclipse.org/gmf`
[3] see Unicases web site: `http://unicase.org`
[4] see AUTOSAR web site: `http://www.autosar.org`
[5] see Eclipse repository web site: `http://dev.eclipse.org/viewcvs/index.cgi/`
[6] see GForge web site: `http://gforge.enseeiht.fr`
[7] see Google Code Search web site: `http://www.google.com/codesearch`
[8] see Atlantic Zoo web site: `http://www.emn.fr/z-info/atlanmod/index.php/Ecore`

### 8.2.7 Study Result

In this section, we present the study results separately for each question mentioned in Section 8.2.2 (*Towards a Catalog of Usage Analyses*). To ease understanding the explanations for usage problems, we clustered them according to high-level explanations.

**Q1. Which classes are not used?** Figure 8.3 quantitatively illustrates for each metamodel the share of used and not used classes in the overall number of non-abstract classes—both as diagram and table. In the diagram, we represent with white the ratio of classes that are used, whereas with other colors we classify the unused classes according to the explanations why they are not used. As presented in Section 8.2.5 (*Study Execution*), we derived these explanations by manually analyzing the documentation and implementation of the metamodels or interviewing the language engineers.




(a) Diagrammatic representation

| Explanation | ecore | genmodel | graphdef | tooldef | mappings | gmfgen | unicase |
|---|---|---|---|---|---|---|---|
| is used | 13 | 11 | 51 | 11 | 24 | 83 | 42 |
| should be used | | | 1 | | 1 | 6 | 9 |
| is too new | | | 3 | | | 6 | 5 |
| should be transient | 1 | | | | | 2 | 1 |
| should be abstract | 1 | | | | | | |
| should be moved | | | | 1 | | | 2 |
| is not implemented | | | | 7 | | | 1 |
| is obsolete | | | | | | | 4 |
| altogether | 15 | 11 | 55 | 19 | 25 | 97 | 64 |

(b) Tabular representation

**Figure 8.3:** Usage of classes (Q1)

Classes that are obsolete, not implemented, or that logically belong to another metamodel can be removed. A class *is obsolete* if it is not intended to be used in the future. For example, in unicase, the 4 classes to define UML stereotypes are no longer required. A class *is not implemented* if it is not used by the interpreters of the modeling language. For example, the tooldef metamodel defines 7 classes to specify menus and toolbars from which according to [Gronback, 2009] currently no code can be generated. A class *should be moved* to another metamodel if it logically belongs to the other metamodel. For example, in tooldef, the class GenericStyleSelector should be moved to mappings which contains also a composite reference targeting this class. Another example is unicase where 2 classes should be moved to a different metamodel of another organization that also uses the framework underlying Unicase.

Our manual investigations revealed that other classes that are not used should be abstract or transient. A class *should be abstract* if it is not intended to be instantiated, and is used only to define common features inherited by its subclasses. For instance, in ecore, EObject—the implicit common superclass of all classes—should be made abstract. A class *should be transient* if its objects are not expected to be made persistent—such a class does not represent a language construct. However, the em-

ployed metamodeling formalism currently does not support to specify that a class is transient. For instance, in ecore, EFactory—a helper class to create objects of the specified metamodel—should be transient.

Finally, there are non-used classes which do not require any change, since they are either *too new* or *should be used*—i.e. we could not find any plausible explanation why they are not used. A class is *too new* if it was recently added and thus is not yet instantiated in models. For the GMF metamodels graphdef and gmfgen, we identified 9 new classes, while for the unicase metamodel, we found 5 classes that are too new to be used.

**Q2. What are the most widely used classes in ecore?** Figure 8.4 shows the classes defined by the ecore metamodel and their corresponding number of objects. Interestingly, the number of objects of the ecore classes has an exponential distribution, a phenomenon observed also in case of the other metamodels. Hence, each metamodel contains a few constructs which are very central to its users. In the case of ecore, we expect that classes, references and attributes are the central constructs. However, the most widely used ecore class is—with more than 44.5% of the analyzed objects—EStringToStringMapEntry which defines key-value-pairs for EAnnotations making up 13.9% of the objects. The fact that the annotation mechanism—which is used to extend the ecore metamodel—is more widely used than the first-class constructs, suggests the need for increasing ecore's expressiveness. As we show in Q7, some often encountered annotations could be lifted to first-class constructs. Another ecore class that is used very often is EGenericType. This is surprising, since we would expect that generic types are very rarely used in metamodels. The investigation of how this class is exactly used, revealed the fact that only 1.8% (2,476) of EGenericType's objects do not define default values for their features and thereby these objects really represent generic types. In the other 98%, the *EGenericType* is only used an indirection to the non-generic type, i.e. in a degenerated manner.

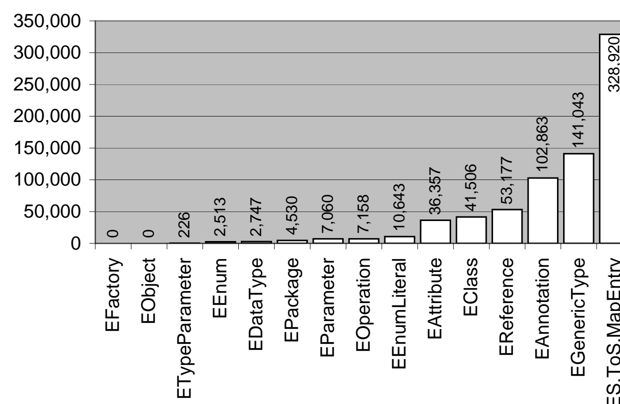
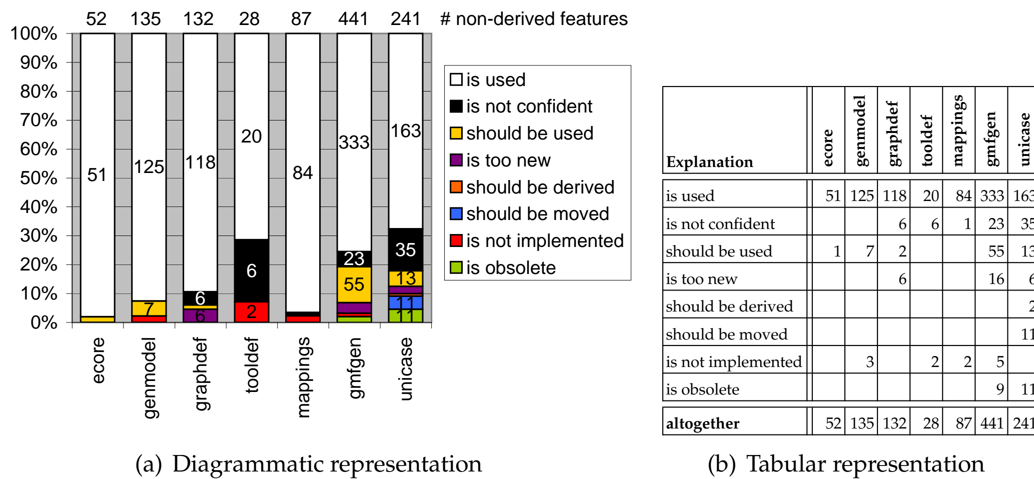



**Figure 8.4:** Usage of ecore classes (Q2)

**Q3. Which features are not used?** Figure 8.5 shows for each metamodel the share of used and unused features in the overall number of non-derived features. In the figure, we remark a correlation between the number of unused features and the overall number of features. In most cases, the more features a metamodel defines, the

less features are actually used. The only exception to this conjecture is the usage of the tooldef metamodel, most of whose features are however not implemented as explained below. The figure also classifies the unused features according to explanations why they are not used.




(a) Diagrammatic representation

| Explanation | ecore | genmodel | graphdef | tooldef | mappings | gmfgen | unicase |
|---|---|---|---|---|---|---|---|
| is used | 51 | 125 | 118 | 20 | 84 | 333 | 163 |
| is not confident | | | 6 | 6 | 1 | 23 | 35 |
| should be used | 1 | 7 | 2 | | | 55 | 13 |
| is too new | | | 6 | | | 16 | 6 |
| should be derived | | | | | | | 2 |
| should be moved | | | | | | | 11 |
| is not implemented | | 3 | | 2 | 2 | 5 | |
| is obsolete | | | | | | 9 | 11 |
| **altogether** | 52 | 135 | 132 | 28 | 87 | 441 | 241 |

(b) Tabular representation

**Figure 8.5:** Usage of features (Q3)

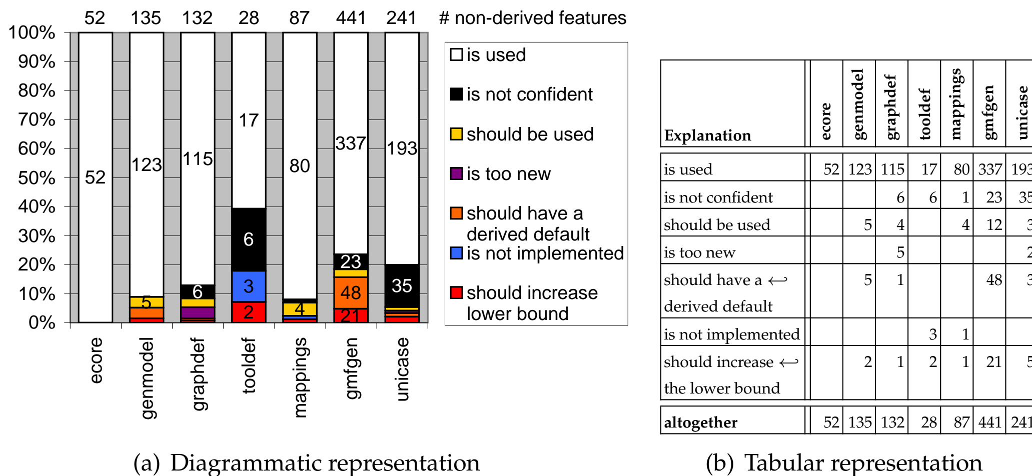
Features that are obsolete, not implemented, or logically belong to another metamodel can be removed from the metamodel. A feature *is obsolete* if it is not intended to be used in the future. If none of the analyzed models use that feature, it is a good candidate to be removed. We identified the 9 obsolete features of gmfgen by investigating the available migrator. Surprisingly, the migrator removes their values, but the language engineers forgot to remove the features from gmfgen. In the case of the unicase metamodel, 11 unused features are actually obsolete. A feature is classified as *not implemented* if it is not used by the interpreters of the modeling language. We have identified 12 not implemented features of the EMF metamodel genmodel and the GMF metamodels tooldef, mappings and gmfgen by checking whether they are accessed by the code generators. A feature *should be moved* to another metamodel if it logically belongs to the other metamodel. The features of the unicase metamodel that have to be moved target the classes that should be moved to another metamodel as found by question Q1.

Another set of non-used features can be changed into derived features, since their values are calculated based on other features. As they are anyway overwritten in the metamodel implementation, setting them explicitly is a mistake, making the language easy to misuse. For example, for the unicase metamodel, we identified 2 features that *should be derived*.

Finally, there are non-used features which do not require changes at all, since they are either too new or our investigation could not identify why the feature is not used. Again, a feature *is too new* to be instantiated, if it was recently added. Like for classes, the 28 too new features were identified by investigating the metamodel histories. The only feature of the ecore metamodel which is not set in any model but *should be used* allows language engineers to define generic exceptions for operations. Apparently, exceptions rarely need to have type parameters defined or assigned. For both genmodel and gmfgen, more than 5% of the features *should be used* but are not

used. The manual investigation revealed that these are customizations of the code generation that are not used at all. We are *not confident* about the usage of a feature if there are no objects in which the feature could be set. This category thus indicates how many features are not used, because the classes in which they are defined are not instantiated.

**Q4. Which features are not used to their full multiplicity?** Figure 8.6 shows in white the share of used features which fulfill these expectations. Again, the violations are classified according to different explanations that we derived after manual investigation of the metamodel documentation and implementation.




(a) Diagrammatic representation

| Explanation | ecore | genmodel | graphdef | tooldef | mappings | gmfgen | unicase |
|---|---|---|---|---|---|---|---|
| is used | 52 | 123 | 115 | 17 | 80 | 337 | 193 |
| is not confident | | | 6 | 6 | 1 | 23 | 35 |
| should be used | | 5 | 4 | | 4 | 12 | 3 |
| is too new | | | 5 | | | | 2 |
| should have a ↩ derived default | | 5 | 1 | | | 48 | 3 |
| is not implemented | | | | 3 | 1 | | |
| should increase ↩ the lower bound | | 2 | 1 | 2 | 1 | 21 | 5 |
| **altogether** | 52 | 135 | 132 | 28 | 87 | 441 | 241 |

(b) Tabular representation

**Figure 8.6:** Usage of feature multiplicity (Q4)

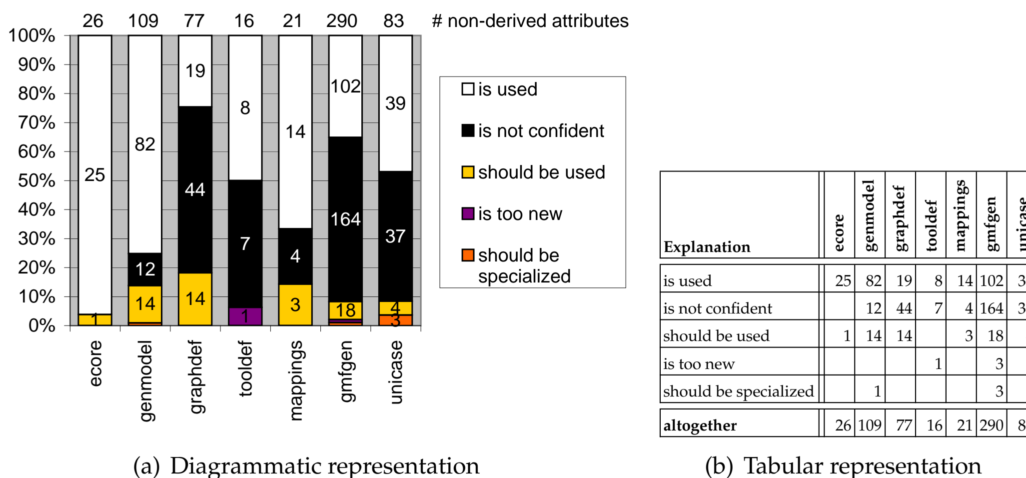
A not completely used feature which can be restricted either *should increase lower bound* or *is not implemented*. Even though we found features whose usage did not completely use the upper bound, we could not find an explanation for any of them. However, we found that some features *should increase lower bound* from 0 to 1. For the EMF and GMF metamodels, we found such features by analyzing whether the code generator does not check whether the feature is set, thereby producing an error if the feature is not set. For the unicase metamodel, the too low lower bounds date back from the days when its language engineers used an object-to-relational mapping to store data. When doing so, a lower bound of 1 was transformed into a database constraint which required to set the feature already when creating a model element. To avoid this restriction, the lower bounds were decreased to 0, when in effect, they should have been 1. As they no longer store the models in a database, the lower bounds could easily be increased. Again, a feature *is not implemented* if the interpreter does not use the feature. In the tooldef metamodel, we found 3 such features which are not interpreted by the code generator, making them also superfluous.

A not completely used feature which requires to extend the metamodeling formalism *should have a derived default*. A feature *should have a derived default* if it has lower bound 0, but in case it is not set, a default value is derived. This technique is mostly used by the code generation metamodels genmodel and gmfgen to be able to customize a value which is otherwise derived from other features. It is implemented by overwriting the API to access the models which is generated from the metamodel. However,

the metamodeling formalism used does not provide a means to specify that a feature has a derived default.

A not completely used feature which does not require changes either *is too new* or *should be used*.

**Q5. Which attributes are not used in terms of their values?** Figure 8.7 illustrates for each metamodel the share of attributes whose values are completely used or not. The figure also classifies the not completely used attributes according to different explanations.




(a) Diagrammatic representation

| Explanation | ecore | genmodel | graphdef | tooldef | mappings | gmfgen | unicase |
|---|---|---|---|---|---|---|---|
| is used | 25 | 82 | 19 | 8 | 14 | 102 | 39 |
| is not confident | | 12 | 44 | 7 | 4 | 164 | 37 |
| should be used | 1 | 14 | 14 | | 3 | 18 | 4 |
| is too new | | | | 1 | | 3 | |
| should be specialized | | 1 | | | | 3 | 3 |
| **altogether** | 26 | 109 | 77 | 16 | 21 | 290 | 83 |

(b) Tabular representation

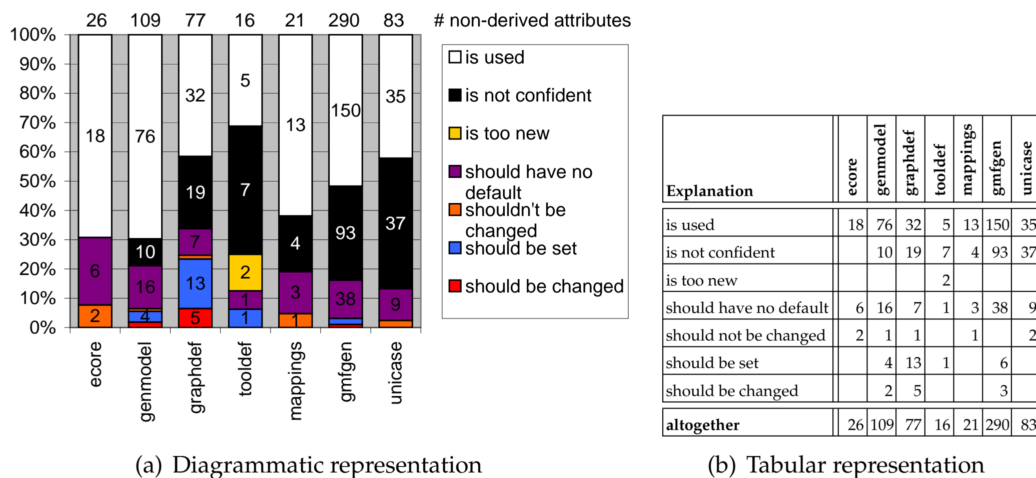
**Figure 8.7:** Usage of complete values by attributes (Q5)

A not completely used attribute that can be changed *should be specialized* by restricting its type. In the unicase, three attributes which use String as domain for UML association multiplicities and UML attribute types can be specialized. We found 4 more such attributes in the genmodel and gmfgen metamodels.

Finally, there are not completely used attributes which do not require changes at all, since they are either *too new*, *should be used* or we do not have enough models to be *confident* about the result. We are only confident if the attribute is set sufficiently often to cover all its values (finite) or 10 values (infinite). In all metamodels, most of the findings fall into one of these categories.

**Q6. Which attributes do not have the most used value as default value?** Figure 8.8 illustrates for each metamodel the share of attributes whose value is the same as the default value (white) and those that have different values. Note that in many cases the language users successfully anticipated the most often values of attributes by setting the appropriate default value. The figure also classifies the deviations according to different explanations.

