Software Language Evolution

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen
op maandag 1 oktober 2012 om 15:00 uur door

Sander Daniël VERMOLEN

doctorandus informatica
geboren te Arnhem

Dit proefschrift is goedgekeurd door de promotor:

Prof. dr. A. van Deursen

Copromotor: Dr. E. Visser

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof. dr. A. van Deursen	Delft University of Technology, promotor
Dr. E. Visser	Delft University of Technology, copromotor
Prof. dr. R. Lämmel	University of Koblenz-Landau
Prof. dr. A. Rensink	University of Twente
Prof. dr. ir. A. P. de Vries	Centrum Wiskunde & Informatica Delft University of Technology
Prof. dr. C. Witteveen	Delft University of Technology
Dr. M. W. Godfrey	University of Waterloo

The work in this thesis has been carried out at the Delft University of Technology, under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The research was financially supported by the Netherlands Organisation for Scientific Research (NWO)/Jacquard project 638.001.610, *MoDSE: Model-Driven Software Evolution*.

ISBN 978-90-79982-13-4

Preface

If you would ask me to summarize this dissertation in one word, it would neither be software, nor language, nor evolution. It would be change. Change is a bit of a funny thing. It is often neglected, often ignored and generally opposed against in whatever way possible. But change is also the trigger for new thoughts and ideas, it is the driving factor of economic growth, it is the thing that makes tomorrow different from today. Some like it, some don't. But sooner or later it will happen. Change is inevitable.

Partially due to rapid development, partially due to the ease of adaptation, change is prominent in computer science. I spend four years researching change in computer science. Four years of my life that did not go as smoothly as most of you might know. Nevertheless, the research went well and I am proud of the result: the book you are holding in your hands.

By now, I changed my career path to industry. I even – more or less – changed my field of work to what some of my PhD colleagues would consider the dark side (physics). But fear not, change drives new ideas and insights and can most of all be highly enjoyable. And for those that do not like change, some things are still the same: I still work with models, they still change all the time and their change still rises the same issues as the ones addressed in the following chapters.

Acknowledgements

There has been much support from many people during my PhD. I thank all of them, but some I would like to thank in particular:

First of all, I thank my copromotor, Eelco Visser. His input and ideas have shaped this dissertation. I thank my promotor, Arie van Deursen, for his advice and many suggestions that greatly improved the chapters. I also thank Mike Godfrey, Ralf Lämmel, Arend Rensink, Arjen de Vries and Cees Witteveen for reviewing my dissertation.

I thank Markus Herrmannsdörfer and Guido Wachsmuth for an excellent and fruitful joint effort. I thank you for co-authoring several papers, but most of all for the pleasant collaboration.

I thank our SERG coffee club, with whom I drank many, many cups of heated liquid, including Sander van der Burg, Eelco Dolstra, Danny Groenewegen, Zef Hemel, Maartje de Jonge, Lennart Kats and Rob Vermaas. Our coffee or tea was always accompanied by a more or less research-related discussion, of which – I am sure – some have altered bits of this dissertation, and of which – I am glad – some did not.

I thank Scott Adams for providing a daily dose of humor in three pictures. I also thank Jarkko Oikarinen for providing a convenient means of communication. A technology, that can be used for good cause, yet – as many posted URLs from a single website with pictures have shown – can also easily be abused.

Finally and most importantly, I thank my parents and my sister for their unconditional support. Whatever happened, I could always count on you and hope I always can.

Sander Vermolen
August 11, 2012
Aalst

Contents

List of Acronyms

API Application Programming Interface

AST Abstract Syntax Tree

ATL Atlas Transformation Language

CMOF Complete MetaObject Facility

DSL Domain Specific Language

DSTL Domain Specific Transformation Language

DTD Document Type Definition

EMF Eclipse Modeling Framework

EMOF Essential MetaObject Facility

ETL Extract, Transform and Load

GMF Graphical Modeling Framework

GPL General Purpose Language

IDE Integrated Development Environment

JPA Java Persistence API

MOF MetaObject Facility

QVT Query/View/Transformation

SDF Syntax Definition Formalism

SQL Structured Query Language

TL Transformation Language

UML Unified Modeling Language

XML Extensible Markup Language

XSLT Extensible Stylesheet Language Transformations

Introduction

1

This dissertation discusses techniques, tools and theory on coupled evolution. Coupled evolution is the connection of software evolution patterns and adequate migrations of conforming artefacts, in order to retain artefact conformance. This dissertation covers various application domains of conformance and coupled evolution, in particular it addresses coupled evolution of metamodels and models and of (object-oriented) schemas and databases.

1.1 MODEL-DRIVEN ENGINEERING

Software development is hard. Programming languages ease software development by offering abstractions through an accessible language. Abstractions make software descriptions more concise, more readable and easier to understand, thus allowing software developers to write more complex software.