In case the default value is intended to represent the most widely used value of an attribute and we found that the language users use other default values, the default value *should be changed*. In this way, the new value anticipates better the actual use and thereby the effort of language users to change the attribute value is necessary in less cases. We found 8 attributes whose default value needs to be updated in




(a) Diagrammatic representation

| Explanation | ecore | genmodel | graphdef | tooldef | mappings | gmfgen | unicase |
|---|---|---|---|---|---|---|---|
| is used | 18 | 76 | 32 | 5 | 13 | 150 | 35 |
| is not confident | | 10 | 19 | 7 | 4 | 93 | 37 |
| is too new | | | | 2 | | | |
| should have no default | 6 | 16 | 7 | 1 | 3 | 38 | 9 |
| should not be changed | 2 | 1 | 1 | | 1 | | 2 |
| should be set | | 4 | 13 | 1 | | 6 | |
| should be changed | | 2 | 5 | | | 3 | |
| **altogether** | 26 | 109 | 77 | 16 | 21 | 290 | 83 |

(b) Tabular representation

**Figure 8.8:** Usage of default values of attributes (Q6)

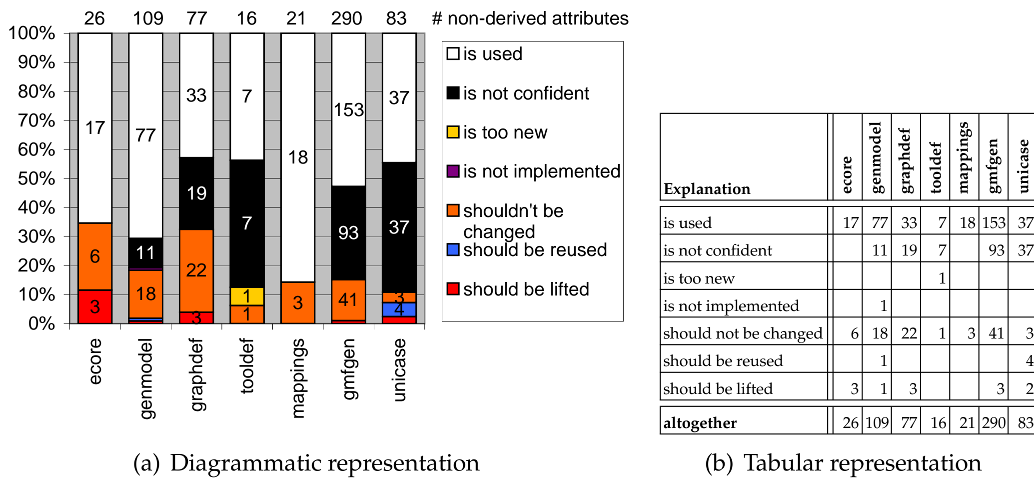
the metamodels genmodel, graphdef and gmfgen. An attribute has a default value that *should be set* if it does not currently have a default value, but the usage analysis identifies a recurrent use of certain values. By setting a meaningful default value, the language users are helped. In nearly each metamodel, we found attributes whose default value needs to be set.

The unexpected default value of an attribute which does not require change either *should have no default*, *shouldn't be changed*, *is too new* or *is not confident*. An attribute *should have no default* value if we cannot define a constant default value for the attribute. In each metamodel, we are able to find attributes of this kind which are usually of type String or Integer. A default value *shouldn't be changed* if we could not find a plausible explanation for setting or changing the default value. Two attributes from the unicase metamodel whose default value shouldn't be changed denote whether a job is done. Most of the attribute values are true—denoting a completed job—but in the beginning the job shouldn't be done. We are *not confident* about an attribute if it is not set at least 10 times.

**Q7. Which attributes have often used values?** Figure 8.9 shows the share of attributes which have often used values or not. The figure also classifies the attribute with recurring values according to explanations.

A recurring value which requires metamodel changes either *should be lifted* or *should be reused*. A value *should be lifted* if the value should be represented by a new class in the metamodel. In ecore, we find often used values for the source of an annotation as well as for the key of an annotation entry. Figure 8.10 illustrates the 6 most widely used values of the annotation source. The GenModel[9] annotations customize the code generation, even though there is a separate generator model. Thereby, some of these annotations can be lifted to first-class constructs of the genmodel metamodel. The ExtendedMetaData[10] extends ecore to accommodate additional constructs of XML Schemas which can be imported. This shows the lack of expressiveness of ecore

[9] http://www.eclipse.org/emf/2002/GenModel
[10] http://org/eclipse/emf/ecore/util/ExtendedMetaData




| Explanation | ecore | genmodel | graphdef | tooldef | mappings | gmfgen | unicase |
|---|---|---|---|---|---|---|---|
| is used | 17 | 77 | 33 | 7 | 18 | 153 | 37 |
| is not confident | | 11 | 19 | 7 | | 93 | 37 |
| is too new | | | | 1 | | | |
| is not implemented | | 1 | | | | | |
| should not be changed | 6 | 18 | 22 | 1 | 3 | 41 | 3 |
| should be reused | | 1 | | | | | 4 |
| should be lifted | 3 | 1 | 3 | | | 3 | 2 |
| **altogether** | 26 | 109 | 77 | 16 | 21 | 290 | 83 |

(a) Diagrammatic representation (b) Tabular representation

**Figure 8.9:** Usage of values often used by attributes (Q7)

in comparison to XML Schema. The next three sources represent stereotypes and result from the import of metamodels from UML class diagrams. The high number of these cases are evidence that many ecore models originate from UML class diagrams. The last source extends ecore which is an implementation of E-MOF with a subset relation between features which is only available in C-MOF. This shows that for many use cases the expressiveness of E-MOF is not enough. Furthermore, the key of an annotation entry is used in 10% of the cases to specify the documentation of a metamodel element. However, it would be better to make the documentation an explicit attribute of EModelElement—the base class for each metamodel element in ecore. A value *should be reused* if—instead of recurrently using the value—we refer to a single definition of the value as object of a new class. In unicase, instead of defining types of UML attributes as Strings, it would be better to group them into separate objects of reusable data types.

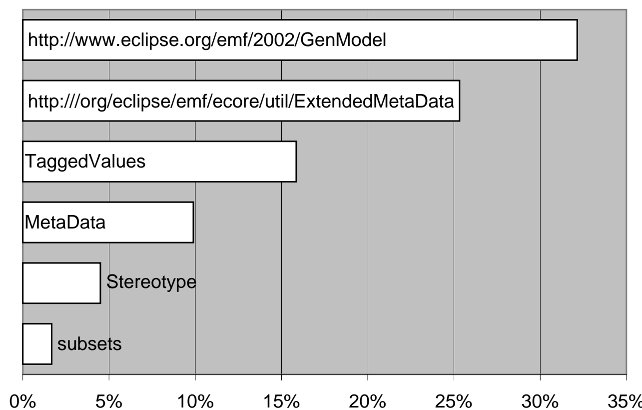



**Figure 8.10:** Most widely used annotations in ecore (Q7)

An attribute with recurring values which does not require change either *shouldn't be changed*, *is not implemented*, *is too new* or we are *not confident*. For a lot of attributes, we have not found a plausible explanation, and thus conservatively assumed that they *shouldn't be changed*.

### 8.2.8 Study Discussion

Based on the results of our analyses, we learned a number of lessons about the usage of metamodels by existing models.

**Metamodels can be more Restrictive.** In all investigated cases, the set of models covers only a subset of the set of all possible models that can be built with a metamodel. We discovered that not all classes are instantiated (Q1), not all features are used (Q3), and that the range of the cardinality of many features does not reflect their definition from the metamodel (Q4). In all these cases, the metamodels can be improved by restricting the number of admissible models.

**Metamodels can Contain Latent Defects.** During our experiments, we discovered in several cases defects of the metamodels—e.g. classes that should not be instantiated and are not declared abstract (Q1), classes and features that are not implemented or that are overwritten by the generator (Q1, Q3, Q4, Q6). Furthermore, in other cases (Q2), we discovered misuses of metamodel constructs.

**Metamodels can be Extended.** Each of the analyzed metamodels offers their users the possibility to add new information (e.g. as annotations, or sets of key-value pairs). The analysis of the manners in which the metamodels are actually extended reveal that the language users recurrently used several annotation patterns. These patterns reveal the need to extend the metamodel with new explicit constructs that capture their intended use (Q2, Q7). Furthermore, the specification of some attributes can be extended with the definition of default values (Q6).

**Metamodeling Formalism can be Improved.** Our metamodeling formalism is very similar to `ecore`. The results indicate that in certain cases the metamodeling formalism is not expressive enough to capture certain constraints. For instance, we would have required to mark a class as transient (Q1) and to state that a feature has a default value derived from other features (Q4). Consequently, we have also identified improvements concerning the metamodeling formalism.

### 8.2.9 Threats to Validity

We are aware of the following threats to the validity of our results.

**Validity of Explanations.** We presented a set of explanations for the deviations between actual and expected usage. In the case of Unicase, we had direct access to the language engineers and hence could ask them directly about their explanations for the analysis results. In the case of the other metamodels, we interpreted the analysis results based only on the documentation and implementation. Consequently, some of our explanations can be mistaken.

**Relevance and Number of Analyzed Models.** We analyzed only a subset of the entire number of existing models. This fact can make our analyses results rather

questionable. In the case of Unicase, we asked the language engineers to provide us with a representative sample of existing models. In the case of the other metamodels, we mined as many models as possible from both public and private (AUTOSAR) repositories to obtain a representative sample of existing models.

### 8.2.10 State of the Art

**Investigating Language Utterances.** In the case of general purpose languages, investigating their use is especially demanding, as a huge number of programs exist. There are some landmark works that investigate the use of general purpose languages [Gil and Maman, 2005, Singer et al., 2009, Lämmel and Pek, 2010]. Gil et al. present a catalog of Java micro patterns which capture common programming practice on the class-level [Gil and Maman, 2005]. By automatically searching these patterns in a number of projects, they show that 75% of the classes are modeled after these patterns. Singer et al. present a catalog of Java nano patterns which capture common programming practice on the method-level [Singer et al., 2009]. There is also work on investigating the usage of domain-specific languages. Lämmel et al. analyze the usage of XML schema by applying a number of metrics to a large corpus [Lämmel et al., 2005]. Lämmel and Pek present their experience with analyzing the usage of the W3C's P3P language [Lämmel and Pek, 2010]. Our empirical results—that many language constructs are more often used than others—are consistent with all these results. We use a similar method for investigating the language utterances, but our work is focused more on identifying improvements for languages. Tolvanen proposes a similar approach for the metamodeling formalism provided by MetaCase [Tolvanen, 1998]. However, even if the sets of analyses are overlapping, our set contains analyses not addressed by Tolvanen's approach (Q2, Q5 and Q7), and provides a validation through a large-scale empirical study that usage analyses do help to identify metamodel improvements.

**Language Improvements.** Once the information about the language use is available, it can serve for language improvements. Atkinson and Kuehne present techniques for language improvements like restricting the language to the used subset, or adding new constructs that reflect directly the needs of the language users [Atkinson and Kühne, 2007]. Sen et al. present an algorithm that prunes a metamodel [Sen et al., 2009]: it takes a metamodel as input as well as a subset of its classes and features and outputs a restricted metamodel that contains only the desired classes and features. By doing this, we obtain a restricted metamodel that contains only necessary constructs. Our work on mining the metamodel usage can serve as input for the pruning algorithm that would generate a language more appropriate to the expectations of its users. Once recurrent patterns are identified, they can serve as source for new language constructs [Bosch, 1998]. The results of our study demonstrate that the analysis of built programs is a feasible way to identify language improvements. Lange et al. show—by performing an experiment—that modeling conventions have a positive impact on the quality of UML models [Lange et al., 2006]. Henderson-Sellers and Gonzalez-Perez report on different uses and misuses of the stereotype mechanism provided by UML [Henderson-Sellers and Gonzalez-Perez, 2006]. By analyzing the models built, we might be able to identify certain types of conventions as

well as misuses of language extension mechanisms.

**Tool Usage.** There is work on analyzing the language use by recording and analyzing the interactions with the language interpreter. Li et al. present a study on the usage of the Alloy analyzer tool [Li et al., 2006]. From the way how the analyzer tool was used, they identified a number of techniques to improve analysis performance. Hage and Keeken present the framework Neon to analyze usage of a programming environment [Hage and van Keeken, 2009]. Neon records data about every compile performed by a programmer and provides a query to analyze the data. To evaluate Neon, the authors executed analyses showing how student programmers improve over time in using the language. The purpose of our approach is not to improve the tools for using the language, but to improve the language itself.

## 8.3 Towards Semantics-Preserving Model Migration

Building an automated model migration is a difficult endeavor, as it needs to preserve the meaning of a possibly unknown number of models. In this section, we analyze whether our operation-based approach to automate model migration can be extended by a means to constructively ensure semantics preservation. The idea is that a coupled operation not only adapts the metamodel and provides the appropriate model migration, but also adapts the semantics definition. Semantics preservation is thus ensured constructively by defining appropriate couples of semantics adaptation and model migration. We showcase the approach using the well-known Petri net example evolution [Wachsmuth, 2007].

### 8.3.1 Adaptation of the Semantics Definition

For our approach to work, we need an explicit definition of the semantics of a modeling language. A specific modeling language $\mathcal{L}_\mathcal{S}$ is required to define the semantics of a modeling language:

**Definition 8.1** (Modeling Language for Defining Semantics)**.** *Let* $\mathcal{S} := \{\mathcal{M} \rightarrow \mathcal{SD}\}$ *be the set of all possible semantics. A modeling language for defining semantics is a modeling language* $\mathcal{L}_\mathcal{S}$ *whose semantics* $S_\mathcal{S} : \mathcal{L}_\mathcal{S} \rightarrow \mathcal{S}$ *maps a semantics definition* $sd \in \mathcal{L}_\mathcal{S}$ *to a semantics* $S_\mathcal{S}(sd) \in \mathcal{S}$*.*

There are different ways to define the semantics of a modeling language, e.g. operational, denotational, axiomatic. In this section, we define the semantics as operations on metamodel classes, similar to Kermeta [Muller et al., 2005]. This operational semantics can be used to execute a model in order to learn about its meaning. However, we believe that the approach is also applicable to other languages for specifying semantics.

**Example 8.1** (Petri Net Modeling Language)**.** *We use the well-known Petri net evolution [Wachsmuth, 2007] as a running example. Figure 8.11 shows the metamodel, and Figure 8.12 the semantics definition of the first version of the Petri net modeling language. To define the semantics, we use a notation that is similar to the Groovy scripting language*

*[Koenig et al., 2007]. The first line of each block defines the signature of an operation which is associated to a class defined by the metamodel. Within the block, all accesses are performed on an object of the respective class.*

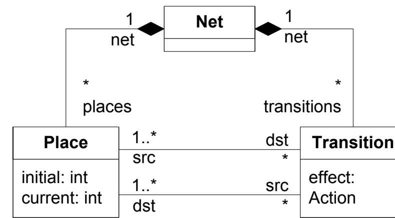



**Figure 8.11:** Metamodel of the Petri net modeling language version 1

```
Net.run(Interaction i):
  init()
  ts = getActivatedTransitions()
  while(!ts.isEmpty()) {
    if(ts.size() == 1) ts.get(0).fire()
    else i.choose(ts).fire()
    ts = getActivatedTransitions()
  }

Interaction.choose(List<Transition> ts):
  ...

Net.init():
  places.each{p -> p.init()}

Net.getActivatedTransitions():
    List<Transition>
  return transitions.collect{t ->
    t.isActivated()}

Transition.isActivated(): boolean
  return src.every{p -> p.isActivated()}

Transition.fire():
  src.each{p -> p.decrement()}
  effect.execute()
  dst.each{p -> p.increment()}

Place.init():
  current = initial

Place.isActivated(): boolean
  return current >= 1

Place.decrement():
  current = current - 1

Place.increment():
  current = current + 1
```

**Figure 8.12:** Semantics definition of the Petri net modeling language version 1

*Petri* Net*s consist of* places *and* transitions*. A* Place *has a number of tokens (*current*) which are* init*ialized with the number of* initial *tokens when the net is started. A* Transition *transfers tokens from source (*src*) to destination (*dst*) places. A transition* isActivated *if every* src *place has at least one token. If multiple transitions are activated at the same time, the user is required to* choose *from these transitions the transition that should be fired—which is implemented by means of the interface* Interaction*. When a transition* fire*s, the tokens of the* src *places are* decrement*ed, its* effect *is* execute*d and the tokens of the* dst *places are* increment*ed. When there are no more activated transitions, the execution is finished. The semantics definition thus maps Petri net models to traces of action executions.*

Not only the models depend on the metamodel, but also the semantics definition. Thereby, when the metamodel is adapted, the semantics definition often needs to be adapted, too.

**Definition 8.2** (Semantics Adaptation). *A semantics adaptation is a function $ads : \mathcal{L}_\mathcal{S} \to \mathcal{L}_\mathcal{S}$ that transforms a semantics definition $sd \in \mathcal{L}_\mathcal{S}$ to $ads(sd) \in \mathcal{L}_\mathcal{S}$.*

However, the dependency between model and metamodel is different from the dependency between semantics definition and metamodel: A model conforms to the metamodel, whereas a semantics definition is defined based on the metamodel. Consequently, the migration of models and the adaptation of semantics definitions are

different from each other. However, they need to be consistent with each other in order to ensure semantics preservation.

**Example 8.2** (Evolved Modeling Language)**.** *Currently, when a transition is fired, the tokens of the source places are decreased by 1, and the tokens of the destination places are increased by 1. However, we want to be able to specify for each source and destination place of a transition numbers that are different from 1. In order to do so, we need to extend the Petri net modeling language with weighted arcs [Wachsmuth, 2007]. Figure 8.13 shows the adapted metamodel, and Figure 8.14 the migrated semantics definition.*

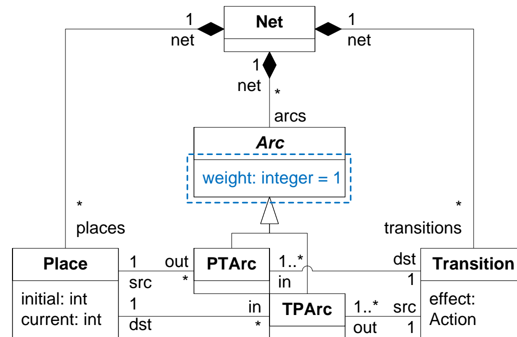



**Figure 8.13:** Metamodel of the Petri net modeling language version 2

```
Net.run(Interaction i):
  init()
  ts = getActivatedTransitions()
  while (!ts.isEmpty()) {
    if(ts.size() == 1) ts.get(0).fire()
    else i.choose(ts).fire()
    ts = getActivatedTransitions()
  }

Interaction.choose(List<Transition> ts):
  ...

Net.init():
  places.each{p -> p.init()}

Net.getActivatedTransitions():
    List<Transition>
  return transitions.collect{t ->
    t.isActivated()}
```

```
Transition.isActivated(): boolean
  return in.every{pt-> pt.isActivated()}

Transition.fire():
  in.each{pt -> pt.decrement()}
  effect.execute()
  out.each{tp -> tp.increment()}

Place.init():
  current = initial

PTArc.isActivated(): boolean
  return src.current >= weight

PTArc.decrement():
  src.current = src.current - weight

TPArc.increment():
  dst.current = dst.current + weight
```

**Figure 8.14:** Semantics definition of the Petri net modeling language version 2

*Arcs are introduced to define the incoming and outgoing weights for transitions. PTArcs define the number of tokens by which the src places of transitions are decremented. TPArcs define the number of tokens by which the dst places of transitions are incremented.*

### 8.3.2 Ensuring Semantics Preservation

Until now, coupled operations only encapsulate metamodel adaptation and model migration and thus can only ensure syntax preservation. To also ensure semantics preservation constructively, a coupled operation needs to be extended to also encapsulate the semantics adaptation.

**Definition 8.3** (Semantics-Preserving Coupled Operation)**.** *A semantics-preserving coupled operation is a syntax-preserving coupled operation that also provides a semantics adaptation* $ads : \mathcal{L}_{\mathcal{S}} \to \mathcal{L}_{\mathcal{S}}$*. It is semantics-preserving for a metamodel* $mm \in \mathcal{MM}$ *and semantics definition* $sd \in \mathcal{L}_{\mathcal{S}}$ *if it preserves the meaning of all models conforming to that metamodel:*

$$\forall m \in \mathcal{L}_{mm} : S_{\mathcal{S}}(sd)(m) \cong S_{\mathcal{S}}(ads(sd))(mig(m))$$

*where* $\cong \subseteq \mathcal{SD}_1 \times \mathcal{SD}_2$ *is the relation between the two semantic domains for the semantics change* $S_{\mathcal{S}}(sd) \mapsto S_{\mathcal{S}}(ads(sd))$ *from* $S_{\mathcal{S}}(sd) = S_1 : \mathcal{L}_{mm} \to \mathcal{SD}_1$ *to* $S_{\mathcal{S}}(ads(sd)) = S_2 : \mathcal{L}_{adm(mm)} \to \mathcal{SD}_2$*.*

Consequently, the model migration and semantics adaptation need to be defined consistently in order to ensure semantics preservation. For reusable coupled operations, the semantics adaptation can be defined in a metamodel-independent manner and thus be reused across metamodels. For custom coupled operations which require manual specification of the model migration, the semantics definition also has to be adapted manually. Since custom coupled operations are rather rare in practice according to our case studies (see Chapter 7 (*Case Studies*)), it is sufficient to manually prove semantics preservation in these cases.

**Example 8.3** (Semantics-Preserving Coupled Operation)**.** *The coupled operation Reference to Class that is required for the Petri net evolution replaces a reference r by an explicit reference class R. The adaptation of the metamodel which is shown in Figure 8.15 makes the reference r composite and creates the reference class R as its new type. Single-valued references s and t are created in the reference class R to target the source and target class C1 and C2 of the original reference r. The model migration replaces links conforming to the reference by objects of the reference class, setting source and target reference appropriately. Now, the coupled operation is already syntax-preserving.*

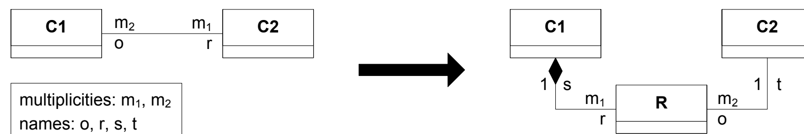



**Figure 8.15:** Metamodel adaptation of the coupled operation Reference to Class

*To make it semantics-preserving, we also need to define an appropriate adaptation of the semantics definition. All accesses to r on instances of C1 need to be replaced by r.collect{b -> b.t}. Similarly, all accesses to the opposite reference o on objects of C2 need to be replaced by o.collect{b -> b.s}.*

### 8.3.3 Case Study

In this section, we use a selection of semantics-preserving coupled operations to evolve the Petri net modeling language from version 1 (see Figure 8.11 and Figure 8.12) to version 2 (see Figure 8.13 and Figure 8.14).

**Reference to Class.** To introduce the reference classes PTArc and TPArc, the coupled operation Reference to Class is applied twice. Figure 8.16 illustrates the adaptation of the metamodel, and Figure 8.17 the adaptation of the semantics definition.

Changes are highlighted in blue and by dashed boxes and solid lines. The model migration introduces reference objects for all links of the references src and dst.

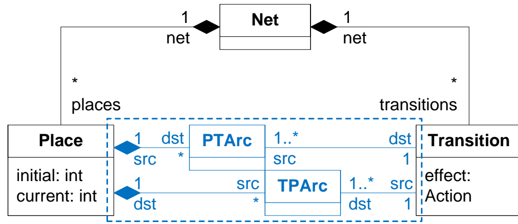



Figure 8.16: Metamodel after application of Reference to Class

```
Transition.isActivated(): boolean
  return src.collect{pt ->
    pt.src}.every{p -> p.isActivated()}

Transition.fire():
  src.collect{pt -> pt.src}.each{p ->
    p.decrement()}
  effect.execute()
  dst.collect{tp -> tp.dst}.each{p ->
    p.increment()}
```

Figure 8.17: Semantics definition after application of Reference to Class

**Refactoring of the Semantics Definition.** To simplify the semantics definition, we can refactor it by folding the collect into the each and every statement. The result of this refactoring is shown in Figure 8.18.

```
Transition.isActivated(): boolean
  return src.every{pt -> pt.src.isActivated()}

Transition.fire():
  src.each{pt -> pt.src.decrement()}
  effect.execute()
  dst.each{tp -> tp.dst.increment()}
```

Figure 8.18: Semantics definition after refactoring

**Rename.** To facilitate understanding the metamodel, a number of references need to be renamed. Figure 8.19 illustrates the metamodel adaptation and highlights which references are renamed to a new name. To be semantics-preserving, the links need to be renamed accordingly in models, and the accesses to these references need to be renamed accordingly in the semantics definition (see Figure 8.20).

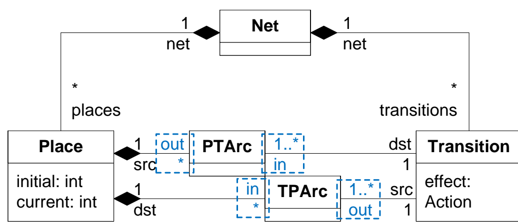



Figure 8.19: Metamodel after application of Rename

```
Transition.isActivated(): boolean
  return in.every{pt ->
    pt.src.isActivated()}

Transition.fire():
  in.each{pt -> pt.src.decrement()}
  effect.execute()
  out.each{tp -> tp.dst.increment()}
```

Figure 8.20: Semantics definition after application of Rename

**Switch Reference Composite.** PTArcs and TPArcs should not be contained by the Places from which they originate, but by the Net. As is shown in Figure 8.21, Switch Reference Composite drops the composite constraint on the references and creates composite references in Net to contain the PTArcs and TPArcs. The model migration needs to ensure that the PTArcs and TPArcs are contained by the Net. No adaptation of the semantics definition is necessary, as the existing accesses lead to the same result.

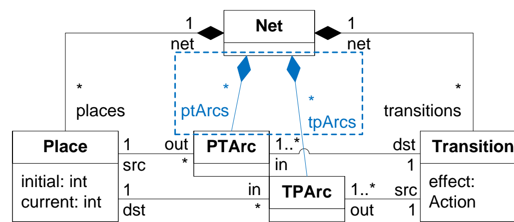



**Figure 8.21:** Metamodel after application of Switch Reference Composite

**Extract Superclass.** A common superclass Arc for PTArc and TPArc needs to be created. The adaptation of the metamodel is shown in Figure 8.22. Neither model migration nor adaptation of the semantics definition is necessary, as Extract Superclass does not affect existing objects or accesses.

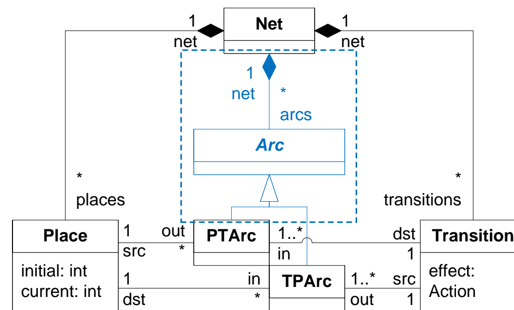



**Figure 8.22:** Metamodel after application of Extract Superclass

**Refactoring of the Semantics Definition.** To prepare the introduction of explicit weights, we need to refactor the semantics definition as shown in Figure 8.23. The operations isActivated, decrement and increment are moved from Place to PTArc or TPArc. The accesses to the operations and to current have to be adapted accordingly.

```
Transition.isActivated(): boolean
  return in.every{pt ->
    pt.isActivated()}

Transition.fire():
  in.each{pt -> pt.decrement()}
  effect.execute()
  out.each{tp -> tp.increment()}
```

```
PTArc.isActivated(): boolean
  return src.current >= 1

PTArc.decrement():
  src.current = src.current - 1

TPArc.increment():
  dst.current = dst.current + 1
```

**Figure 8.23:** Semantics definition after refactoring

**New Attribute.** The last step is to introduce the attribute weight as shown in Figure 8.13 and Figure 8.14. To allow for weights different from 1, the semantics definition has to be manually adapted to use the value of weight. By choosing 1 as a default value for weight, no model migration is needed and the semantics is preserved.

### 8.3.4 Revisiting the Library

In the last sub section, we exemplified the extension of a number of reusable coupled operations with a semantics adaptation. In the following, we revisit all the reusable coupled operations in the library (see Section 5.2 (*Library of Reusable Coupled Operations*)) and discuss whether and how they can be extended with an adaptation of the semantics definition.

**Structural Primitives.** Operations that create metamodel elements usually are constructors, i.e. extend the set of models that can be built with a metamodel. To cater for the new models, the semantics definition needs to be extended, too. However, extending the semantics definition cannot be automated, since additional knowledge is necessary to define the semantics for the new models. There are also creation operations that are refactorings, i.e. preserve the set of models, e.g. Create Enumeration. However, these operations usually are performed before operations that connect the new metamodel elements to the rest of the metamodel, thus making them available.

Operations that delete metamodel elements usually are destructors, i.e. restrict the language syntax, thereby requiring to restrict the semantics definition. In principal, restriction does not require additional information and could possibly be automated. However, the use of a deleted metamodel element in the semantics definition can be intertwined with the use of a preserved metamodel element, thereby making it difficult to automatically remove the use of the deleted metamodel element. Consequently, we would rather support the language engineer to identify parts that are no longer valid with respect to the adapted metamodel. Again, there are deletion operations that are refactorings which do not require restricting the semantics definition.

**Non-Structural Primitives.** Rename and Change Package change the fully qualified name of metamodel elements. To consistently adapt the semantics definition, the affected uses of these names have to be changed, too. When we make a class abstract, we can only restrict the semantics definition, in case the class has no subclass, which is however a degenerated case. When we drop the abstractness of a class, we need to extend the semantics definition in any case. Add Supertype requires to extend the semantics definition by using the features that are now available, whereas Remove Supertype requires to remove the uses of these features. After Make Attribute Identifier, we can use an attribute as identifier in the semantics definition. However, after Drop Attribute Identifier, we can no longer rely on the uniqueness of the attribute value in the semantics definition. Make Reference Composite and Switch Reference Composite only change the containment structure of the model, but do not affect the uses of the metamodel by the semantics definition. The same holds for Make Reference Opposite and Drop Reference Opposite.

**Specialization / Generalization Operations.** Generalization operations usually extend the language syntax and thereby require to also extend the semantics definition. Generalize Attribute and Generalize Reference require to cater for the extended multiplicities or additional possible values of attributes and references. Specialize Supertype makes new features available that need to be used in the semantics definition by the class whose supertype is specialized.

Specialization operations usually restrict the set of syntactically correct models, thereby requiring to restrict the semantics definition. Specialize Attribute and Specialize Reference require to restrict the use of multiplicities and possible values. Generalize Supertype removes inherited features from a class which can thus no longer be used in the semantics definition for that class. Specialize Composite Reference specializes the type of the composite reference and thus may lead to new available features.

**Inheritance Operations.** Pull up Feature makes a feature also available in a superclass, whereas Push down Feature removes it from the superclass. If the superclass is not abstract, we have to deal with the added or removed feature in the semantics definition. Usually, Pull up Feature leads to pulling part of the semantics definition to the superclass, whereas Push down Feature requires pushing part of the semantics definition to the subclass. The same holds for Extract Superclass and Inline Superclass which additionally create or delete the superclass. However, Fold Superclass and Unfold Superclass are more involved. Fold Superclass is performed to remove the redundancy between classes that define similar features, thereby requiring to remove the redundancy in the semantics definition. To be semantics-preserving, Unfold Superclass requires to automatically add the redundancy to the semantics definition for the subclass. Extract Subclass requires to move the part of the semantics definition that uses the extracted feature to the new subclass. Inline Subclass requires to integrate the semantics definition for the inlined subclass into the semantics definition for the superclass.

**Delegation Operations.** Extract Class requires to extend accesses to the extracted features with the reference between context class and extracted class. Afterwards, the part of the semantics definition that only relates to the extracted features has to be moved over to the extracted class. Inline Class requires to move these parts back to the context class and to remove the indirection in accesses. For Fold Class to be semantics-preserving, the semantics definition for the replaced class needs to be equivalent to the one of the replacing class. In case of Unfold Class, we just need to copy the semantics definition to the unfolded class. Move Feature over Reference requires to add an indirection to accesses to the feature in the semantics definition. Collect Feature over Reference is more involved, since it may also restrict the syntax of the modeling language.

**Replacement Operations.** Subclasses to Enumeration requires to integrate the semantics definition for the subclasses into the superclass, activating it by a switch over the enumeration attribute. Enumeration to Subclasses requires to replace a query on the enumeration attribute by an appropriate type check. After Enumeration to Subclasses, we might want to modularize the semantics definition for the superclass according to the subclasses. Reference to Class requires to introduce indirections, whereas Class to Reference requires to remove them. For Inheritance to Delegation to be semantics-preserving, we need to introduce indirections for features previously inherited from the superclass. In case of Delegation to Inheritance, we need to remove these indirections. Reference to Identifier requires to replace accesses to the reference by a helper method to resolve the identifier, whereas Identifier to Reference requires to replace the resolutions of the identifier by direct accesses to the reference.

**Merge / Split Operations.** Merge operations merge metamodel elements and thus require to merge their semantics definitions. For Merge Classes to be semantics-preserving, the semantics definition for the merged classes have to be equivalent. After Merge Features, accesses to the removed features are no longer valid and may be replaced by helper methods that derive the previous values of these features. Merge Enumerations only requires to rename the accesses to the literals of the removed enumeration.

Split operations split metamodel elements and thus require to split their semantics definition. For Split Class to be semantics-preserving, the semantics definition needs to be duplicated, and may need to be refined later on. Specialize Reference by Type replaces the access to all instances of a certain class by accesses to instances of each subclass of that class.

**Summary.** We have seen that for many reusable coupled operations it is possible to extend them with an appropriate semantics adaptation. However, it is more involved for those operations which require to remove redundancy in the semantics definition. Moreover, the reusable coupled operations can be grouped according to their impact on the language expressiveness.

*Refactoring*s preserve the set of models, possibly modulo variation, and thus allow language engineers to define a bidirectional mapping between models of different versions. As a consequence, there is also a unique way to adapt the semantics definition in order to preserve the meaning of the models. In a nutshell, the semantics adaptation can be fully automated for refactorings.

*Constructor*s increase the set of models, and thus require to extend the semantics definition to cater for the new models. However, constructors can often be extended in a semantics-preserving way, if the new models are not considered in a first step. In a second step, the language engineer needs to extend the semantics adaptation to take into account the new models. The language engineer can be supported by highlighting the places in the semantics adaptation which require extension.

*Destructor*s decrease the set of models, and thus require to restrict the semantics definition to cater for the removed models. It is often difficult to automatically remove the parts of the semantics definition that are no longer valid, since this might be intertwined with parts that are still required. However, the language engineer can be supported by highlighting the invalid parts of the semantics definition. Often, destructors are used to remove metamodel elements that are no longer used in the semantics definition anyway.

### 8.3.5 State of the Art

Rose et al. classify approaches to automate the migration of models into manual specification, operation-based and matching approaches [Rose et al., 2009].

*Manual specification* approaches provide languages tailored for model migration to manually specify the migration. Sprinkle and Karsai present a graph transformation language that requires to specify the migration only for the metamodel difference, automatically copying unaffected model elements [Sprinkle and Karsai, 2004].

Narayanan et al. present MCL (Model Change Language) to specify the metamodel changes which can be used for both model migration [Narayanan et al., 2009] and model transformation adaptation [Levendovszky et al., 2010]. When specifying the semantics as a model transformation, MCL can be used to perform semantics-preserving model migration. Flock also automatically unsets model elements which are no longer syntactically correct [Rose et al., 2010d]. However, the manual specification approaches do not provide explicit support to ensure that the specified migration is semantics-preserving.

*Operation-based* approaches specify the model migration as a sequence of coupled operations which encapsulate metamodel adaptation and model migration. Wachsmuth classifies a set of coupled operations according to semantics and instance preservation properties [Wachsmuth, 2007]. However, Wachsmuth uses the term to denote the preservation of the metamodel's semantics, i.e. the set of syntactically correct models, rather than the preservation of the modeling language's semantics, i.e. the function that maps each model to its meaning. Our approach COPE extends Wachsmuth's approach by a means to manually specify custom coupled operations and to define new reusable coupled operations (see Section 5.3 (*Language to Specify the Coupled Evolution*)). We have shown that model migration cannot be automated in certain cases, when taking the semantics of a modeling language into account (see Section 5.4 (*Limitations of Automating Model Migration*)).

*Matching* approaches try to detect a model migration based on the matching between two metamodel versions. Gruschko et al. classify primitive metamodel changes into non-breaking, breaking resolvable and breaking non-resolvable, and envision to automatically detect a model migration for breaking resolvable changes [Becker et al., 2007]. Moreover, Cicchetti et al. are able to detect complex changes in the difference between two metamodel versions [Cicchetti et al., 2008]. Garcés et al. present a matching language that allows the user to customize the matching process and to add new matching patterns [Garcés et al., 2009]. However, the matching approaches all work only on the metamodel and thus are not able detect a semantics-preserving model migration.

## 8.4 Summary

In this chapter, we extended the operation-based approach of COPE to a method for evolutionary metamodeling. This method supports not only model migration, but also other tasks that are important when evolving a modeling language: identifying metamodel changes, choosing the corresponding operations and migrating other artifacts depending on the metamodel. In the center of the method are the operations that define the intention behind the changes. The identified changes have to be implemented by operations, and the operations can be extended to also migrate artifacts other than the models. Besides the method itself, we have highlighted two sub tasks of the evolutionary method.

First, we developed an approach to identify metamodel changes by analyzing the usage of a metamodel by its models. The approach defines patterns to find deviations between the actual usage mined from the models and the expected usage derived from the metamodel. If the actual usage deviates from the expected usage, operations

can be proposed to align the metamodel better with the actual usage. To demonstrate the usefulness of the approach, we performed an empirical study that applied the approach to 7 metamodels and their models. The study showed that the actual usage does deviate from the expected usage in practice and that the deviations would often require adapting the metamodel.