Some programming languages offer support for a broad range of software domains and are therefore generally referred to as *general-purpose programming languages* (GPLs). Examples are C, Java and Python. Due to their wide applicability, GPLs typically offer abstractions over the solution space – the computing platform – rather than abstractions over the problem space – the software domain. GPLs focus on exploiting the computing platform, rather than simplifying software development for a particular problem domain.

The solution-focused abstractions of GPLs enforce a solution-focused software description. The software developer is required to link the problem domain to the solution space in order to describe software. On the one hand, this requires a thorough understanding of the technical computation space. On the other hand, it enforces a computation-oriented (or technical) software description. There is a large semantic gap between the problem and the defined solution.

Model-driven engineering (MDE) aims to create problem-space abstractions through domain-specific models. Instead of writing program code in a general purpose language, software is modeled in a modeling language designed for one particular domain. For example, object role modeling (ORM) targets the domain of data structure definition, the hypertext markup language (HTML) targets the domain of web page layout, a scene description language (SDL) targets the domain of rendering 3-dimensional scenes, the structured query language (SQL) targets the domain of relational database querying. Models are close to the problem they solve and thereby easier to understand, validate and develop. Models can generally be interpreted, or transformed

into executable code automatically through compilation. The interpreter implementation, or the generated code are typically set in a GPL, making use of the solution-space abstractions offered.

Models are the primary software artefacts of model-driven engineering. The structure of information in a model is described in a metamodel. Metamodels come in many forms. When the models are textual, their metamodel is implicitly defined as part of their grammar. When models are graphical, the metamodel is generally explicitly defined. If models are modeled in terms of objects and object relations, metamodels describe object types and relation characteristics. The metamodel defines concepts such as object features, inheritance structure, relation cardinalities and inverse relations.

A model *conforms* to a metamodel when the model complies with the structure defined by the metamodel: All modeled objects must comply with the structure defined in non-abstract classes, all field values must be correctly typed, all references must comply with associations and all metamodel restrictions, such as cardinalities and inverses, must be satisfied. Although conformance can be formalized through a set of constraints [Paige et al., 2007], conformance restrictions are often implicit.

Being a model itself, a metamodel has a metamodel, generally referred to as meta-metamodel. It describes the structure of a metamodel. Meta-metamodels generally conform to themselves.

1.2 YELLOWGRASS – TWO EXAMPLE MODELS

Any software of reasonable size is bound to have bugs. Reporting and keeping track of these bugs is part of software development. Bug trackers ease the management of bugs. One such bug tracker is YellowGrass[1]. YellowGrass is a web application, which uses tags to manage software issues (such as bugs, new features and improvement suggestions). Tags are simple strings, which YellowGrass turns into a powerful organization tool. A more extensive description of YellowGrass can be found in Appendix A. This chapter (and later chapters) addresses YellowGrass as running example.

When operational, YellowGrass processes information, such as issues, user names, project descriptions and tags. It uses a database for persistent storage. The data in this database complies with the structure defined by YellowGrass's class diagram, a simplified version of which is shown in Figure 1.1. It describes issues, which are grouped into project and reported by users. Each project has several members (users), who can comment on issues and tag issues.

The class diagram of YellowGrass is a metamodel. It defines the structure of a model, namely the data stored in the database. Different YellowGrass

[1] http://yellowgrass.org

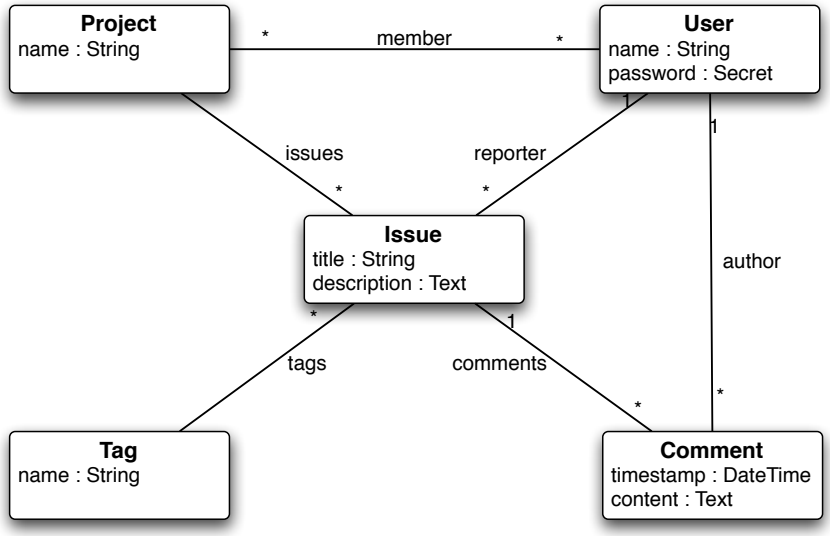


Figure 1.1 Simplified version of the YellowGrass class diagram.

instances can have different databases and thereby different models, yet provided they use the same YellowGrass version, they share the same metamodel. Conformance of the data is guarded by the database management system (and partially by the application as we will see later). A breach of conformance may cause data loss, as neither the application, nor the database is designed to deal with incorrectly structured data.