Second, we developed an approach to automate the adaptation of the semantics definition after adaptation of the metamodel. More specifically, the reusable coupled operations are extended with an adaptation of the semantics definition. By consistently defining the model migration and adaptation of the semantics definition for these coupled operations, we can ensure semantics preservation by the model migration in a constructive manner. By a small case study, we demonstrated the feasibility of the approach. Moreover, we discussed how all reusable coupled operations in the library can be extended by an appropriate adaptation of the semantics definition. Future work is required to implement this approach and to apply it to a larger case study.

Chapter 9

# Summary

In this final chapter, we summarize the contributions of this thesis. We also describe our current work and possible directions for further research.

**Contents**


**9.1 Contributions** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **265**
**9.2 Outlook** . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **268**


Section 9.1 (*Contributions*) presents the contributions, and Section 9.2 (*Outlook*) the directions for future work.

## 9.1 Contributions

In this thesis, we presented our approach COPE to support the migration of models in response to metamodel adaptation—we call this the coupled evolution of metamodels and models. In particular, the approach addresses the following two challenges: First, model migration needs to be automated as far as possible in order to reduce the effort for metamodel adaptation. Second, model migration needs to preserve the semantics of a possibly unknown set of built models in order to not lose meaningful information during migration. More specifically, this thesis made the following contributions to the state of the art.

**Automatability of Model Migration in Practice.** To determine the potential for automation, we performed an empirical study on the history of two industrial metamodels from BMW Car IT [Herrmannsdoerfer et al., 2008a]. In order to do so, we classified metamodel changes according to the automatability of the corresponding model migration. The model migration is either specific to a restricted set of models, specific to the models of a single metamodel or independent of the metamodel. We found no model-specific changes, indicating that we can always specify a transformation that is able to migrate all models of these two metamodels. One third of the changes were metamodel-specific, requiring expressiveness to specify the transformation. Finally, two third of the changes were metamodel-independent, allowing to

reuse the migrating transformation across metamodels. Consequently, a practical approach to specify model migration needs to cater for both expressiveness and reuse. (see Chapter 3 (*State of the Practice: Automatability of Model Migration*))

**Cross-Space Survey on Coupled Evolution.** We analyzed existing approaches with respect to the challenges and the requirements derived from the empirical study. First, existing model migration approaches do not combine both the desired level of reuse and expressiveness. Second, most of the approaches require to specify the migration, after the metamodel changes have already been carried out, losing the intention behind the changes. However, the migration problem does not only exist in the context of metamodels and models, but also in the context of database schemas and databases, grammars and programs as well as formats and documents. To identify related approaches from the other technical spaces, we performed a systematic literature review. From the identified approaches, we extracted a feature model that helped us to classify and compare the existing approaches. We learned from other technical spaces that an operation-based approach allows us to combine expressiveness with reuse as well as to record the model migration together with the metamodel adaptation. (see Chapter 4 (*State of the Art: A Cross-Space Survey on Coupled Evolution*))

**Method for Evolutionary Metamodeling.** In our operation-based approach COPE [Herrmannsdoerfer et al., 2009a], the model migration is specified as a sequence of coupled operations. Each coupled operation encapsulates both metamodel adaptation and model migration. The operation-based approach allows us to address both challenges. First, expressiveness and reuse can be combined by different kinds of coupled operations: Custom coupled operations allow language engineers to manually express custom migrations, whereas reusable coupled operations allow them to reuse recurring migrations. Second, the model migration can be recorded as a sequence of coupled operations in order to not lose the intention behind the metamodel changes. As the coupled operations can be used to incrementally adapt the metamodel to one's needs, an operation-based approach thus enables a method for the evolutionary development of metamodels. (see Section 5.1 (*COPE in a Nutshell*) and Section 8.1 (*The Process of Evolutionary Metamodeling*))

**Language to Encode Coupled Operations.** To be able to specify coupled operations, we developed a language embedded into the general-purpose scripting language Groovy [Herrmannsdoerfer et al., 2008b]. Since a coupled operation usually implements a rather small change, the language only requires to specify the difference for both metamodel and model. Therefore, the language provides a complete set of primitives to express both metamodel adaptation and model migration. The primitives operate on a generic instance model which decouples a model from its metamodel in order to ease migration. However, the language enforces conformance of a model to its metamodel at the boundaries of a coupled operation. Expressiveness is provided by embedding the primitives into the Turing-complete scripting language. Reuse is provided by using the abstraction mechanism of functions that allow language engineers to specify a coupled operation independently of a metamodel by means of parameters. (see Section 5.3 (*Language to Specify the Coupled Evolution*))

**Library of Reusable Coupled Operations.** While performing the case studies with COPE, we built up a library of 61 reusable coupled operations using the coupled evolution language [Herrmannsdoerfer et al., 2010b]. These reusable coupled operations have proven sufficient enough to cover most model migrations that occur in practice. To facilitate finding the appropriate coupled operation for a certain change at hand, the library is organized according to a number of criteria. First, the reusable coupled operations can be grouped according to their semantics, allowing to find an operation based on one's intention. Second, they can be characterized by their impact on the set of conforming models or on existing models, permitting to reason about model migration. Third, reusable coupled operations can be related to each other by an inverse relationship, allowing to revert both the metamodel adaptation and the model migration. (see Section 5.2 (*Library of Reusable Coupled Operations*))

**Limitations of Automating Model Migration.** In cases where the migration is specific to a restricted set of models, the model migration cannot inherently be automated. We formally characterized these cases by proving semantics preservation through coupled operations [Herrmannsdoerfer and Ratiu, 2009, Herrmannsdoerfer and Ratiu, 2010]. Model-specific migration is necessary if the metamodel change requires to refine the semantics of existing models and there are multiple models by which an existing model can be refined. Then, model-specific information may be necessary to choose the correct model. To support model-specific migrations, we can either determine the effort for manual migration by analyzing existing models, we can enable interactive migration, or we can allow language users to refine a model migration for a certain subset of existing models. (see Section 5.4 (*Limitations of Automating Model Migration*))

**Metamodel to Record Coupled Operations.** From our experience, language engineers prefer to use the metamodel editor over specifying the coupled evolution in the language. Consequently, COPE provides further abstraction from this language by a non-invasive integration into a metamodel editor [Herrmannsdoerfer, 2011]. The integration is based on a versioning metamodel to record the history of a metamodel as a sequence of coupled operations [Herrmannsdoerfer, 2009]. The history metamodel is expressive enough to capture all changes to the metamodel as well as to group changes to custom and reusable coupled operations. A migrator specified in the language can be automatically generated from a history model conforming to this metamodel. COPE provides advanced functions to inspect, refactor and reverse engineer the history model based on the history metamodel. (see Chapter 6 (*Tool Support*))

**Evaluation through Case Studies.** We have implemented COPE based on the Eclipse Modeling Framework (EMF) which is one of the most widely used modeling frameworks. Thereby, it was possible to perform six real-life case studies with the implemented tool. To deeply evaluate the approach, we performed different kinds of case studies: reverse engineering, forward engineering and comparison case studies. First, we reverse engineered the history model of the Palladio Component Model (PCM) [Herrmannsdoerfer et al., 2009a] as well as the history models for all metamodels from the Graphical Modeling Framework (GMF) [Herrmannsdoerfer et al., 2009c]. Second, we forward engineered the history model

of the Quamoco quality metamodel as well as the Unicase UML-like metamodel. Third, we compared COPE to other model transformation and migration tools in the Transformation Tool Contest (TTC) [Herrmannsdoerfer, 2010] as well as by means of a case study performed together with the authors of other migration tools [Rose et al., 2010a]. The case studies demonstrate that COPE addresses both challenges. First, all the case studies confirmed the results of the empirical study that most of the migration can be covered by reusable coupled operations. Second, the comparison case studies indicated that COPE is more likely to lead to a semantics-preserving model migration. (see Chapter 7 (*Case Studies*))

**Metamodel Usage Analysis for Identifying Metamodel Improvements.** After the language engineers released a modeling language, they may want to validate the metamodel. An approach to validate the usability of the metamodel is to analyze the models built with the metamodel. Therefore, we defined a number of patterns to compare the actual usage in existing models to the expected usage derived from the metamodel [Herrmannsdoerfer et al., 2010a]. Using these patterns, we can identify deviations between actual and expected usage and recommend operations to better align the metamodel with its usage. To identify the potential of this method, we performed an empirical study applying the patterns to seven metamodels and their models. We found a significant number of deviations, many of which could be used to improve the usability of the metamodel. The patterns helped either to restrict the metamodel or to extend it by introducing first-class constructs for often used values. (see Section 8.2 (*Metamodel Usage Analysis for Identifying Metamodel Improvements*))

**Semantics-Preserving Model Migration.** When evolving a modeling language, we also need to adapt the semantics definition in response to metamodel adaptation. In case the adaptation of the semantics definition is consistent with the model migration, we can constructively ensure that the semantics is preserved during model migration [Herrmannsdoerfer and Koegel, 2010b]. Consequently, reusable coupled operations need to be extended with an appropriate adaptation of the semantics definition. We demonstrated the applicability of the approach on the well-known example evolution of the petri net modeling language. Finally, we showed that many reusable coupled operations from the library can be easily extended with an appropriate semantics adaptation. For custom coupled operations, which however occur rarely in practice, the adaptation of the semantics definition needs to be specified manually. (see Section 8.3 (*Towards Semantics-Preserving Model Migration*))

## 9.2 Outlook

While the coupled evolution approach presented in this thesis addresses major shortcomings of existing approaches, there are still open problems that need to be solved. This section discusses possible improvements of the presented approach and illustrates directions for further research on coupled evolution in particular and evolutionary metamodeling in general.

**Dissemination – Eclipse Project Edapt.** The community around the Eclipse Mod-

eling Framework (EMF) has recognized the need for tools supporting metamodel adaptation and model migration. However, there is currently no adequate tool support for model migration in EMF. Therefore, we have been invited to present our tool COPE at several Eclipse conferences and demo camps. There has been quite some interest in COPE, and COPE already has a number of active users. Due to the large interest, we decided to make COPE open source under the Eclipse Public License (EPL)[1]. Therefore, we applied for the Eclipse project Edapt[2] which has been accepted. In order to further increase the attention for COPE, we are currently making the tool available through the Eclipse project Edapt. There are plans to integrate the different projects around EMF into a workbench for developing EMF-based modeling languages. We plan to propose COPE as a solution to support the maintenance of modeling languages within this workbench.

**Verification and Validation of Coupled Operations.** Currently, syntax preservation of a coupled operation can only be checked dynamically, while executing it on a certain model. However, it is difficult to constructively ensure syntax preservation, as COPE decouples a model from its metamodel within a coupled operation to be able to specify only the difference for both metamodel and model. To some degree, reusable coupled operations constructively ensure syntax preservation, since they provide a consistent and well-tested couple of metamodel adaptation and model migration. The problem is more important for custom coupled operations which require to manually specify an appropriate model migration for a recorded metamodel adaptation. There are two solutions to this problem. First, we can develop a static analysis to verify syntax preservation in a model-independent way. Second, we can refine the coupled evolution language with a syntax that ensures syntax preservation in a more constructive manner. We plan to evaluate the two solutions in order to statically ensure syntax preservation.

In contrast to the verification of properties, validation is more concerned with whether the migration performs as intended. Usually, the primary goal of model migration is to preserve the semantics of existing models. In this thesis, we already outlined an approach to ensure semantics preservation in a constructive manner. This approach mostly targets reusable coupled operations which are extended by an adaptation of the semantics definition. However, there is no support to ensure semantics preservation for custom coupled operations. Moreover, the approach does not work for modeling languages which do not have an explicit semantics definition. As an advantage of the operation-based approach, we can independently validate semantics preservation for each coupled operation. In order to validate coupled operations, we plan to develop a framework for the rigorous testing of model migrations. This may include specific coverage criteria as well as a method to derive new test models.

**Migration of other Artifacts.** In this thesis, we were mostly concerned with the migration of models in response to metamodel adaptation. However, there are also other artifacts—like e.g. editors and interpreters—which depend on the metamodel and which thus have to be migrated. According to our terminology, also the models defining the concrete syntax and semantics depend on the metamodel. We first

[1]see EPL web site: `http://www.eclipse.org/legal/epl-v10.html`
[2]see Edapt web site: `http://www.eclipse.org/edapt/`

focused on model migration, as the number of models of a successful modeling language typically outnumbers the number of other artifacts. In this thesis, we already showed that the migration of other artifacts by coupled operations is feasible: In a case study, we used the coupled evolution language to express the migration of change histories, and we extended reusable coupled operations with an adaptation of the semantics definition to ensure semantics preservation. We plan to refine the approach to be able to specify the migration of any other artifacts depending on the metamodel. Therefore, we need to extend the coupled evolution language with additional constructs to specify the migration of other artifacts and we need to elaborate our mechanism to extend existing reusable coupled operations. Our final goal is to provide an integrated approach to support the maintenance of modeling languages.

**Language Profiling.** In this thesis, we examined an approach to identify metamodel improvements by analyzing models built with the metamodel. Through an empirical study, we showed that—even in the case of mature languages—the analysis of models can reveal issues with the modeling language. Consequently, we are convinced that the analysis of models built with a modeling language is a useful tool for evolutionary metamodeling. However, the presented approach is restricted to very simple usage patterns and to EMF as a modeling framework. Due to the results of the empirical study, it is promising to extend the approach to a workbench for language profiling, i.e. for analyzing the use of the language in order to assess its quality. First, we can refine the presented analyses by fine-tuning them according to the results from the empirical study. Second, we need to add new analyses which are currently missing in the catalog. For instance, we might even be able to automatically identify constraints which are not yet implemented. Third, we can build a language that allows the language engineer to easily specify new patterns of language use. Forth, we might want to apply the approach also to modeling languages that are not built using EMF. Then, we can also analyze models of languages that are more widely used in industry like e.g. Matlab Simulink. Fifth, we envision as a future direction of research the definition of techniques that anonymize the content of the model and send the language engineers only a limited information needed for analyzing the language use. Then, we can also profile modeling languages that are used by different organizations and where the models carry intellectual property.

**Quality of Modeling Languages.** Through our case studies, we have found out that a significant number of changes are performed to improve the quality of metamodels. To avoid these changes in the first place, we have to get a better understanding about the quality of modeling languages. More specifically, we have to answer the following questions: What does quality mean for metamodels? How can we measure the quality of metamodels? How can we improve the quality of metamodels? To answer these questions, we could apply the quality modeling approach developed in the Quamoco[3] project to modeling languages. Therefore, we need to define a quality model with factors that influence the quality of modeling languages as well as measures that reliably quantify the factors. To build such a quality model, we need to search the literature for guidelines and metrics that are defined for modeling languages or that can at least be transferred to modeling languages. From the quality

[3]see Quamoco web site: `http://www.quamoco.de`

model, we can then generate a guideline that can be used to support engineering of high-quality modeling languages. Moreover, the quality model can be used to automatically assess the modeling language in a continuous manner. Thereby, we can early identify and avoid defects in the metamodels and thus secure the quality of the modeling language. Finally, the approach presented in this thesis can be used to implement metamodel improvements.

Appendix A

# Papers Excluded from the Survey

This section lists the papers which were excluded from the survey presented in Chapter 4 (*State of the Art: A Cross-Space Survey on Coupled Evolution*) based on the exclusion criteria.

## A.1 Excluded Papers Within the Relevant Domain

Following is a list of papers, which were within the domain of the survey, yet were rejected as they fall outside the scope definition. For each publication, we list its bibliographic information, followed by the reason for rejection.

- Amiel, E., Bellosta, M.-J., Dujardin, E., and Simon, E. (1994). Supporting exceptions to schema consistency to ease schema evolution in oodbms. In *VLDB '94: Proceedings of the 20th International Conference on Very Large Data Bases*, pages 108–119, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. **Focus on schema evolution, no database migration: no coupling**.
- Andrade, L. F. and Fiadeiro, J. L. (2001). Coordination technologies for managing information system evolution. In *CAiSE '01: Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, pages 374–387, London, UK. Springer-Verlag. **Does not adress coupled evolution, rather system evolution in general**.
- Andrikopoulos, V., Benbernou, S., and Papazoglou, M. P. (2008). Managing the evolution of service specifications. In *CAiSE '08: Proceedings of the 20th international conference on Advanced Information Systems Engineering*, pages 359–374, Berlin, Heidelberg. Springer-Verlag. **Extension is not completely defined by the intensional definition**.
- Ariav, G. (1991). Temporally oriented data definitions: managing schema evolution in temporally oriented databases. *Data Knowl. Eng.*, 6(6):451–467. **Primary focus on providing temporal features to schema of temporally oriented databases, there is little discussion on a coupling to the temporally oriented content**.

- Banerjee, J., Chou, H.-T., Garza, J. F., Kim, W., Woelk, D., Ballou, N., and Kim, H.-J. (1987). Data model issues for object-oriented applications. *ACM Trans. Inf. Syst.*, 5(1):3–26. **Several extensions for ORION, one is schema evolution, but only without real migration**.
- Beech, D. and Mahbod, B. (1988). Generalized version control in an object-oriented database. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 14–22, Washington, DC, USA. IEEE Computer Society. **On database versioning, not schema versioning**.
- Bernstein, P. A. (2003). Applying model management to classical meta data problems. In *CIDR*. **No migration of elements**.
- Bernstein, P. A., Haas, L. M., Jarke, M., Rahm, E., and Wiederhold, G. (2000). Panel: Is generic metadata management feasible? In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 660–662, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. **Summary of a panel discussion, out of scope**.
- Bertino, E. (1992). A view mechanism for object-oriented databases. In *EDBT '92: Proceedings of the 3rd International Conference on Extending Database Technology*, pages 136–151, London, UK. Springer-Verlag. **On views, no focus on coupling**.
- Blaha, M. and Premerlani, W. (1996). A catalog of object model transformations. In *WCRE '96: Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, page 87, Washington, DC, USA. IEEE Computer Society. **Describes a library of transformations on class diagrams, but without the co-transformation of instances**.
- Bratsberg, S. E. (1992). Unified class evolution by object-oriented views. In *ER '92: Proceedings of the 11th International Conference on the Entity-Relationship Approach*, pages 423–439, London, UK. Springer-Verlag. **Focused on preventing coupled evolution**.
- Cabasino, S., Paolucci, P. S., and Todesco, G. M. (1992). Dynamic parsers and evolving grammars. *SIGPLAN Not.*, 27(11):39–48. **Support the addition of context-sensitive rules by dynamically evolving the grammar when parsing a program**.
- Casais, E. (1992). An incremental class reorganization approach. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 114–132, London, UK. Springer-Verlag. **Evolution of object-oriented software, about recommending refactorings**.
- Chou, H.-T. and Kim, W. (1988). Versions and change notification in an object-oriented database system. In *DAC '88: Proceedings of the 25th ACM/IEEE Design Automation Conference*, pages 275–281, Los Alamitos, CA, USA. IEEE Computer Society Press. **Primarily focused on versions and change notifications, only briefly mentions dynamic schema evolution, no coupling**.
- Clifford, J. and Croker, A. (1987). The historical relational data model (hrdm) and algebra based on lifespans. In *Proceedings of the Third International Conference on Data Engineering*, pages 528–537, Washington, DC, USA. IEEE Computer

Society. **On database versioning, but not schema versioning**.

- Cohen, Y. and Feldman, Y. A. (2003). Automatic high-quality reengineering of database programs by abstraction, transformation and reimplementation. *ACM Trans. Softw. Eng. Methodol.*, 12(3):285–316. **Migration of legacy data to modern database systems**.
- Drumm, C., Schmitt, M., Do, H.-H., and Rahm, E. (2007). Quickmig: automatic schema matching for data migration projects. In *CIKM '07: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 107–116, New York, NY, USA. ACM. **Migration of legacy data into modern database systems**.
- Ewald, C. A. and Orlowska, M. E. (1993). A procedural approach to schema evolution. In *CAiSE '93: Proceedings of Advanced Information Systems Engineering*, pages 22–38, London, UK. Springer-Verlag. **Preserve well-formedness during schema evolution, no co-evolution**.
- Fan, H. and Poulovassilis, A. (2004). Schema evolution in data warehousing environments – a schema transformation-based approach. In Atzeni, P., Chu, W., Lu, H., Zhou, S., and Ling, T. W., editors, *Conceptual Modeling – ER 2004*, volume 3288 of *Lecture Notes in Computer Science*, pages 639–653. Springer Berlin / Heidelberg. **About evolution of schema integration framework on evolution of schema modeling language, such that schema integration can be reused: Extension not completely defined by intensional definition**.
- Gibbs, S., Casais, E., Nierstrasz, O., Pintado, X., and Tsichritzis, D. (1990). Class management for software communities. *Commun. ACM*, 33(9):90–103. **Only mentions class evolution as problem in software communities, no real approach**.
- Grandi, F. (2004). Svmgr: A tool for the management of schema versioning. In Atzeni, P., Chu, W., Lu, H., Zhou, S., and Ling, T. W., editors, *Conceptual Modeling – ER 2004*, volume 3288 of *Lecture Notes in Computer Science*, pages 860–861. Springer Berlin / Heidelberg. **Only a short tool description**.
- Hainaut, J.-L. (2006). The transformational approach to database engineering. In Lämmel, R., Saraiva, J., and Visser, J., editors, *Generative and Transformational Techniques in Software Engineering*, volume 4143 of *Lecture Notes in Computer Science*, pages 95–143. Springer Berlin / Heidelberg. **Hainaut states in the conclusion: "Several problems still are to be addressed ... How can data structure transformations be propagated to the other components of the information system, notably the data (data conversion)?"**.
- Ipser, Jr., E. A. (1992). Exploratory language design. *SIGPLAN Not.*, 27(4):41–50. **Introduces evolutionary language design, but does not provide an approach to migrate language utterances**.
- Kim, W., Banerjee, J., Chou, H.-T., and Garza, J. F. (1990a). Object-oriented database support for cad. *Comput. Aided Des.*, 22(10):469–479. **Several extensions for ORION, one is schema evolution, but only without real migration**.
- Kim, W., Garza, J. F., Ballou, N., and Woelk, D. (1990b). Architecture of the orion next-generation database system. *IEEE Trans. on Knowl. and Data Eng.*,

2(1):109–124. **A general description of ORION, it only briefly discusses dynamic schema evolution in ORION. Not relevant enough for the survey, the ORION features are already discussed in other included papers**.

- Li, Q. and McLeod, D. (1994). Conceptual database evolution through learning in object databases. *IEEE Trans. on Knowl. and Data Eng.*, 6(2):205–224. **The paper discusses the direction of ontoloware and does not explicitly talk about instance migration**.
- Liu, L., Zicari, R., Hürsch, W., and Lieberherr, K. J. (1997). The role of polymorphic reuse mechanisms in schema evolution in an object-oriented database. *IEEE Trans. on Knowl. and Data Eng.*, 9(1):50–67. **Migration of queries in response to schema evolution**.
- López, J.-R. and Olivé, A. (2000). A framework for the evolution of temporal conceptual schemas of information systems. In *CAiSE '00: Proceedings of the 12th International Conference on Advanced Information Systems Engineering*, pages 369–386, London, UK. Springer-Verlag. **Propagation of evolution from conceptual to logical schemas to extensions and programs, no co-evolution**.
- Markowitz, V. M. and Makowsky, J. A. (1988). Incremental restructuring of relational schemas. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 276–284, Washington, DC, USA. IEEE Computer Society. **Markowitz et al. state: "We assume in this paper that the database state is empty. The coupling of schema restructuring manipulations with state mappings is investigated in [26]."**.
- McBrien, P. and Poulovassilis, A. (1997). A formal framework for er schema transformation. In *ER '97: Proceedings of the 16th International Conference on Conceptual Modeling*, pages 408–421, London, UK. Springer-Verlag. **Paper focuses on schema integration, there is no evolution aspect involved**.
- McBrien, P. and Poulovassilis, A. (1998). A formalisation of semantic schema integration. *Inf. Syst.*, 23(5):307–334. **Papers focuses on schema integration, there is no evolution aspect involved**.
- McBrien, P. and Poulovassilis, A. (1999). Automatic migration and wrapping of database applications - a schema transformation approach. In *ER '99: Proceedings of the 18th International Conference on Conceptual Modeling*, pages 96–113, London, UK. Springer-Verlag. **Papers focuses on schema integration, there is no evolution aspect involved**.
- McKenzie, E. and Snodgrass, R. T. (1990). Schema evolution and the relational algebra. *Inf. Syst.*, 15(2):207–232. **Extends relational algebra with a time concept to handle evolution of data and schemas, it has little focus on establishing a coupling**.
- Melnik, S. (2005). Model management: First steps and beyond. In *BTW. Volume 65 of LNI., GI (2005) 455*. **Does not focus on coupled evolution, only mentions it as an issue in model management**.
- Moerkotte, G. and Zachmann, A. (1993). Towards more flexible schema management in object bases. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 174–181, Washington, DC, USA. IEEE Computer Soci-

ety. **On preserving schema consistency during evolution**.

- Narayanaswamy, K. and Rao, K. V. B. (1988). An incremental mechanism for schema evolution in engineering domains. In *Proceedings of the Fourth International Conference on Data Engineering*, pages 294–301, Washington, DC, USA. IEEE Computer Society. **On schema evolution, which does not affect existing data, thus preventing the need for database migration, thus not on coupled evolution**.
- Norrie, M. C., Steiner, A., Würgler, A., and Wunderli, M. (1996). A model for classification structures with evolution control. In *ER '96: Proceedings of the 15th International Conference on Conceptual Modeling*, pages 456–471, London, UK. Springer-Verlag. **Focus on evolution of objects, not their structure description. Thereby no vertical coupling.**
- Odberg, E. (1994). Category classes: flexible classification and evolution in object-oriented databases. In *CAiSE '94: Proceedings of the 6th international conference on Advanced information systems engineering*, pages 406–420, Secaucus, NJ, USA. Springer-Verlag New York, Inc. **Evolution of objects (and their classification) in the extension, no evolution of the classification scheme**.
- Oertly, F. and Schiller, G. (1989). Evolutionary database design. In *Proceedings of the Fifth International Conference on Data Engineering*, pages 618–624, Washington, DC, USA. IEEE Computer Society. **Data migration is not discussed**.
- Olivé, A., Costal, D., and Sancho, M.-R. (1999). Entity evolution in isa hierarchies. In *ER '99: Proceedings of the 18th International Conference on Conceptual Modeling*, pages 62–80, London, UK. Springer-Verlag. **Focus on entity evolution, no instance adaptation: no coupling**.
- Osborn, S. L. (1989). The role of polymorphism in schema evolution in an object-oriented database. *IEEE Trans. on Knowl. and Data Eng.*, 1(3):310–317. **Migration of queries in response to schema evolution**.
- Peters, R. J. and Özsu, M. T. (1995). Axiomatization of dynamic schema evolution in objectbases. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 156–164, Washington, DC, USA. IEEE Computer Society. **Peters et al. state: "Due to space restrictions, change propagation is not addressed in this paper."**.
- Peters, R. J. and Özsu, M. T. (1997). An axiomatic model of dynamic schema evolution in objectbase systems. *ACM Trans. Database Syst.*, 22(1):75–114. **Discusses a formalization of dynamic schema evolution, there is little discussion of evolution propagation through coupling**.
- Poulovassilis, A. and Mc. Brien, P. (1998). A general formal framework for schema transformation. *Data Knowl. Eng.*, 28(1):47–71. **Papers focuses on schema integration, there is no evolution aspect involved**.
- Shapiro, M. (1989). Persistence and migration for C++ objects. In *Proceedings of the 1989 European Conference on Object-Oriented Programming, East Midland Conference Centre, University of Nottingham, 10-14 July 1989*, page 191. Cambridge University Press. **Migration of runtime C++ objects from a node to another in a distributed setting**.

- Shu, N. C., Housel, B. C., and Lum, V. Y. (1975). Convert: a high level translation definition language for data conversion. *Commun. ACM*, 18(10):557–567. **Focus on evolution of data, not their schema. Thereby no vertical coupling.**
- Takahashi, J. (2002). Hybrid relations for database schema evolution. In *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International*, pages 465–470. IEEE. **Proposes hybrid relations to prevent the need for a coupling at evolution**.
- Tewksbury, L., Moser, L., and Melliar-Smith, P. (2001). Live upgrades of corba applications using object replication. In *ICSM '01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 488, Washington, DC, USA. IEEE Computer Society. **Migration of objects in a running application when their classes change**.
- Thiran, P., Hainaut, J.-L., Houben, G.-J., and Benslimane, D. (2006). Wrapper-based evolution of legacy information systems. *ACM Trans. Softw. Eng. Methodol.*, 15(4):329–359. **Integration of legacy systems into modern systems**.
- Thiran, P., Houben, G.-J., Hainaut, J.-L., and Benslimane, D. (2004). Updating legacy databases through wrappers: Data consistency management. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 58–67, Washington, DC, USA. IEEE Computer Society. **Integration of legacy systems into modern systems**.
- Tratt, L. (2008). Evolving a dsl implementation. pages 425–441. **Avoids migration of programs written with the evolving DSL**.
- van Deursen, A. and Klint, P. (1998). Little languages: little maintenance. *Journal of Software Maintenance*, 10(2):75–92. **Effort for program evolution rather than that for language evolution**.
- Zabback, P., Onyuksel, I., Scheuermann, P., and Weikum, G. (1998). Database reorganization in parallel disk arrays with i/o service stealing. *IEEE Trans. on Knowl. and Data Eng.*, 10(5):855–858. **About load balancing during migration rather than about migration itself**.

## A.2 Excluded Papers Outside the Relevant Domain

This section enumerates the papers we rejected for falling outside the considered domain. Each subsection is devoted to a particular domain. Note that these are not intended to be complete lists for any of the domains, they merely enumerate the papers considered for the survey.

### A.2.1 Process Evolution

- Agostini, A. and Michelis, G. D. (2000). Improving flexibility of workflow management systems. In *Business Process Management, Models, Techniques, and Empirical Studies*, pages 218–234, London, UK. Springer-Verlag.
- Edmond, D. and ter Hofstede, A. H. M. (2000). A reflective infrastructure for

workflow adaptability. *Data & Knowledge Engineering*, 34(3):271 – 304.

- Ellis, C. A. and Keddara, K. (2000). A workflow change is a workflow. In *Business Process Management, Models, Techniques, and Empirical Studies*, pages 201–217, London, UK. Springer-Verlag.
- Han, Y. and Sheth, A. (1998). On adaptive workflow modeling. In *Proceedings of the 4th International Conference on Information Systems Analysis and Synthesis*, pages 108–116. Orlando, Florida.
- Heinl, P., Horn, S., Jablonski, S., Neeb, J., Stein, K., and Teschke, M. (1999). A comprehensive approach to flexibility in workflow management systems. In *Proceedings of the international joint conference on Work activities coordination and collaboration*, WACC '99, pages 79–88, New York, NY, USA. ACM.
- Jaccheri, L., Larsen, J., and Conradi, R. (1992). Software process modeling and evolution in epos. In *Software Engineering and Knowledge Engineering, 1992. Proceedings., Fourth International Conference on*, pages 574 –581.
- Joeris, G. and Herzog, O. (1998). Managing evolving workflow specifications. In *Cooperative Information Systems, 1998. Proceedings. 3rd IFCIS International Conference on*, pages 310 –319.
- Kradolfer, M. and Geppert, A. (1999). Dynamic workflow schema evolution based on workflow type versioning and workflow migration. In *Cooperative Information Systems, 1999. CoopIS '99. Proceedings. 1999 IFCIS International Conference on*, pages 104 –114.
- Liu, C. and Conradi, R. (1993). Automatic replanning of task networks for process model evolution in EPOS. In *Proceedings of the 4th European Software Engineering Conference on Software Engineering*, ESEC '93, pages 434–450, London, UK. Springer-Verlag.
- Reichert, M. and Dadam, P. (1998). ADEPTflex – supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems*, 10:93–129. 10.1023/A:1008604709862.
- Reichert, M., Rinderle, S., and Dadam, P. (2003). On the common support of workflow type and instance changes under correctness constraints. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE*, volume 2888 of *Lecture Notes in Computer Science*, pages 407–425. Springer Berlin / Heidelberg. 10.1007/978-3-540-39964-3_26.
- Rinderle, S., Reichert, M., and Dadam, P. (2003). Evaluation of correctness criteria for dynamic workflow changes. In *Proceedings of the 2003 international conference on Business process management*, BPM'03, pages 41–57, Berlin, Heidelberg. Springer-Verlag.
- van der Aalst, W. M. P. and Basten, T. (2002). Inheritance of workflows: an approach to tackling problems related to change. *Theoretical Computer Science*, 270(1-2):125 – 203.
- van der Aalst, W. M. P., Basten, T., Verbeek, H. M. W., Verkoulen, P. A. C., and Voorhoeve, M. (1999). Adaptive workflow: On the interplay between flexibility and support. In Filipe, J. and Cordeiro, J., editors, *Proceedings of the*

*first International Conference on Enterprise Information Systems*, volume 2, pages 353–360. Setúbal, Portugal.

### A.2.2 Software Evolution

- Balaban, I., Tip, F., and Fuhrer, R. (2005). Refactoring support for class library migration. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 265–279, New York, NY, USA. ACM.
- Buckley, J., Mens, T., Zenger, M., Rashid, A., and Kniesel, G. (2005). Towards a taxonomy of software change: Research articles. *J. Softw. Maint. Evol.*, 17:309–332.
- Demeyer, S., Ducasse, S., and Nierstrasz, O. (2000). Finding refactorings via change metrics. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 166–177, New York, NY, USA. ACM.
- Godfrey, M. W. and Zou, L. (2005). Using origin analysis to detect merging and splitting of source code entities. *IEEE Trans. Softw. Eng.*, 31:166–181.
- Henkel, J. and Diwan, A. (2005). Catchup!: capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 274–283, New York, NY, USA. ACM.
- Lehman, M. M. and Ramil, J. F. (2003). Software evolution: background, theory, practice. *Inf. Process. Lett.*, 88:33–44.
- Lieberherr, K. J. and Xiao, C. (1993). Object-oriented software evolution. *IEEE Trans. Softw. Eng.*, 19:313–343.
- Lum, V., Dadam, P., Erbe, R., Guenauer, J., Pistor, P., Walch, G., Werner, H., and Woodfill, J. (1984). Designing dbms support for the temporal dimension. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, SIGMOD '84, pages 115–130, New York, NY, USA. ACM.
- Opdyke, W. F. (1992). Refactoring object-oriented frameworks. Technical report, Champaign, IL, USA.
- Roock, S. and Havenstein, A. (2002). Refactoring tags for automatic refactoring of framework. In *In XP'02: Proceedings of Extreme Programming Conference*, pages 182–185.
- Tokuda, L. and Batory, D. (2001). Evolving object-oriented designs with refactorings. *Automated Software Engg.*, 8:89–120.
- Tourwé, T. and Mens, T. (2003). Automated support for framework-based software evolution. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, pages 148–, Washington, DC, USA. IEEE Computer Society.

### A.2.3 Ontology Evolution


- Noy, N. F. and Klein, M. (2004). Ontology evolution: Not the same as schema evolution. *Knowl. Inf. Syst.*, 6:428–440.
- Pittas, N., Jones, A. C., and Gray, W. A. (2001). Evolution support in large-scale interoperable systems: a metadata driven approach. In *Proceedings of the 12th Australasian database conference*, ADC '01, pages 161–168, Washington, DC, USA. IEEE Computer Society.
- Plessers, P., De Troyer, O., and Casteleyn, S. (2005). Event-based modeling of evolution for semantic-driven systems. In Pastor, O. and Falcão e Cunha, J., editors, *Advanced Information Systems Engineering*, volume 3520 of *Lecture Notes in Computer Science*, pages 49–62. Springer Berlin / Heidelberg. 10.1007/11431855_6.


### A.2.4 Difference Calculation & Representation


- Cicchetti, A., Ruscio, D. D., and Pierantonio, A. (2007). A metamodel independent approach to difference representation. *Journal of Object Technology*, 6(9):165–185.
- Rivera, J. E. and Vallecillo, A. (2008). Representing and operating with model differences. In Aalst, W., Mylopoulos, J., Sadeh, N. M., Shaw, M. J., Szyperski, C., Paige, R. F., and Meyer, B., editors, *Objects, Components, Models and Patterns*, volume 11 of *Lecture Notes in Business Information Processing*, pages 141–160. Springer Berlin Heidelberg. 10.1007/978-3-540-69824-1_9.
- Treude, C., Berlik, S., Wenzel, S., and Kelter, U. (2007). Difference computation of large models. In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 295–304, New York, NY, USA. ACM.


### A.2.5 Schema Matching & Integration


- Abiteboul, S., Cluet, S., and Milo, T. (2002). Correspondence and translation for heterogeneous data. *Theor. Comput. Sci.*, 275:179–213.
- Chen, Y. and Benn, W. (1999). Integrating heterogeneous oo schemas. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, page 101.
- Del Fabro, M. D. and Valduriez, P. (2007). Semi-automatic model integration using matching transformations and weaving models. In *Proceedings of the 2007 ACM symposium on Applied computing*, SAC '07, pages 963–970, New York, NY, USA. ACM.
- McBrien, P. and Poulovassilis, A. (2002). Schema evolution in heterogeneous database architectures, a schema transformation approach. In *Proceedings of the 14th International Conference on Advanced Information Systems Engineering*, CAiSE '02, pages 484–499, London, UK, UK. Springer-Verlag.
- Milo, T. and Zohar, S. (1998). Using schema matching to simplify heteroge-

neous data translation. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 122–133, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

- Navathe, S. B. (1980). Schema analysis for database restructuring. *ACM Trans. Database Syst.*, 5:157–184.
- Rahm, E. and Bernstein, P. A. (2001). A survey of approaches to automatic schema matching. *The VLDB Journal*, 10:334–350.
- Wiederhold, G. (1992). Mediators in the architecture of future information systems. *Computer*, 25:38–49.