As the data in a database is a model because it conforms to the class diagram, the class diagram itself can also be considered a model (a data model), as it conforms to a data modeling language. The modeling language defines the structure of the class diagram, introducing concepts such as classes, class names, associations and association cardinalities. In Figure 1.1 the modeling language would be a variant on UML class diagrams. In YellowGrass's source code, the data model is defined textually in a language called WebDSL [Visser, 2008a]. WebDSL is a modeling language for defining web applications. A sub-language of WebDSL supports the definition of data models. Hence, in the context of YellowGrass we see different layers of conformance, namely: the data in a YellowGrass database conforms to the YellowGrass data model, whereas the YellowGrass data model conforms to the WebDSL data modeling language.

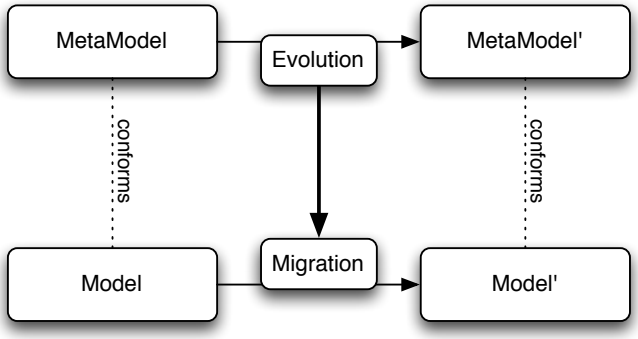


Figure 1.2 Coupled evolution overview

1.3 COUPLED EVOLUTION

Changing requirements, an increased knowledge of the domain and technological progress require software to *evolve* [Lehman and Belady, 1985]. Being an intrinsic part of software development, metamodels also evolve. Preventing metamodel evolution by backwards compatible changes is often insufficient as it reduces the quality of the metamodel [Casais, 1995].

As models conform to metamodels, metamodel evolution may break model conformance. Consequently, existing models may no longer be suitable as input to model transformations or code generation, they can sometimes no longer be edited or validated and their semantics is unclear. To prevent breaking conformance, metamodel evolution requires *model migration* [Sprinkle, 2003].

Model migration can be applied implicitly by manually editing a model upon metamodel evolution. However, manual editing is tedious and not feasible for larger models, or larger sets of models. Instead, model migration can be automated by explicitly specifying a migration. As writing migrations is generally far from trivial and error-prone, manual migration writing hampers the evolution process. To completely automate the evolution process, adequate model migrations need to be derived from the metamodel evolution, which is known as *coupled evolution* [Lämmel, 2004, Visser, 2008b].

Figure 1.2 shows coupled evolution in the context of conformance graphically. At the top, a metamodel evolves to a new version. At the bottom, a model conforming to the old metamodel is migrated to a new model conforming to the new metamodel. The dashed lines represent model conformance. The vertical arrow represents coupled evolution, in which migration is derived from evolution.

Also YellowGrass is subject to evolution. It has evolved from a simple issue tracker offering support for small projects, to a more feature-rich issue tracker, offering support for more extensive project management. YellowGrass's data

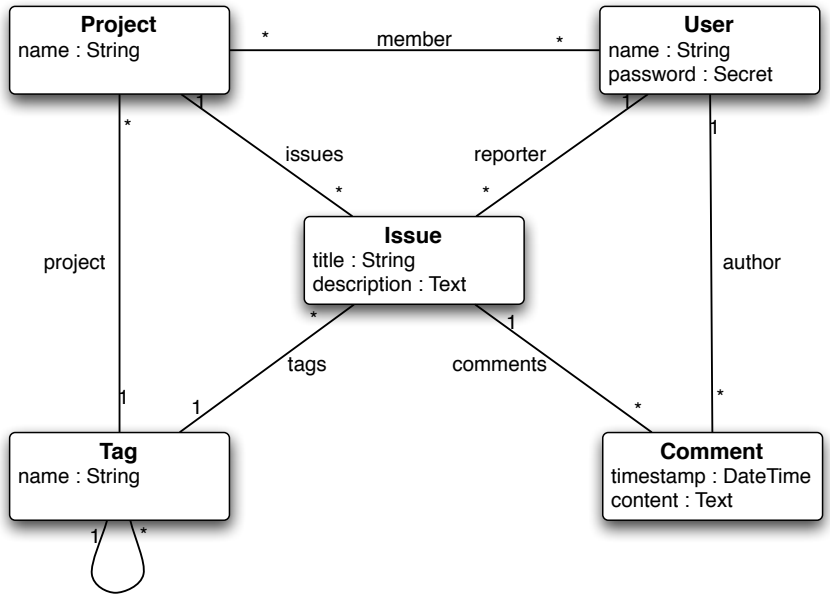


Figure 1.3 Improved version of the YellowGrass class diagram.

model evolved along with the application. Figure 1.3 shows a revised version of the data model from Figure 1.1.

The new version offers support for tagging tags. To this end, two associations are added: one between tags and projects to make tags project-specific and allow different projects to tag tags differently; and one between tags to register the tagging of tags. When we try to connect the improved application to an existing database, the new application will fail. The existing database neither stores references between tags and projects, nor references between tags. The existing data does not conform to the new data model. To prevent the loss of existing data, we need to migrate the database to conform to the revised data model by creating (and instantiating) the added associations. Figure 1.2 (left) outlines the coupled evolution process for YellowGrass's data model.

Additionally, WebDSL – YellowGrass's modeling language – evolved over time. The data modeling language was extended with additional constructs (such as to define default values) and adapted slightly to improve readability. Some of these changes created new WebDSL versions that were not backward compatible and thus needed existing applications, such as YellowGrass, to be changed. When a change is not backward compatible, it breaks the conformance relation. Coupled evolution reestablishes this relation by migration. Figure 1.4 (right) outlines coupled evolution for WebDSL. Note that coupled evolution for WebDSL may imply a need to change the YellowGrass data model. Yet, this change does not necessarily enforce a database migra-

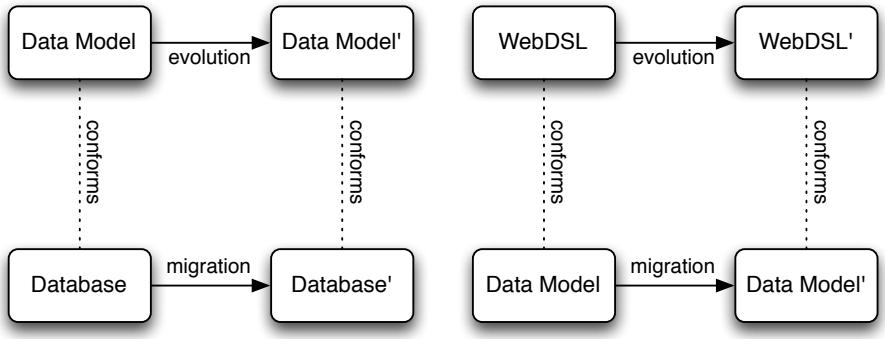


Figure 1.4 Two scenarios of coupled evolution. One in the context of the YellowGrass data model (left) and one in the context of the WebDSL data modeling sub language (right).

tion. Migrations in coupled evolution generally persist semantics, thereby not affecting other conformance relations.

To derive migrations automatically, the evolution – implicitly applied by the developer – needs to be made explicit. There are three common approaches to obtain an evolution specification: Firstly, evolution can be *specified manually* by the developer. This is likely to yield the correct evolution, yet requires additional development effort. Secondly, evolution can be *recorded*. This provides the correct evolution, but restricts development to a recording editor and the provided edit operations. Thirdly, evolution can be *detected* afterwards. This poses no restrictions on the development process, yet does not guarantee the correct evolution.

Evolution specifications can either be difference-based or operator-based. *Difference-based approaches* use a declarative evolution specification referred to as difference model [Cicchetti et al., 2008, Garcés et al., 2009]. Difference models captures differences, rather than how these differences were applied. *Operator-based approaches* model evolution by a sequence of operator applications [Wachsmuth, 2007b]. Each operator represents a change to the metamodel and can generally be coupled to a suitable model migration to form a coupled operator.

1.4 COUPLED EVOLUTION SPACES

Both evolution and conformance are common concepts throughout different technological spaces [Kurtev et al., 2002]. Most spaces commonly address coupled evolution to ease software maintenance. The terminology and requirements may differ across technological spaces, but the coupled evolution

principles are equivalent. Coupled evolution occurs most prominently in the spaces of dataware, grammarware, xmlware and modelware.

In *dataware*, the structure of data is modeled using data models (or schemas). When data models evolve, conforming data sets (or databases) need migration [Roddick, 1992]. For example, evolution of the YellowGrass data model needs migration of a YellowGrass database. Data sets are generally large, but frequently small in number. Evolution and migration applies to both object-oriented as well as relational schemas. Data migration is common, yet most often defined manually. The space of *xmlware* is similar to dataware with respect to the structure definitions, yet xml documents are generally smaller than the data sets faced in dataware, e.g., [Su et al., 2001] and [Guerrini and Mesiti, 2008].

The structure of sentences (or words) is captured in a grammar in the technological space of *grammarware*. Grammar evolution needs adaptation of sentences, e.g., [Staudt et al., 1987] and [Jürgens and Pizka, 2006]. Sentences are small compared to data sets, but generally larger in number. As the data modeling language of YellowGrass (WebDSL) is textual, evolution of WebDSL (of its grammar) and associated migration of YellowGrass is an example of coupled evolution in grammarware. Support for evolution in grammarware is limited, and in practice, most conformance-breaking evolutions are prevented by maintaining backward compatibility.