# Bibliography

[W3C, 2008] (2008). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C. `http://www.w3.org/TR/REC-xml/`. (cited on p 96)

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (cited on p 29)

[Al-Jadir and Léonard, 1998] Al-Jadir, L. and Léonard, M. (1998). Multiobjects to ease schema evolution in an OODBMS. In Ling, T. W., Ram, S., and Lee, M.-L., editors, *Conceptual Modeling - ER 98, 17th International Conference on Conceptual Modeling, Singapore, November 16-19, 1998, Proceedings*, volume 1507 of *Lecture Notes in Computer Science*, pages 316–333. Springer. (cited on p 92)

[Allen and Cartwright, 2002] Allen, E. and Cartwright, R. (2002). The case for run-time types in generic Java. In *PPPJ '02/IRE '02: Proceedings of the inaugural conference on the Principles and Practice of programming, 2002 and Proceedings of the second workshop on Intermediate representation engineering for virtual machines, 2002*, pages 19–24, Maynooth, County Kildare, Ireland, Ireland. National University of Ireland. (cited on p 18)

[Ambler and Sadalage, 2006] Ambler, S. W. and Sadalage, P. J. (2006). *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional. (cited on pp 90, 232)

[Andany et al., 1991] Andany, J., Léonard, M., and Palisser, C. (1991). Management of schema evolution in databases. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 161–170, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. (cited on p 93)

[Atkinson and Kühne, 2007] Atkinson, C. and Kühne, T. (2007). A tour of language customization concepts. *Advances in Computers*, 70:105–161. (cited on p 253)

[AUTOSAR Development Partnership, 2008] AUTOSAR Development Partnership (2008). AUTOSAR Specification V3.1. (cited on p 17)

[Banerjee et al., 1987] Banerjee, J., Kim, W., Kim, H.-J., and Korth, H. F. (1987). Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Rec.*, 16(3):311–322. (cited on pp 92, 110, 116, 119, 120, 122, 123, 125, 126, 129, 130, 135)

[Becker et al., 2007] Becker, S., Goldschmidt, T., Gruschko, B., and Koziolek, H. (2007). A process model and classification scheme for semi-automatic meta-model evolution. In *Proc. 1st Workshop MDD, SOA und IT-Management (MSI'07)*, pages 35–46. GiTO-Verlag. (cited on pp 57, 98, 116, 119, 120, 122, 123, 125, 126, 129, 263)

[Benatallah, 1999] Benatallah, B. (1999). A unified framework for supporting dynamic schema evolution in object databases. In Akoka, J., Bouzeghoub, M., Comyn-Wattiau,

I., and Métais, E., editors, *Conceptual Modeling - ER 99, 18th International Conference on Conceptual Modeling, Paris, France, November, 15-18, 1999, Proceedings*, volume 1728 of *Lecture Notes in Computer Science*, pages 16–30. Springer. (cited on pp 91, 94)

[Benz, 2007] Benz, S. (2007). Combining test case generation for component and integration testing. In *Proceedings of the 3rd International Workshop on Advances in Model-based Testing (A-MOST)*, pages 23–33, New York, NY, USA. ACM. (cited on p 74)

[Bézivin and Heckel, 2006] Bézivin, J. and Heckel, R. (2006). Guest editorial to the special issue on language engineering for model-driven software development. *Software and Systems Modeling*, 5(3):231–232. (cited on p 15)

[Bosch, 1998] Bosch, J. (1998). Design patterns as language constructs. *JOOP - Journal of Object-Oriented Programming*, 11(2):18–32. (cited on p 253)

[Bouneffa and Boudjlida, 1995] Bouneffa, M. and Boudjlida, N. (1995). Managing schema changes in object-relationship databases. In Papazoglou, M. P., editor, *OOER 95: Object-Oriented and Entity-Relationship Modelling, 14th International Conference, Gold Coast, Australia, December 12-15, 1995, Proceedings*, volume 1021 of *Lecture Notes in Computer Science*, pages 113–122. Springer. (cited on p 93)

[Braun, 2003] Braun, P. (2003). Metamodel-based integration of tools. In *Proceeding of ESEC/FSE 2003, TIS 2003 Workshop on Tool Integration in System Development*. Citeseer. (cited on pp 29, 63)

[Braun, 2004] Braun, P. (2004). *Metamodellbasierte Kopplung von Werkzeugen in der Softwareentwicklung*. PhD thesis, Technische Universität München. (cited on pp 29, 63)

[Braun and Marschall, 2003] Braun, P. and Marschall, F. (2003). Transforming object oriented models with BOTL. *Electronic Notes in Theoretical Computer Science*, 72(3):103 – 117. GT-VMT'2002, Graph Transformation and Visual Modeling Techniques (First International Conference on Graph Transformation). (cited on p 62)

[Brèche, 1996] Brèche, P. (1996). Advanced primitives for changing schemas of object databases. In *CAiSE '96: Proceedings of the 8th International Conference on Advances Information System Engineering*, pages 476–495, London, UK. Springer-Verlag. (cited on pp 92, 116, 119, 120, 122, 123, 125, 126, 129)

[Brèche et al., 1995] Brèche, P., Ferrandina, F., and Kuklok, M. (1995). Simulation of schema change using views. In *Database and Expert Systems Applications*, volume 978, pages 247–258. Springer Berlin / Heidelberg. (cited on p 94)

[Broy et al., 2010] Broy, M., Feilkas, M., Herrmannsdoerfer, M., Merenda, S., and Ratiu, D. (2010). Seamless model-based development: From isolated tools to integrated model engineering environments. *Proceedings of the IEEE*, 98(4):526 – 545. (cited on p 44)

[Broy and Stølen, 2001] Broy, M. and Stølen, K. (2001). *Specification and development of interactive systems: focus on streams, interfaces, and refinement*. Springer-Verlag New York, Inc., Secaucus, NJ, USA. (cited on p 50)

[Bruegge et al., 2008] Bruegge, B., Creighton, O., Helming, J., and Koegel, M. (2008). Unicase – an ecosystem for unified software engineering research tools. In *ICGSE '08: Distributed software development: methods and tools for risk management ; ICGSE Workshop 2008*, volume Online. (cited on p 197)

[Buchwald and Jakumeit, 2010] Buchwald, S. and Jakumeit, E. (2010). A GrGen.NET solution of the model migration case for the Transformation Tool Contest 2010. In *Transformation Tool Contest (TTC 2010)*. (cited on p 208)

[Burger and Gruschko, 2010] Burger, E. and Gruschko, B. (2010). A change metamodel for the evolution of MOF-based metamodels. In *Modellierung 2010*, volume P-161 of *GI-LNI*. (cited on p 116)

[Bézivin, 2005] Bézivin, J. (2005). On the unification power of models. *Software and Systems Modeling*, 4(2):171–188. (cited on p 30)

[Casais, 1995] Casais, E. (1995). Managing class evolution in object-oriented systems. pages 201–244. (cited on pp 91, 94)

[Chen et al., 2005] Chen, K., Sztipanovits, J., and Neema, S. (2005). Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 35–43, New York, NY, USA. ACM. (cited on pp 29, 45, 49)

[Cicchetti et al., 2010] Cicchetti, A., Meyers, B., and Wimmer, M. (2010). Abstract and concrete syntax migration of instance models. In *Transformation Tool Contest (TTC 2010)*. (cited on p 208)

[Cicchetti et al., 2008] Cicchetti, A., Ruscio, D. D., Eramo, R., and Pierantonio, A. (2008). Automating co-evolution in model-driven engineering. In Ceballos, S., editor, *Enterprise Distributed Object Computing Conference, 2008. EDOC'08. 12th International IEEE*. IEEE Computer Society. (cited on pp 57, 61, 66, 78, 99, 116, 119, 120, 122, 123, 125, 126, 129, 215, 263)

[Cicchetti et al., 2009] Cicchetti, A., Ruscio, D. D., and Pierantonio, A. (2009). Managing dependent changes in coupled evolution. In *ICMT2009 - International Conference on Model Transformation*. Springer LNCS. (cited on p 99)

[Clamen, 1994] Clamen, S. M. (1994). Schema evolution and integration. *Distributed and Parallel Databases*, 2(1):101–126. (cited on p 93)

[Claypool et al., 1998] Claypool, K. T., Jin, J., and Rundensteiner, E. A. (1998). SERF: schema evolution through an extensible, re-usable and flexible framework. In *CIKM '98: Proceedings of the seventh international conference on Information and knowledge management*, pages 314–321, New York, NY, USA. ACM. (cited on p 92)

[Claypool et al., 2000] Claypool, K. T., Rundensteiner, E. A., and Heineman, G. T. (2000). ROVER: A framework for the evolution of relationships. In *Conceptual Modeling - ER 2000*, volume 1920 of *Lecture Notes in Computer Science*, pages 893–917. Springer Berlin / Heidelberg. (cited on pp 92, 116, 119, 120, 122, 123, 125, 126, 129)

[Codd, 1970] Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387. (cited on p 89)

[Cook et al., 2007] Cook, S., Jones, G., Kent, S., and Wills, A. (2007). *Domain-specific development with Visual Studio DSL Tools*. Addison-Wesley Professional. (cited on pp 16, 30)

[Crestana-Jensen et al., 2000] Crestana-Jensen, V., Lee, A., and Rundensteiner, E. (2000). Consistent schema version removal: an optimization technique for object-oriented views. *Knowledge and Data Engineering, IEEE Transactions on*, 12(2):261 –280. (cited on p 94)

[Curino et al., 2008a] Curino, C., Moon, H. J., Tanca, L., and Zaniolo, C. (2008a). Schema evolution in Wikipedia - toward a web information system benchmark. In *ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems, Volume DISI, Barcelona, Spain, June 12-16, 2008*, pages 323–332. (cited on pp 66, 90, 197)

[Curino et al., 2008b] Curino, C., Moon, H. J., and Zaniolo, C. (2008b). Graceful database schema evolution: the PRISM workbench. *PVLDB*, 1(1):761–772. (cited on p 90)

[Curino et al., 2009] Curino, C. A., Moon, H. J., Ham, M., and Zaniolo, C. (2009). The PRISM workwench: Database schema evolution without tears. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1523–1526, Washington, DC, USA. IEEE Computer Society. (cited on p 90)

[Czarnecki and Eisenecker, 2000] Czarnecki, K. and Eisenecker, U. W. (2000). *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co.,

New York, NY, USA. (cited on p 15)

[Czarnecki and Helsen, 2006] Czarnecki, K. and Helsen, S. (2006). Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645. (cited on pp 63, 64, 219)

[Dassault Systèmes, 2010] Dassault Systèmes (2010). Catia. `http://www.catia.com`. (cited on p 19)

[Deissenboeck et al., 2007] Deissenboeck, F., Wagner, S., Pizka, M., Teuchert, S., and Girard, J. (2007). An activity-based quality model for maintainability. In *IEEE International Conference on Software Maintenance, 2007. ICSM 2007*, pages 184–193. (cited on pp 53, 190)

[Dig and Johnson, 2006] Dig, D. and Johnson, R. (2006). How do APIs evolve? a story of refactoring. *J. Softw. Maint. Evol.*, 18(2):83–107. (cited on pp 18, 66, 67, 116, 119, 120, 122, 123, 125, 126, 129)

[Ehrig et al., 2008] Ehrig, K., Küster, J. M., and Taentzer, G. (2008). Generating instance models from meta models. *Software and Systems Modeling*. (cited on p 32)

[Eichler et al., 2006] Eichler, H., Scheidgen, M., and Soden, M. (2006). A meta-modelling framework for modelling semantics in the context of existing domains platforms. In Hajo Eichler, T. R., editor, *ECMDA Workshop on Integration of Model Driven Development and Model Driven Testing*. Fraunhofer FOKUS, Berlin. (cited on p 52)

[Favre, 2003] Favre, J.-M. (2003). Meta-model and model co-evolution within the 3D software space. In *Proceedings of the ELISA workshop Evolution of Large-scale Industrial Software Evolution*, pages 98–109. (cited on pp 54, 66)

[Favre, 2005] Favre, J.-M. (2005). Languages evolve too! changing the software time scale. In *Principles of Software Evolution, Eighth International Workshop on*, pages 33–42. (cited on pp 17, 53, 81, 83, 187, 188)

[Ferrandina et al., 1995] Ferrandina, F., Meyer, T., Zicari, R., Ferran, G., and Madec, J. (1995). Schema and database evolution in the O2 object database system. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 170–181, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. (cited on p 92)

[Flouris et al., 2008] Flouris, G., Manakanatas, D., Kondylakis, H., Plexousakis, D., and Antoniou, G. (2008). Ontology change: Classification and survey. *Knowl. Eng. Rev.*, 23(2):117–152. (cited on p 85)

[Fowler, 1999] Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (cited on pp 72, 116, 119, 120, 122, 123, 125, 126, 129, 187)

[Fowler, 2005] Fowler, M. (2005). Language workbenches: The killer-app for domain specific languages? `http://martinfowler.com/articles/languageWorkbench.html`. (cited on pp 16, 43)

[France and Rumpe, 2007] France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 37–54, Washington, DC, USA. IEEE Computer Society. (cited on pp 15, 16, 17, 28)

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley. (cited on p 216)

[Garcés et al., 2009] Garcés, K., Jouault, F., Cointe, P., and Bézivin, J. (2009). A domain specific language for expressing model matching. In *Proceedings of the 5ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM09)*. (cited on pp 66, 218, 220)

[Garcés et al., 2009] Garcés, K., Jouault, F., Cointe, P., and Bézivin, J. (2009). Managing model adaptation by precise detection of metamodel changes. In *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 34–49.

Springer Berlin / Heidelberg. (cited on pp 67, 78, 99, 215, 218, 220, 263)

[Garlan et al., 1994] Garlan, D., Krueger, C. W., and Lerner, B. S. (1994). TransformGen: automating the maintenance of structure-oriented environments. *ACM Trans. Program. Lang. Syst.*, 16(3):727–774. (cited on pp 67, 95)

[Geest et al., 2008] Geest, G. d., Vermolen, S., Deursen, A. v., and Visser, E. (2008). Generating version convertors for domain-specific languages. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 197–201, Washington, DC, USA. IEEE Computer Society. (cited on pp 67, 99)

[Gil and Maman, 2005] Gil, J. and Maman, I. (2005). Micro patterns in Java code. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005*, pages 97–116. (cited on p 253)

[Goldschmidt et al., 2008] Goldschmidt, T., Becker, S., and Uhl, A. (2008). Classification of concrete textual syntax mapping approaches. In *ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture*, pages 169–184, Berlin, Heidelberg. Springer-Verlag. (cited on p 45)

[Greenfield et al., 2004] Greenfield, J., Short, K., Cook, S., and Kent, S. (2004). *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. John Wiley & Sons. (cited on pp 16, 29)

[Gronback, 2009] Gronback, R. C. (2009). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional. (cited on pp 216, 245)

[Grønmo et al., 2009] Grønmo, R., Møller-Pedersen, B., and Olsen, G. (2009). Comparison of three model transformation languages. In *Model Driven Architecture - Foundations and Applications*, volume 5562 of *LNCS*, pages 2–17. Springer. (cited on p 219)

[Gruschko, 2006] Gruschko, B. (2006). Towards structured revisions of meta models and semi-automatic model migration. In *Eclipse Modeling Symposium at Eclipse Summit Europe 2006*. (cited on p 98)

[Gruschko et al., 2007] Gruschko, B., Kolovos, D. S., and Paige, R. F. (2007). Towards synchronizing models with evolving metamodels. In *Workshop on Model-Driven Software Evolution at CSMR 2007*. (cited on pp 57, 66, 78, 98, 223)

[Guerrini and Mesiti, 2008] Guerrini, G. and Mesiti, M. (2008). X-Evolution: A comprehensive approach for XML schema evolution. In *Database and Expert Systems Application, 2008. DEXA '08. 19th International Conference on*, pages 251–255. (cited on p 97)

[Guerrini et al., 2007] Guerrini, G., Mesiti, M., and Sorrenti, M. A. (2007). XML schema evolution: Incremental validation and efficient document adaptation. In *Database and XML Technologies*, volume 4704 of *Lecture Notes in Computer Science*, pages 92–106. Springer Berlin / Heidelberg. (cited on p 97)

[Guizzardi, 2005] Guizzardi, G. (2005). *Ontological foundations for structural conceptual models*. PhD thesis, University of Twente, Enschede, The Netherlands. (cited on p 15)

[Hage and van Keeken, 2009] Hage, J. and van Keeken, P. (2009). Neon: A library for language usage analysis. In *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 35–53. Springer Berlin / Heidelberg. (cited on p 254)

[Harel and Rumpe, 2004] Harel, D. and Rumpe, B. (2004). Meaningful modeling: what's the semantics of "semantics"? *Computer*, 37(10):64–72. (cited on pp 29, 49)

[Henderson-Sellers and Gonzalez-Perez, 2006] Henderson-Sellers, B. and Gonzalez-Perez, C. (2006). Uses and abuses of the stereotype mechanism in UML 1.x and 2.0. In *MoDELS '06*, volume 4199 of *LNCS*, pages 16–26. Springer. (cited on p 253)

[Herrmannsdoerfer, 2009] Herrmannsdoerfer, M. (2009). Operation-based versioning of

metamodels with COPE. In *CVSM '09: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 49–54, Washington, DC, USA. IEEE Computer Society. (cited on pp 22, 149, 267)

[Herrmannsdoerfer, 2010] Herrmannsdoerfer, M. (2010). Migrating UML activity models with COPE. In *Transformation Tool Contest (TTC 2010)*. (cited on pp 23, 167, 208, 268)

[Herrmannsdoerfer, 2011] Herrmannsdoerfer, M. (2011). COPE - a workbench for the coupled evolution of metamodels and models. In Malloy, B., Staab, S., and van den Brand, M., editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 286–295. Springer Berlin / Heidelberg. (cited on pp 22, 149, 267)

[Herrmannsdoerfer et al., 2008a] Herrmannsdoerfer, M., Benz, S., and Juergens, E. (2008a). Automatability of coupled evolution of metamodels and models in practice. In Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., and Völter, M., editors, *Model Driven Engineering Languages and Systems (MODELS 2008)*, volume 5301/2008 of *Lecture Notes in Computer Science*, pages 645–659. Springer Berlin / Heidelberg. (cited on pp 21, 65, 265)

[Herrmannsdoerfer et al., 2008b] Herrmannsdoerfer, M., Benz, S., and Juergens, E. (2008b). COPE: A language for the coupled evolution of metamodels and models. In *1st International Workshop on Model Co-Evolution and Consistency Management*. (cited on pp 22, 105, 266)

[Herrmannsdoerfer et al., 2009a] Herrmannsdoerfer, M., Benz, S., and Juergens, E. (2009a). COPE - automating coupled evolution of metamodels and models. In *ECOOP 2009 - Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 52–76. Springer Berlin / Heidelberg. (cited on pp 22, 105, 167, 266, 267)