The space currently most active in terms of coupled evolution research is *modelware*. Models conform to metamodels. Metamodel evolution needs model migration, e.g., [Sprinkle, 2003], [Gruschko et al., 2007] and [Garcés et al., 2009]. As in grammarware, models are generally large in number, but relatively small and not in constant use.

The following chapters mostly target the modelware and dataware spaces. Yet, discussed principles can directly be ported to xmlware and are similar to the principles encountered in grammarware. A complete discussion of publications for each of the spaces can be found in Chapter 2.

1.5 PROBLEM STATEMENT

The combination of evolution and conformance implies a need for migration. Enabling evolution by implementing migrations manually is tedious and error-prone. Through coupled evolution, the evolution process can be automated by generating migrations automatically. For example, for databases, coupled evolution can automate the migration of data when the schema is adapted. In the metamodeling space, coupled evolution can automate the transformation of models when their metamodel evolves.

Various approaches to coupled evolution in the context of conformance exist in various technological spaces within software engineering. Each space uses its own terminology. Each space faces its own space-specific require-

ments, such as high performance for migration of large databases, or the complexity of migrating models under metamodel constraints that go beyond the typical structure restrictions found in databases (e.g. inverses, or cardinalities). Each space offers its own solution directions. New coupled evolution approaches can benefit from existing approaches by reusing concepts and solutions. Yet, it is largely unknown how approaches from different technological spaces relate to one another. What are the commonalities and differences? Which ideas can be ported to other spaces? And which concepts are space-specific?

Despite a significant body of research, existing approaches are frequently obtrusive – requiring changes to the development methodologies – or not applicable to realistic cases – offering support for simplistic evolution scenarios or small-sized artefacts. The goal of this thesis is to support the evolution process, by seeking a coupled evolution approach, which is non-obtrusive, in line with existing development methodologies, requiring little additional effort and that reduces the likelihood of error and data loss. Next, we seek to generalize the coupled evolution solution to the various technological spaces.

1.6 CHALLENGES & RESEARCH QUESTIONS

1.6.1 *Coupled Evolution Across Technological Spaces*

Coupled evolution and conformance occur in various technological spaces, yielding numerous solution approaches. Most of these approaches address a single space, thus making their implementation space-dependent. Nevertheless, ideas and concepts are generally more widely applicable and although rarely done, they may well be used in other spaces.

Some spaces offer publications comparing approaches [Roddick, 1992, Casais, 1995, Benatallah, 1999, Rashid and Sawyer, 2005, Rose et al., 2009]. However, it is largely unknown how approaches from different technological spaces relate to each other. Consequently, most concepts are reinvented repeatedly for each space. New techniques facing the coupled evolution problem are likely to start from scratch, mostly being unaware of existing research.

To prevent having to reinvent solutions, we need to find and compare existing approaches from the different spaces and to identify their commonalities and differences across spaces. We aim to reveal reusable concepts, that can help new approaches to start from a solid and proved basis instead of from scratch. Also, we target to find avenues for further research, allowing the discovery of new solution areas.

Surveying publications across spaces is not a simple task. Obtaining a complete set of publications on coupled evolution for a single space is hard, obtaining a complete set of publications for multiple spaces, which generally publish to different venues, or publish in different journals is even harder.

Additionally, comparing publications between spaces is difficult. Different spaces tend to use different terminology, to some extent different concepts and frequently focus on more or less space-specific restrictions (such as the large size of a data set in the dataware space, or the rich constraint set defined in metamodels in the modelware space). We aim to find existing solutions that can be reused across spaces and to identify avenues for further research. These problems amount to the following research question:

Research Question 1

How do we characterize and compare coupled evolution approaches across technological spaces?

The commonalities between different approaches make different solution approaches implement similar functionality. As coupled evolution in the context of conformance is a domain of active development, new approaches arise frequently. To prevent the repetition of work, and to alleviate the commonalities between approaches, we aim to identify the underlying concepts and capture them in a reusable space-independent framework to coupled evolution. Therefore we ask the following research question:

Research Question 2

How can coupled evolution concepts and solutions be generalized across technological spaces?

1.6.2 *Coupled Evolution Design*

Coupled evolution approaches use a set of coupled operators, which preserve conformance, to automate evolution. The quality of the approach largely depends on the quality of the operator set. The set should be large enough to cover common and realistic evolution scenarios. Yet increasing the size of the set reduces its usability, thus requiring a careful operator selection as well as an operator organization to ease selection.

Existing approaches implement small sets of operators, offering a proof of concept of the approach rather than a practically viable solution. They are neither applied to realistic cases, nor compared to operator sets from other solutions. Therefore, we analyze evolution for the modelware space. We aim to find common evolution patterns and to determine their characteristics (such as their effect on the conformance relation) as well as their automation potential through coupled evolution.