[Herrmannsdoerfer et al., 2009b] Herrmannsdoerfer, M., Haberl, W., and Baumgarten, U. (2009b). Model-level simulation for COLA. In *MISE '09: Proceedings of the 2009 ICSE Workshop on Modeling in Software Engineering*, pages 38–43, Washington, DC, USA. IEEE Computer Society. (cited on p 52)

[Herrmannsdoerfer and Koegel, 2010a] Herrmannsdoerfer, M. and Koegel, M. (2010a). Towards a generic operation recorder for model evolution. In *IWMCP '10: Proceedings of the 1st International Workshop on Model Comparison in Practice*, pages 76–81, New York, NY, USA. ACM. (cited on pp 149, 199)

[Herrmannsdoerfer and Koegel, 2010b] Herrmannsdoerfer, M. and Koegel, M. (2010b). Towards semantics-preserving model migration. In *Proceedings of the International Workshop on Models and Evolution*. (cited on pp 23, 231, 268)

[Herrmannsdoerfer et al., 2008c] Herrmannsdoerfer, M., Konrad, S., and Berenbach, B. (2008c). Tabular notations for state machine-based specifications. *Cross Talk, The Journal of defense Software Engineering*. (cited on p 46)

[Herrmannsdoerfer and Ratiu, 2009] Herrmannsdoerfer, M. and Ratiu, D. (2009). Limitations of automating model migration in response to metamodel adaptation. In *Proc. of the Joint ModSE-MCCM Workshop on Models and Evolution*. (cited on pp 22, 105, 267)

[Herrmannsdoerfer and Ratiu, 2010] Herrmannsdoerfer, M. and Ratiu, D. (2010). Limitations of automating model migration in response to metamodel adaptation. In *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 205–219. Springer Berlin / Heidelberg. (cited on pp 22, 105, 267)

[Herrmannsdoerfer et al., 2010a] Herrmannsdoerfer, M., Ratiu, D., Koegel, M., Herrmannsdoerfer, M., Ratiu, D., and Koegel, M. (2010a). Metamodel usage analysis for identifying metamodel improvements. In Malloy, B., Staab, S., and van den Brand, M., editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 62–81. Springer Berlin / Heidelberg. (cited on pp 23, 231, 268)

[Herrmannsdoerfer et al., 2009c] Herrmannsdoerfer, M., Ratiu, D., and Wachsmuth, G. (2009c). Language evolution in practice: The history of GMF. In *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 3–22. Springer Berlin / Heidelberg. (cited on pp 22, 167, 215, 267)

[Herrmannsdoerfer et al., 2010b] Herrmannsdoerfer, M., Vermolen, S., Wachsmuth, G., Herrmannsdoerfer, M., Vermolen, S., and Wachsmuth, G. (2010b). An extensive catalog of operators for the coupled evolution of metamodels and models. In Malloy, B., Staab, S., and van den Brand, M., editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 163–182. Springer Berlin / Heidelberg. (cited on pp 22, 105, 267)

[Hildisch et al., 2007] Hildisch, A., Steurer, J., and Stolle, R. (2007). HMI generation for plug-in services from semantic descriptions. In *Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems (SEAS)*, Washington, DC, USA. IEEE Computer Society. (cited on p 73)

[Horn, 2010] Horn, T. (2010). Model migration with GReTL. In *Transformation Tool Contest (TTC 2010)*. (cited on p 208)

[Hößler et al., 2005] Hößler, J., Soden, M., and Eichler, H. (2005). *Models and Human Reasoning*, chapter Coevolution of Models, Metamodels and Transformations, pages 129–154. Wissenschaft und Technik Verlag, Berlin. (cited on p 99)

[Huber et al., 1996] Huber, F., Schätz, B., Schmidt, A., and Spies, K. (1996). AutoFocus: A tool for distributed systems specification. In *FTRTFT '96: Proceedings of the 4th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 467–470, London, UK. Springer-Verlag. (cited on p 53)

[Hussey and Paternostro, 2006] Hussey, K. and Paternostro, M. (2006). Advanced features of EMF. Tutorial at EclipseCon 2006, California, USA. [Accessed 07 September 2009] Available at: `http://www.eclipsecon.org/2006/Sub.do?id=171`. (cited on p 218)

[Jouault and Bézivin, 2006] Jouault, F. and Bézivin, J. (2006). KM3: A DSL for metamodel specification. In Gorrieri, R. and Wehrheim, H., editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185. Springer Berlin / Heidelberg. (cited on pp 30, 31)

[Jouault and Kurtev, 2006] Jouault, F. and Kurtev, I. (2006). Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844/2006 of *Lecture Notes in Computer Science*, pages 128–138. Springer Berlin / Heidelberg. (cited on pp 63, 220)

[Juergens and Pizka, 2006] Juergens, E. and Pizka, M. (2006). The language evolver Lever – tool demonstration –. *Electronic Notes in Theoretical Computer Science*, 164(2):55 – 60. Proceedings of the Sixth Workshop on Language Descriptions, Tools, and Applications (LDTA 2006), Sixth Workshop on Language Descriptions, Tools, and Applications. (cited on p 96)

[Kalnina et al., 2010] Kalnina, E., Kalnins, A., Iraids, J., Sostaks, A., and Celms, E. (2010). Model migration with MOLA. In *Transformation Tool Contest (TTC 2010)*. (cited on p 208)

[Kang et al., 1990] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute. (cited on p 86)

[Karsai et al., 2009] Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schindler, M., and Völkl, S. (2009). Design guidelines for domain specific languages. In *The 9th OOPSLA Workshop on Domain-Specific Modeling*. (cited on p 236)

[Kelly and Pohjonen, 2009] Kelly, S. and Pohjonen, R. (2009). Worst practices for domain-specific modeling. *IEEE Software*, 26(4):22–29. (cited on p 236)

[Kelly and Tolvanen, 2007] Kelly, S. and Tolvanen, J.-P. (2007). *Domain-Specific Modeling*. John Wiley & Sons. (cited on pp 15, 16, 27, 28, 30)

[Kieburtz et al., 1996] Kieburtz, R. B., McKinney, L., Bell, J. M., Hook, J., Kotov, A., Lewis, J., Oliva, D. P., Sheard, T., Smith, I., and Walton, L. (1996). A software engineering experiment in software component generation. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 542–552, Washington, DC, USA. IEEE Computer Society. (cited on p 27)

[Kim, 1990] Kim, W. (1990). *Introduction to object-oriented databases*. MIT Press, Cambridge, MA, USA. (cited on p 89)

[Kim and Chou, 1988] Kim, W. and Chou, H.-T. (1988). Versions of schema for object-oriented databases. In *VLDB '88: Proceedings of the 14th International Conference on Very Large Data Bases*, pages 148–159, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. (cited on p 93)

[Kitchenham and Charters, 2007] Kitchenham, B. and Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report. (cited on p 84)

[Kläs et al., 2010] Kläs, M., Lampasona, C., Nunnenmacher, S., Wagner, S., Herrmannsdoerfer, M., and Lochmann, K. (2010). How to evaluate meta-models for software quality? In *DASMA Metrik Kongress*. (cited on p 190)

[Kleppe, 2008] Kleppe, A. (2008). *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional. (cited on pp 16, 17, 29, 37)

[Kleppe and Rensink, 2008] Kleppe, A. G. and Rensink, A. (2008). A graph-based semantics for UML class and object diagrams. Technical Report TR-CTIT-08-06, Centre for Telematics and Information Technology, University of Twente, Enschede, Netherlands. (cited on pp 31, 32, 33, 34, 35, 37, 38, 40)

[Kleppe et al., 2003] Kleppe, A. G., Warmer, J., and Bast, W. (2003). *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (cited on pp 16, 26, 28, 176)

[Klint et al., 2005] Klint, P., Lämmel, R., and Verhoef, C. (2005). Toward an engineering discipline for grammarware. *ACM Trans. Softw. Eng. Methodol.*, 14(3):331–380. (cited on pp 18, 66, 114)

[Koch et al., 2010] Koch, A., Jubeh, R., and Zündorf, A. (2010). UML 1.4 to 2.1 activity diagram model migration with Fujaba - a case study. In *Transformation Tool Contest (TTC 2010)*. (cited on p 208)

[Koegel, 2008] Koegel, M. (2008). Towards software configuration management for unified models. In *CVSM '08: Proceedings of the 2008 international workshop on Comparison and versioning of software models*, pages 19–24, New York, NY, USA. ACM. (cited on p 198)

[Koegel et al., 2009a] Koegel, M., Helming, J., and Seyboth, S. (2009a). Operation-based conflict detection and resolution. In *CVSM '09: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, pages 43–48, Washington, DC, USA. IEEE Computer Society. (cited on p 198)

[Koegel et al., 2009b] Koegel, M., Herrmannsdoerfer, M., Helming, J., and Li, Y. (2009b). State-based vs. operation-based change tracking. In *Models and Evolution - Joint MoDSE-MCCM Workshop*, pages 132–141. online. (cited on pp 164, 198)

[Koegel et al., 2010a] Koegel, M., Herrmannsdoerfer, M., Li, Y., Helming, J., and David, J. (2010a). Comparing state- and operation-based change tracking on models. In *Proceedings of the IEEE International EDOC Conference*. (cited on p 198)

[Koegel et al., 2010b] Koegel, M., Herrmannsdoerfer, M., von Wesendonk, O., and Helming, J. (2010b). Operation-based conflict detection. In *IWMCP '10: Proceedings of the 1st International Workshop on Model Comparison in Practice*, pages 21–30, New York, NY, USA. ACM.

(cited on p 166)

[Koenig et al., 2007] Koenig, D., Glover, A., King, P., Laforge, G., and Skeet, J. (2007). *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA. (cited on pp 110, 136, 255)

[Kolovos, 2009] Kolovos, D. (2009). *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, University of York, United Kingdom. (cited on p 218)

[Krüger et al., 2006] Krüger, I. H., Mathew, R., and Meisinger, M. (2006). Efficient exploration of service-oriented architectures using aspects. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 62–71, New York, NY, USA. ACM. (cited on p 53)

[Kugele et al., 2007] Kugele, S., Tautschnig, M., Bauer, A., Schallhart, C., Merenda, S., Haberl, W., Kühnel, C., Müller, F., Wang, Z., Wild, D., Rittmann, S., and Wechs, M. (2007). COLA – the component language. Technical Report TUM-I0714, Institut für Informatik, Technische Universität München. (cited on p 53)

[Kurtev et al., 2002] Kurtev, I., Bézivin, J., and Aksit, M. (2002). Technological spaces: An initial appraisal. In *CoopIS, DOA Federated Conferences, Industrial track*. (cited on pp 66, 81, 82)

[Lämmel, 2001] Lämmel, R. (2001). Grammar adaptation. In *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021/2001 of *Lecture Notes in Computer Science*, pages 550–570. Springer Berlin / Heidelberg. (cited on pp 96, 99, 179)

[Lämmel, 2004] Lämmel, R. (2004). Coupled software transformations (extended abstract). In *First International Workshop on Software Evolution Transformations*. (cited on p 83)

[Lämmel et al., 2005] Lämmel, R., Kitsis, S., and Remy, D. (2005). Analysis of XML schema usage. In *Conference Proceedings XML 2005*. (cited on p 253)

[Lämmel and Lohmann, 2001] Lämmel, R. and Lohmann, W. (2001). Format Evolution. In *Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001)*, volume 155 of *books@ocg.at*, pages 113–134. OCG. (cited on p 97)

[Lämmel and Pek, 2010] Lämmel, R. and Pek, E. (2010). Vivisection of a non-executable, domain-specific language - understanding (the usage of) the P3P language. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ICPC '10, pages 104–113, Washington, DC, USA. IEEE Computer Society. (cited on p 253)

[Lämmel and Verhoef, 2001] Lämmel, R. and Verhoef, C. (2001). Semi-automatic grammar recovery. *Softw. Pract. Exper.*, 31(15):1395–1448. (cited on pp 67, 189)

[Lämmel and Zaytsev, 2009a] Lämmel, R. and Zaytsev, V. (2009a). An introduction to grammar convergence. In *IFM '09: Proceedings of the 7th International Conference on Integrated Formal Methods*, pages 246–260, Berlin, Heidelberg. Springer-Verlag. (cited on p 96)

[Lämmel and Zaytsev, 2009b] Lämmel, R. and Zaytsev, V. (2009b). Recovering grammar relationships for the Java language specification. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:178–186. (cited on pp 67, 96, 114, 189, 197)

[Lange et al., 2006] Lange, C. F. J., DuBois, B., Chaudron, M. R. V., and Demeyer, S. (2006). An experimental investigation of UML modeling conventions. In *Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 27–41. Springer Berlin / Heidelberg. (cited on p 253)

[Lano and Rahimi, 2010] Lano, K. and Rahimi, S. K. (2010). Model migration transformation specification in UML-RSDS. In *Transformation Tool Contest (TTC 2010)*. (cited on p 208)

[Lautemann, 1996] Lautemann, S.-E. (1996). An introduction to schema versioning in OODBMS. In *DEXA '96: Proceedings of the 7th International Workshop on Database and Expert Systems Applications*, pages 132–139, Washington, DC, USA. IEEE Computer Society.

(cited on p 93)

[Lautemann, 1997] Lautemann, S.-E. (1997). A propagation mechanism for populated schema versions. In *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering*, pages 67–78, Los Alamitos, CA, USA. IEEE Computer Society. (cited on p 93)

[Ledeczi et al., 2001] Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., and Volgyesi, P. (2001). The Generic Modeling Environment. In *WISP*. (cited on pp 30, 97)

[Lerner, 1997] Lerner, B. S. (1997). TESS: automated support for the evolution of persistent types. In *ASE '97: Proceedings of the 12th international conference on Automated software engineering (formerly: KBSE)*, page 172, Washington, DC, USA. IEEE Computer Society. (cited on p 92)

[Lerner, 2000] Lerner, B. S. (2000). A model for compound type changes encountered in schema evolution. *ACM Trans. Database Syst.*, 25(1):83–127. (cited on pp 66, 92)

[Lerner and Habermann, 1990] Lerner, B. S. and Habermann, A. N. (1990). Beyond schema evolution to database reorganization. *SIGPLAN Not.*, 25(10):67–76. (cited on p 92)

[Levendovszky et al., 2010] Levendovszky, T., Balasubramanian, D., Narayanan, A., and Karsai, G. (2010). A novel approach to semi-automated evolution of DSML model transformation. In *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 23–41. Springer Berlin / Heidelberg. (cited on p 263)

[Li, 1999] Li, X. (1999). A survey of schema evolution in object-oriented databases. In *31st International Conference on Technology of Object-Oriented Language and Systems (TOOLS)*, page 362. IEEE Computer Society. (cited on p 95)

[Li et al., 2006] Li, X., Shannon, D., Walker, J., Khurshid, S., and Marinov, D. (2006). Analyzing the uses of a software modeling tool. *Electronic Notes in Theoretical Computer Science*, 164(2):3 – 18. Proceedings of the Sixth Workshop on Language Descriptions, Tools, and Applications (LDTA 2006), Sixth Workshop on Language Descriptions, Tools, and Applications. (cited on p 254)

[Lientz and Swanson, 1980] Lientz, B. P. and Swanson, E. B. (1980). *Software Maintenance Management*. Addison-Wesley. (cited on pp 54, 182, 184, 187, 188, 195)

[Liu et al., 1993] Liu, C.-T., Chrysanthis, P. K., and Chang, S.-K. (1993). Schema evolution through changes to ER diagrams. *J. Inf. Sci. Eng.*, 9(4):657–683. (cited on p 94)

[Liu et al., 1994] Liu, C.-T., Chrysanthis, P. K., and Chang, S.-K. (1994). Database schema evolution through the specification and maintenance of changes on entities and relationships. In Loucopoulos, P., editor, *Entity-Relationship Approach - ER 94, Business Modelling and Re-Engineering, 13th International Conference on the Entity-Relationship Approach, Manchester, U.K., December 13-16, 1994, Proceedings*, volume 881 of *Lecture Notes in Computer Science*, pages 132–151. Springer. (cited on p 94)

[Marschall, 2005] Marschall, F. (2005). *Modelltransformationen als Mittel der modellbasierten Entwicklung von Software-Systemen*. PhD thesis, Technische Universität München. (cited on pp 62, 115)

[Mealy, 1976] Mealy, G. (1976). A method for synthesizing sequential circuits. *Computer design development: principal papers*, 34:58. (cited on pp 56, 107)

[Mens and Demeyer, 2008] Mens, T. and Demeyer, S. (2008). *Software Evolution*. Springer Publishing Company, Incorporated. (cited on p 17)

[Mens and Tourwé, 2004] Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Trans. Softw. Eng.*, 30(2):126–139. (cited on p 232)

[Mens and Van Gorp, 2006] Mens, T. and Van Gorp, P. (2006). A taxonomy of model trans-

formation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142. (cited on pp 62, 63, 64, 109, 219)

[Mens et al., 2005] Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., and Jazayeri, M. (2005). Challenges in software evolution. In *8th International Workshop on Principles of Software Evolution (IWPSE)*, pages 13–22. (cited on p 17)

[Mesiti et al., 2006] Mesiti, M., Celle, R., Sorrenti, M. A., and Guerrini, G. (2006). X-evolution: A system for XML schema evolution and document adaptation. In *Advances in Database Technology - EDBT 2006*, volume 3896 of *Lecture Notes in Computer Science*, pages 1143–1146. Springer Berlin / Heidelberg. (cited on p 97)

[Meyer, 1996] Meyer, B. (1996). Schema evolution: Concepts, terminology, and solutions. *Computer*, 29(10):119–121. (cited on pp 54, 66)

[Meyer, 2000] Meyer, B. (2000). Principles of language design and evolution. In *Millennial Perspectives in Computer Science (Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Tony Hoare*, pages 229–246. Palgrave. (cited on p 18)

[Monk and Sommerville, 1993] Monk, S. and Sommerville, I. (1993). Schema evolution in OODBs using class versioning. *SIGMOD Rec.*, 22(3):16–22. (cited on p 93)

[Moore, 1956] Moore, E. (1956). Gedanken-experiments on sequential machines. *Automata studies*, 34:129–153. (cited on pp 27, 56, 106)

[Muller et al., 2005] Muller, P.-A., Fleurey, F., and Jézéquel, J.-M. (2005). Weaving executability into object-oriented meta-languages. In *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 264–278. Springer Berlin / Heidelberg. (cited on pp 52, 254)

[Narayanan et al., 2009] Narayanan, A., Levendovszky, T., Balasubramanian, D., and Karsai, G. (2009). Automatic domain model migration to manage metamodel evolution. In *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pages 706–711. Springer Berlin / Heidelberg. (cited on pp 66, 78, 98, 263)

[Nguyen and Rieu, 1989] Nguyen, G. T. and Rieu, D. (1989). Schema evolution in object-oriented database systems. *Data Knowl. Eng.*, 4(1):43–67. (cited on p 92)

[Object Management Group, 2001] Object Management Group (2001). Unified Modeling Language (UML) specification version 1.4. `http://www.omg.org/spec/UML/1.4/`. (cited on p 205)

[Object Management Group, 2003] Object Management Group (2003). Model Driven Architecture (MDA) guide version 1.0.1. `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf`. (cited on p 28)

[Object Management Group, 2004] Object Management Group (2004). Human-usable textual notation (HUTN) specification version 1.0. `http://www.omg.org/spec/HUTN/1.0/`. (cited on p 46)

[Object Management Group, 2006a] Object Management Group (2006a). Meta Object Facility (MOF) core specification version 2.0. `http://www.omg.org/spec/MOF/2.0/`. (cited on pp 29, 30, 97, 130)

[Object Management Group, 2006b] Object Management Group (2006b). Object Constraint Language (OCL) specification version 2.0. `http://www.omg.org/spec/OCL/2.0/`. (cited on pp 40, 41, 44, 51, 177)

[Object Management Group, 2007] Object Management Group (2007). XML Metadata Interchange (XMI) specification version 2.1.1. `http://www.omg.org/spec/XMI/2.1.1/`. (cited on pp 44, 46, 205)

[Object Management Group, 2008a] Object Management Group (2008a). MOF model to text transformation language version 1.0. `http://www.omg.org/spec/MOFM2T/1.0/`.

(cited on pp 44, 51)

[Object Management Group, 2008b] Object Management Group (2008b). MOF Query/View/Transformation (QVT) specification version 1.0. `http://www.omg.org/spec/QVT/1.0/`. (cited on pp 29, 44, 51, 63)

[Object Management Group, 2009] Object Management Group (2009). Unified Modeling Language (UML) superstructure version 2.2. `http://www.omg.org/spec/UML/2.2/`. (cited on pp 16, 17, 28, 30, 32, 41, 43, 53, 97, 106, 205)

[Overbey and Johnson, 2009] Overbey, J. L. and Johnson, R. E. (2009). Regrowing a language: refactoring tools allow programming languages to evolve. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 493–502, New York, NY, USA. ACM. (cited on pp 67, 96)

[Paige et al., 2000] Paige, R. F., Ostroff, J. S., and Brooke, P. J. (2000). Principles for modeling language design. *Information and Software Technology*, 42(10):665 – 675. (cited on p 236)

[Penney and Stein, 1987] Penney, D. J. and Stein, J. (1987). Class modification in the GemStone object-oriented DBMS. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 111–117, New York, NY, USA. ACM. (cited on p 92)

[Pizka and Juergens, 2007a] Pizka, M. and Juergens, E. (2007a). Automating language evolution. In *TASE '07: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 305–315, Washington, DC, USA. IEEE Computer Society. (cited on p 96)

[Pizka and Juergens, 2007b] Pizka, M. and Juergens, E. (2007b). Tool supported multi level language evolution. In *Software and Services Variability Management Workshop - Concepts, Models and Tools*, number 3 in Helsinki University of Technology Software Business and Engineering Institute Research Reports, pages 48–67. (cited on p 96)

[Pons, 1997] Pons, A., K. R. (1997). Schema evolution in object databases by catalogs. In *Database Engineering and Applications Symposium, 1997. IDEAS '97. Proceedings., International*, pages 368 –376. (cited on pp 95, 116, 119, 120, 122, 123, 125, 126, 129)

[Pretschner et al., 2007] Pretschner, A., Broy, M., Kruger, I. H., and Stauner, T. (2007). Software engineering for automotive systems: A roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 55–71, Washington, DC, USA. IEEE Computer Society. (cited on p 15)

[Ra and Rundensteiner, 1995a] Ra, Y.-G. and Rundensteiner, E. A. (1995a). Towards supporting hard schema changes in TSE. In *CIKM '95: Proceedings of the fourth international conference on Information and knowledge management*, pages 290–295, New York, NY, USA. ACM. (cited on p 94)

[Ra and Rundensteiner, 1995b] Ra, Y.-G. and Rundensteiner, E. A. (1995b). A transparent object-oriented schema change approach using view evolution. In *ICDE '95: Proceedings of the Eleventh International Conference on Data Engineering*, pages 165–172, Los Alamitos, CA, USA. IEEE Computer Society. (cited on p 94)

[Ra and Rundensteiner, 1997] Ra, Y.-G. and Rundensteiner, E. A. (1997). A transparent schema-evolution system based on object-oriented view technology. *IEEE Trans. on Knowl. and Data Eng.*, 9(4):600–624. (cited on p 94)

[Rahm and Bernstein, 2006] Rahm, E. and Bernstein, P. A. (2006). An online bibliography on schema evolution. *SIGMOD Rec.*, 35(4):30–31. (cited on pp 66, 89)

[Rashid and Sawyer, 2000] Rashid, A. and Sawyer, P. (2000). Object database evolution using separation of concerns. *SIGMOD Rec.*, 29(4):26–33. (cited on p 93)

[Rashid and Sawyer, 2005] Rashid, A. and Sawyer, P. (2005). A database evolution taxonomy

for object-oriented databases: Research articles. *J. Softw. Maint. Evol.*, 17(2):93–141. (cited on pp 91, 93)

[Robbes and Lanza, 2007] Robbes, R. and Lanza, M. (2007). A change-based approach to software evolution. *Electron. Notes Theor. Comput. Sci.*, 166:93–109. (cited on pp 77, 175)

[Roddick, 1992] Roddick, J. F. (1992). Schema evolution in database systems: an annotated bibliography. *SIGMOD Rec.*, 21(4):35–40. (cited on p 89)

[Roddick, 1995] Roddick, J. F. (1995). A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383 – 393. (cited on p 90)

[Ronström, 2000] Ronström, M. (2000). On-line schema update for a telecom database. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 329 –338. (cited on p 89)

[Rose et al., 2010a] Rose, L., Herrmannsdoerfer, M., Williams, J., Kolovos, D., Garcés, K., Paige, R., and Polack, F. (2010a). A comparison of model migration tools. In Petriu, D., Rouquette, N., and Haugen, O., editors, *Model Driven Engineering Languages and Systems*, volume 6394 of *Lecture Notes in Computer Science*, pages 61–75. Springer Berlin / Heidelberg. (cited on pp 23, 167, 268)

[Rose et al., 2010b] Rose, L., Kolovos, D., Paige, R., and Polack, F. (2010b). Migrating activity diagrams with Epsilon Flock. In *Transformation Tool Contest (TTC 2010)*. (cited on p 208)

[Rose et al., 2010c] Rose, L. M., Kolovos, D. S., Paige, R. F., and Polack, F. A. (2010c). Model migration case for TTC 2010. In *TTC 2010: Proc. Transformation Tool Contest Workshop*. (cited on pp 205, 207)

[Rose et al., 2010d] Rose, L. M., Kolovos, D. S., Paige, R. F., and Polack, F. A. C. (2010d). Model migration with Epsilon Flock. In *Theory and Practice of Model Transformations*, volume 6142 of *Lecture Notes in Computer Science*, pages 184–198. Springer Berlin / Heidelberg. (cited on pp 66, 78, 98, 215, 218, 263)

[Rose et al., 2009] Rose, L. M., Paige, R. F., Kolovos, D. S., and Polack, F. A. (2009). An analysis of approaches to model migration. In *Proc. Models and Evolution (MoDSE-MCCM) Workshop, 12th ACM/IEEE International Conference on Model Driven Engineering, Languages and Systems*. (cited on pp 66, 95, 96, 97, 100, 116, 262)

[Rumpe, 1998] Rumpe, B. (1998). A note on semantics (with an emphasis on UML). In Kilov, H. and Rumpe, B., editors, *Second ECOOP Workshop on Precise Behavioral Semantics*. Technische Universität München, TUM-I9813. (cited on pp 50, 141)

[Sadilek and Wachsmuth, 2008] Sadilek, D. A. and Wachsmuth, G. (2008). Prototyping visual interpreters and debuggers for domain-specific modelling languages. In *ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture*, pages 63–78, Berlin, Heidelberg. Springer-Verlag. (cited on p 52)

[Sadilek and Weißleder, 2008] Sadilek, D. A. and Weißleder, S. (2008). Testing metamodels. In *ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture*, pages 294–309, Berlin, Heidelberg. Springer-Verlag. (cited on p 54)

[Schätz, 2010] Schätz, B. (2010). UML model migration with PETE. In *Transformation Tool Contest (TTC 2010)*. (cited on p 208)

[Schwaber and Beedle, 2002] Schwaber, K. and Beedle, M. (2002). *Agile software development with Scrum*, volume 18. Prentice Hall Upper Saddle River, NJ. (cited on p 191)

[Selic, 2007] Selic, B. (2007). A systematic approach to domain-specific language design using UML. *Object-Oriented Real-Time Distributed Computing, IEEE International Symposium on*, 0:2–9. (cited on p 16)

[Sen et al., 2009] Sen, S., Moha, N., Baudry, B., and Jézéquel, J.-M. (2009). Meta-model prun-

ing. In *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 32–46, Berlin, Heidelberg. Springer-Verlag. (cited on p 253)

[Shneiderman and Thomas, 1982] Shneiderman, B. and Thomas, G. (1982). An architecture for automatic relational database system conversion. *ACM Trans. Database Syst.*, 7(2):235–257. (cited on p 90)

[Singer et al., 2009] Singer, J., Brown, G., Lujan, M., Pocock, A., and Yiapanis, P. (2009). Fundamental nano-patterns to characterize and classify Java methods. In *9th Workshop on Language Descriptions, Tools and Applications*, pages 204–218. (cited on p 253)

[Sjøberg, 1993] Sjøberg, D. (1993). Quantifying schema evolution. *Information and Software Technology*, 35(1):35 – 44. (cited on pp 66, 90)

[Skarra and Zdonik, 1986] Skarra, A. H. and Zdonik, S. B. (1986). The management of changing types in an object-oriented database. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 483–495, New York, NY, USA. ACM. (cited on p 93)

[Sockut and Goldberg, 1979] Sockut, G. H. and Goldberg, R. P. (1979). Database reorganization—principles and practice. *ACM Comput. Surv.*, 11(4):371–395. (cited on p 90)

[Spinellis, 2001] Spinellis, D. (2001). Notable design patterns for domain-specific languages. *J. Syst. Softw.*, 56(1):91–99. (cited on p 27)

[Sprinkle and Karsai, 2004] Sprinkle, J. and Karsai, G. (2004). A domain-specific visual language for domain model evolution. *Journal of Visual Languages and Computing*, 15(3-4):291–307. (cited on pp 18, 66, 78, 98, 262)

[Sprinkle, 2003] Sprinkle, J. M. (2003). *Metamodel driven model migration*. PhD thesis, Vanderbilt University, Nashville, TN, USA. Director-Karsai, Gabor. (cited on pp 17, 57, 60, 61, 64, 98)

[Stachowiak, 1973] Stachowiak, H. (1973). *Allgemeine Modelltheorie*. Springer-Verlag, Wien. (cited on p 26)

[Staudt et al., 1987] Staudt, B. J., Krueger, C. W., and Garlan, D. (1987). A structural approach to the maintenance of structure-oriented environments. In *SDE 2: Proceedings of the second ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 160–170, New York, NY, USA. ACM. (cited on p 95)

[Steinberg et al., 2009] Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional. (cited on pp 16, 97, 109, 131, 242)

[Steinke, 2006] Steinke, S. (2006). A380 cable problems threaten airbus. `http://www.flug-revue.rotor.com/FRHeft/FRHeft06/FRH0612/FR0612b.htm`. (cited on p 19)

[Street and Pettit, 2005] Street, J. A. and Pettit, R. G. (2005). The impact of UML 2.0 on existing UML 1.4 models. In *Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 431–444. Springer Berlin / Heidelberg. (cited on p 99)

[Su et al., 2001] Su, H., Kramer, D., Chen, L., Claypool, K. T., and Rundensteiner, E. A. (2001). XEM: Managing the evolution of XML documents. In *Eleventh International Workshop on Research Issues in Data Engineering on Document Management for Data Intensive Business and Scientific Applications*, pages 103–110, Washington, DC, USA. IEEE Computer Society. (cited on p 97)

[Taentzer et al., 2005] Taentzer, G., Ehrig, K., Guerra, E., Lara, J. D., Levendovszky, T., Prange, U., and Varro, D. (2005). Model transformations by graph transformations: A comparative study. In *Model Transformations in Practice Workshop at MoDELS 2005, Montego*, page 05. (cited on p 219)

[Tan and Goh, 2005] Tan, M. and Goh, A. (2005). Keeping pace with evolving XML-Based specifications. In *Current Trends in Database Technology - EDBT 2004 Workshops*, volume 3268 of *Lecture Notes in Computer Science*, pages 280–288. Springer Berlin / Heidelberg. (cited on p 97)

[Tisi et al., 2009] Tisi, M., Jouault, F., Fraternali, P., Ceri, S., and Bézivin, J. (2009). On the use of higher-order model transformations. In Paige, R., Hartman, A., and Rensink, A., editors, *Model Driven Architecture - Foundations and Applications*, volume 5562 of *Lecture Notes in Computer Science*, pages 18–33. Springer Berlin / Heidelberg. (cited on p 62)

[Tolvanen, 1998] Tolvanen, J.-P. (1998). *Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence*. PhD thesis, University of Jyväskylä. (cited on p 253)

[Tresch and Scholl, 1993] Tresch, M. and Scholl, M. H. (1993). Schema transformation without database reorganization. *SIGMOD Rec.*, 22(1):21–27. (cited on p 94)

[van Deursen et al., 2000] van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36. (cited on p 17)

[van Sterkenburg, 2003] van Sterkenburg, P. (2003). *A practical guide to lexicography*. John Benjamins Publishing Co. (cited on p 82)

[Ventrone and Heiler, 1991] Ventrone, V. and Heiler, S. (1991). Semantic heterogeneity as a result of domain evolution. *SIGMOD Rec.*, 20(4):16–20. (cited on p 90)

[Vermolen and Visser, 2008] Vermolen, S. D. and Visser, E. (2008). Heterogeneous coupled evolution of software languages. In Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., and Völter, M., editors, *Model Driven Engineering Languages and Systems (MODELS 2008)*, volume 5301 of *Lecture Notes in Computer Science*, pages 630–644. Springer. (cited on p 95)

[Wachsmuth, 2007] Wachsmuth, G. (2007). Metamodel adaptation and model co-adaptation. In *ECOOP 2007 - Object-Oriented Programming*, volume 4609/2007 of *Lecture Notes in Computer Science*, pages 600–624. Springer Berlin / Heidelberg. (cited on pp 66, 78, 98, 99, 114, 116, 119, 120, 122, 123, 125, 126, 129, 180, 215, 254, 256, 263)

[Walmsley, 2001] Walmsley, P. (2001). *Definitive XML Schema*. Prentice Hall PTR, Upper Saddle River, NJ, USA. (cited on p 96)

[Weiss and Lai, 1999] Weiss, D. M. and Lai, C. T. R. (1999). *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (cited on p 27)

[Winskel, 1993] Winskel, G. (1993). *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA. (cited on p 49)

[Zicari, 1991] Zicari, R. (1991). A framework for schema updates in an object-oriented database system. In *Data Engineering, 1991. Proceedings. Seventh International Conference on*, pages 2 –13. (cited on p 92)

[Zimmermann et al., 2005] Zimmermann, T., Weißgerber, P., Diehl, S., and Zeller, A. (2005). Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, pages 429–445. (cited on p 181)

# Index


abstract class, 38
adaptive maintenance, 54
attribute, 39

backwards-compatible metamodel change, 59
breaking metamodel change, 57

class, 30, 36
composite reference, 40
concrete syntax, 45
conformance, 37
  preservation
    metamodel, 109
    model, 109
constructor, 114
corrective maintenance, 54
coupled change, 67
  metamodel-independent, 71
  metamodel-specific, 70
  model-independent, 70
  model-specific, 69
coupled evolution, 83
coupled operation, 107
  constructor, 114
  custom, 110, 137
  destructor, 114
  inverse, 115
    safe, 115
    self-, 115
  model-preserving, 114
  refactoring, 114
  reusable, 112, 138
  safely model-migrating, 114
  semantics-preserving, 141, 257
  sequential composition, 107
  syntax-preserving, 140
custom coupled operation, 110, 137

data type, 38
dataware, 83, 89
  object-oriented, 91
  relational, 89
default value, 40
derived feature, 40
destructor, 114
directed multigraph, 32
Domain-Specific Modeling, 28

Eclipse Modeling Framework, 43
  Ecore, 131
Ecore, 131
empirical study, 168
  discussion, 168
  execution, 168
  goal, 168
  object, 168
  result, 168
  threats to validity, 168
enumeration, 39
extension, 83

feature, 39

generic instance model, 131
grammarware, 83, 95
graph constraint, 34
Graphical Modeling Framework, 47

higher-order model transformation, 62
history model, 158

identifier attribute, 40

intensional definition, 82

language history, 108
  metamodel, 158

maintenance
  adaptive, 54
  corrective, 54
  perfective, 54
  preventive, 54
meta hierarchy, 30
Meta Object Facility, 30
  abstract class, 38
  attribute, 39
  class, 30, 36
  Complete, 30
  composite reference, 40
  data type, 38
  default value, 40
  derived feature, 40
  enumeration, 39
  Essential, 30, 38
  feature, 39
  identifier attribute, 40
  multiplicity, 39
  ordered feature, 40
  package, 40
  primitive type, 38
  reference, 30, 36
  type, 39
metametamodel, 37
metamodel, 36
  adaptation, 107
  change, 56
    backwards-compatible, 59
    breaking, 57
    coupled, 67
    metamodel-only, 67
  greatest common, 55
metamodel-independent coupled change, 71
metamodel-only metamodel change, 67
metamodel-specific coupled change, 70
metamodeling language, 37
model, 32
  empty, 55
  migration, 107
    model-specific, 145
    semantics-preserving, 61
    syntax-preserving, 60
  sub-, 55
  transformation, 62
    definition, 62
    endogenous, 63
    exogenous, 63
    higher-order, 62
    in-place, 64
    language, 62
    out-of-place, 64
Model-Driven Architecture, 28
model-independent coupled change, 70
model-specific coupled change, 69
modeling language, 37
  abstract syntax, 29
  concrete syntax, 45
  for defining semantics, 254
  semantics, 49
modelware, 83, 97
multiplicity, 39

ordered feature, 40

package, 40
perfective maintenance, 54
preventive maintenance, 54
primitive type, 38

refactoring, 114
reference, 30, 36
reusable coupled operation, 112, 138
  composite
    delegation, 124
    inheritance, 123
    merge / split, 128
    replacement, 126
    specialization / generalization, 121
  primitive
    non-structural, 119
    structural, 118

semantics, 49
  adaptation, 255
  change, 57
  refactoring, 145
  refinement, 145

semantics-preserving
  coupled operation, 141, 257
  model migration, 61
Software Factories, 29
software language, 82
  evolution, 83
  extension, 83
  intensional definition, 82
  migration, 83
  utterance, 82
syntax-preserving
  coupled operation, 140
  model migration, 60

technical space, 86
  dataware, 83, 89
  grammarware, 83, 95
  modelware, 83, 97
  XMLware, 83, 96
type, 39
type graph, 32
  with constraints, 35

UML
  class diagram, 43
  object diagram, 41
utterance, 82

XMLware, 83, 96