Research Question 3

What metamodel evolution patterns can be distinguished, which allow automation in the context of migration?

Evolution is generally implicit, yet needs to be formalized to be used by coupled operators. A formalization needs to relate back to the actual evolution, such that it is understandable by a developer; needs to be sufficiently complete to cover arbitrary evolution scenarios; and needs to be automatically processable.

Research Question 4

How can software language evolution be formalized, such that it both functionally and understandably represents the developer's evolution intent?

1.6.3 *Coupled Evolution Implications*

The main goal of coupled evolution in the context of conformance is to ease software evolution by automating the migration of conforming artefacts. Coupled evolution prevents having to manually construct artefact migration upon evolution. As evolution is an ongoing process, coupled evolution is not a one-time event, but needs to be present throughout the software lifetime. Continuous coupled evolution support needs to fit into the software development process. Major development process changes are likely to hamper software development and unlikely to take ground.

Existing approaches to conformance-preserving coupled evolution have a significant impact on the software development process. Some approaches require the developer to consider evolution twice: namely, once by applying the evolution as in regular software development and once by specifying it explicitly (e.g., [Rashid and Sawyer, 2000, 2005]). Other approaches restrict development to a specific (generally recording) editor (e.g., [Herrmannsdoerfer et al., 2009]). Yet other approaches may pose an implicit and generally hidden risk of data loss on the migration process (e.g., [Hibernate, 2008]).

To encourage the use of coupled evolution, we focus on reducing the impact coupled evolution has on regular development as much as possible. Although deriving a migration from an explicit evolution is generally fast and fully automated, specifying the evolution and executing the migration requires manual effort. Therefore, we search for ways to improve the process of both and thereby ease the development process. Additionally, software evolution poses a risk to software usage. Availability of running systems may be compromised and information may be lost unintentionally. In particular, manual migration has a risk of human-error. Coupled evolution automates the evolution process. On the one hand, this reduces the risk when common

evolution steps are reused and repeatedly tested. On the other hand, automation reduces developer checks and increases the risk of error. The latter can cause undesired data loss. To prevent data loss, yet not hamper software development, we search for techniques to prevent data loss, without requiring significant development effort.

Research Question 5

How do we support coupled evolution unobtrusively and prevent the undesired loss of information during migration?

1.7 RESEARCH METHODOLOGIES

In our work, we distinguish two research methodologies. Firstly, we do research of an analytical nature, in which we examine the status quo as to increase our understanding (Chapters 2 and 3). Secondly, we do research of a constructive nature, in which we develop new techniques, designs and theories to change and preferably improve the status quo (Chapters 4, 5, 6).

In analytical research, we examine the status quo by analyzing literature, or existing software evolution cases. As we cannot examine all possible evolution cases, we aim to select cases that are representative. Thus they need to be realistic and of significant size and complexity. We include industrial cases to complete the representation, yet prefer open (public) case studies, to allow reproduction of the results and to allow the results to be used in comparisons performed by other researchers.

Research of a constructive nature opens up new opportunities and possibilities. These new possibilities go beyond directly quantifiable improvements of for example performance. Better evolution support may drastically alter the evolution of software, not just by shortening or lengthening it, but by changing its course. Existing case studies only offer evolution using traditional techniques and are therefore not suited for validation of constructive research. To enable validation, we therefore developed new cases using new technologies. Both YellowGrass and Researchr (discussed in appendices A and B) started out as validation cases, yet grew to be much-used products. Their development was aided by the tools developed for this thesis. Although these case studies are less in number than the cases used in our analytical research, the cases offer valuable research input and because they have been examined in more detail are in most scenarios more valuable than cases of which the evolution is only recorded by a set of source code versions and software documentation.

Finally, research has a high risk of repeating past work. In Chapter 2, we can see that often publications address similar contributions yet lack a clear discussion of the differences. To prevent duplication, we performed an

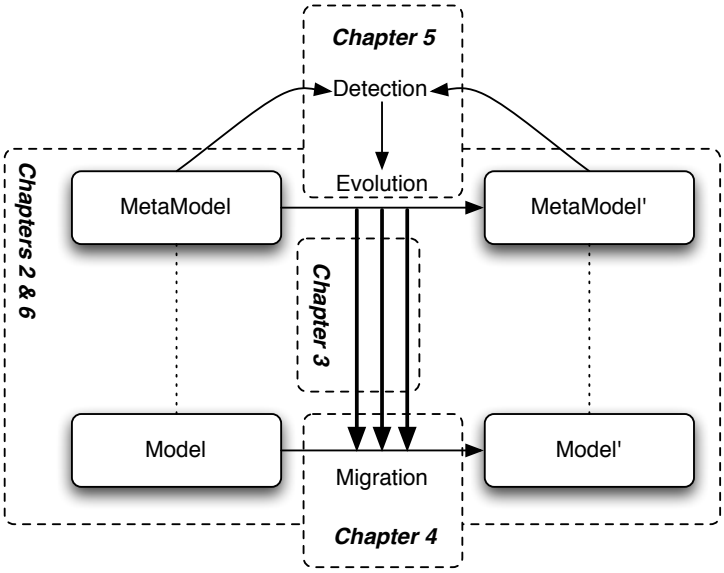


Figure 1.5 Overview of chapter topics

extensive literature survey and provide thorough reviews of related work in each of the chapters. We try to explain how our work compares to or differs from existing work and highlight the contributions it offers.

1.8 THESIS OVERVIEW

The different chapters discuss different topics in the space of coupled evolution. Figure 1.5 provides a graphical overview.

Chapter 2 discusses existing approaches to coupled evolution by means of a literature survey (research questions 1 and partially research question 2). It presents a space-independent feature model, focused on determining commonalities and differences between approaches. It addresses the application of the feature model and the interpretation of its results, thereby discussing specifics within spaces, avenues for future research, portability of techniques across spaces, and correlated features.

Operator-based coupled evolution approaches use an operator set to model evolution and derive migration. The success and applicability of an approach strongly depend on the quality of its operator set. *Chapter 3* discusses a catalog of coupled operators, which is based on an analysis of real-life case studies and a set of existing literature (research questions 3 & 5). It aims to be complete enough to apply to any realistic case, yet small enough to remain usable. The catalog is organized along operator criteria assessing operator impact on models and metamodels.

When using a model-driven approach to develop software, the runnable software is generally generated from the model in several step. When using coupled evolution to support evolution of the software model, it needs to bridge these steps in order to derive the correct migration. *Chapter 4* presents an implementation of coupled evolution for a web application language (WebDSL) and an underlying WebDSL database. It covers evolution representation, efficient database migration as well as how to bridge an object relational mapping in coupled evolution (research questions 4 & 5).

Coupled evolution needs an explicit evolution definition. Manually specifying such evolution is redundant and error-prone. Recording such evolution restricts development to a specific (recording) editor. As to not hamper the evolution process, *Chapter 5* discusses reconstruction of evolution (research question 5). It discusses reconstruction of complex evolution operators, addressing operator dependencies; mixed, overlapping and incorrectly ordered complex operator components; and operator interference, where the effect of one operator is partially or completely hidden by other operators.

Coupled evolution approaches occur in various technological spaces. These approaches focus on a single, homogeneous space, solving the coupled evolution problems locally and repeatedly. *Chapter 6* presents a systematic, heterogeneous approach to coupled evolution, providing space-specific transformation language generation and heterogeneous evolution interpretation (research question 2).

Chapter 7 concludes the thesis.

1.9 ORIGIN OF CHAPTERS

Except for chapter 2, the core chapters of this thesis are directly based on peer-reviewed publications. Chapter 2 to has been submitted for review. Each chapter has distinct core contributions and contains a certain degree of redundancy to ensure self-containment to allow them to be read separately. The authors of the publications forming the basis of chapters 2 and 3 are alphabetically ordered. There is an equal division of contribution between the authors of both publications.

- Chapter 2 is submitted for publication in ACM Computing Surveys and is titled *Coupled Software Language Evolution – A Survey across Technical Spaces* – [Herrmannsdoerfer et al., 2011]
- Chapter 3 is an updated version of the SLE 2010 paper *An Extensive Catalog of Operators for the Coupled Evolution of Metamodels and Models*. [Herrmannsdoerfer et al., 2010b]
- Chapter 4 is an updated version of the GPCE 2011 paper *Generating Database Migrations for Evolving Web Applications*. [Vermolen et al., 2011]

- Chapter 5 is an updated version of the SLE 2011 paper *Reconstructing Complex Metamodel Evolution*. [Vermolen et al., 2012]
- Chapter 6 is an updated version of the MODELS 2008 paper *Heterogeneous Coupled Evolution of Software Languages*. [Vermolen and Visser, 2008]

Other publications resulting from the research for this dissertation, yet which have not directly been incorporated in the following chapters are:

- *Generating Version Convertors for Domain Specific Languages* [G. de Geest and S. D. Vermolen and A. van Deursen and E. Visser, 2008]
- *Software Language Evolution* [Vermolen, 2008]

A Survey on Coupled Software Language Evolution

2

ABSTRACT

Like any software artifact, software languages are subject to evolution. When a software language evolves, existing language elements may no longer conform to the evolved language. To prevent loss of information, existing elements need to be migrated. Coupled evolution automates the migration of existing elements by attaching a migration specification to the evolution of a language definition. Software language evolution affects different technological spaces such as dataware, grammarware, XMLware, and modelware. In each technological space, different coupled evolution approaches have been proposed. However, it is largely unknown how these approaches relate to each other. To address this, we perform a systematic literature survey on coupled evolution approaches. We derive a feature model focused on determining commonalities and differences between approaches from different technological spaces. In this chapter, we present the application of the feature model and the interpretation of its results, within each technological space as well as across technological spaces. We address specifics within spaces, avenues for future research, portability of techniques across spaces, and correlated features.

2.1 INTRODUCTION

Various areas of computer science deal with information stored in artefacts (or *elements*). For example, programs store the description of an application, databases store application data and an XML document can store the configuration of an application. Together, we refer to these elements as *software* [Kleppe, 2008]. Common to all software is that the stored information is structured to some format. Programs conform to a grammar, databases conform to a data model and XML documents conform to a schema. The format of a piece of software, or more generally of a collection of software is described by the *software language*.

We say that software elements *conform* to a software language, when the software follows the structure outlined in the language. A software language is needed for the software elements to be understood, to be extended or to be processed automatically. *Breaking conformance* generally has as consequence

that some, or all of the information stored in the software artefact is lost. When a database no longer conforms to a data model (gets corrupted), data stored in the database may be lost. When a program is not syntax-correct, it cannot be parsed and thus not compiled. A recovery, or manual intervention is needed to prevent the program from being lost. Conformance to a language is essential for the preservation of information.

Due to changing requirements, not only software elements are subject to change, also software languages commonly need to be adapted [Favre, 2005]. A data model needs to be extended when new functionality is added to the application using the data described by the data model. A grammar needs to be extended when a new software design construct is introduced. For some types of software languages (e.g. data models), change is more common than for others (e.g. grammars), but all of them are bound to change at some point if they are in use. These changes may break the conformance of software. The software needs to be transformed to conform to the language again. We refer to such conformance-recovering transformations as *migrations*. When a software element is small (small programs or little data in the database), manually editing the element may provide a suitable migration. Yet, *manual migration* is tedious and error-prone, and thereby often leads to avoiding software language changes in practice [Casais, 1995]. Instead, *automated migration* is needed to recover the conformance relation.

Software language changes are rarely singular events. Changes are applied repeatedly and often continuously, constituting *software language evolution*. Evolution of software languages does not just require a single migration, it requires repeated migration to persist the conformance of elements. Even if a single migration is automated, constructing migration repeatedly is tedious and has a high risk of introducing bugs and thus losing information. To automate software language evolution, common evolution steps or patterns can be related to suitable software migrations, which is known as *coupled evolution* [Lämmel, 2004, Visser, 2008b]. Software languages are used in different *technological spaces* [Kurtev et al., 2002], such as programming languages, modeling languages, XML formats, and database schemas. Thereby, coupled evolution is also used in different spaces, to evolve different types of software elements.

PROBLEM. Different technological spaces propose different approaches to coupled evolution. Each space comes with its own terminology and approaches are only positioned within a single technological space. In some technological spaces, there are publications that compare the approaches proposed for the technological space with each other [Roddick, 1992, Casais, 1995, Benatallah, 1999, Rashid and Sawyer, 2005, Rose et al., 2009]. However, it is largely unknown how the approaches from different technological spaces relate to each other. Consequently, when a new technological space develops or the requirements for coupled evolution in a technological space change, new approaches are often developed from scratch—without considering the approaches already available in the other technological spaces. To overcome

this problem, we are interested in the following five research questions:

1. Which space-independent features can we identify to characterize approaches for coupled evolution in different technological spaces?
2. To which extent is each of these features represented in the different technological spaces?
3. What is the relation between the different features across technological space boundaries?
4. What is the relation between approaches in different technological spaces?
5. What are possible avenues for future research in coupled evolution?

CONTRIBUTION. To answer these research questions, we perform a systematic literature survey [Kitchenham and Charters, 2007] on coupled evolution approaches in different technological spaces. We systematically search publications on coupled evolution, using 29 initial sources for publications and exhaustive citation browsing. We select publications using clearly defined selection criteria, which we disambiguated by means of a first pilot study. We derive a feature model independent of technological spaces and disambiguate its application by means of a second pilot study. In this chapter, we present the application of the feature model and the interpretation of its results, within each technological space as well as across technological spaces. We address specifics within spaces, avenues for future research, portability of techniques across spaces, and correlated features.

OUTLINE. In Section 2.2, we first introduce the terminology used throughout the survey. The next sections follow the survey methodology as visualized in Figure 2.1. The larger boxes in the figure outline the main sections of the survey and discuss the methodology in detail: In Section 2.3, we present the publication selection process used to obtain a complete set of publications. Section 2.4 discusses the derivation of the feature model and its usage in classifying coupled evolution approaches. We address the different technological spaces separately in Sections 2.5 to 2.8, discussing features within each technological space. Section 2.9 interprets the results of the survey across technological spaces. In Section 2.10, we evaluate the survey results in light of the methodology of the literature survey, before we conclude in Section 2.11.

2.2 TERMINOLOGY

Different technological spaces have established different terminology. We rely on this existing terminology whenever we discuss a particular technological space. However, a unifying terminology is needed to compare approaches and solutions from different technological spaces, such as in a feature model

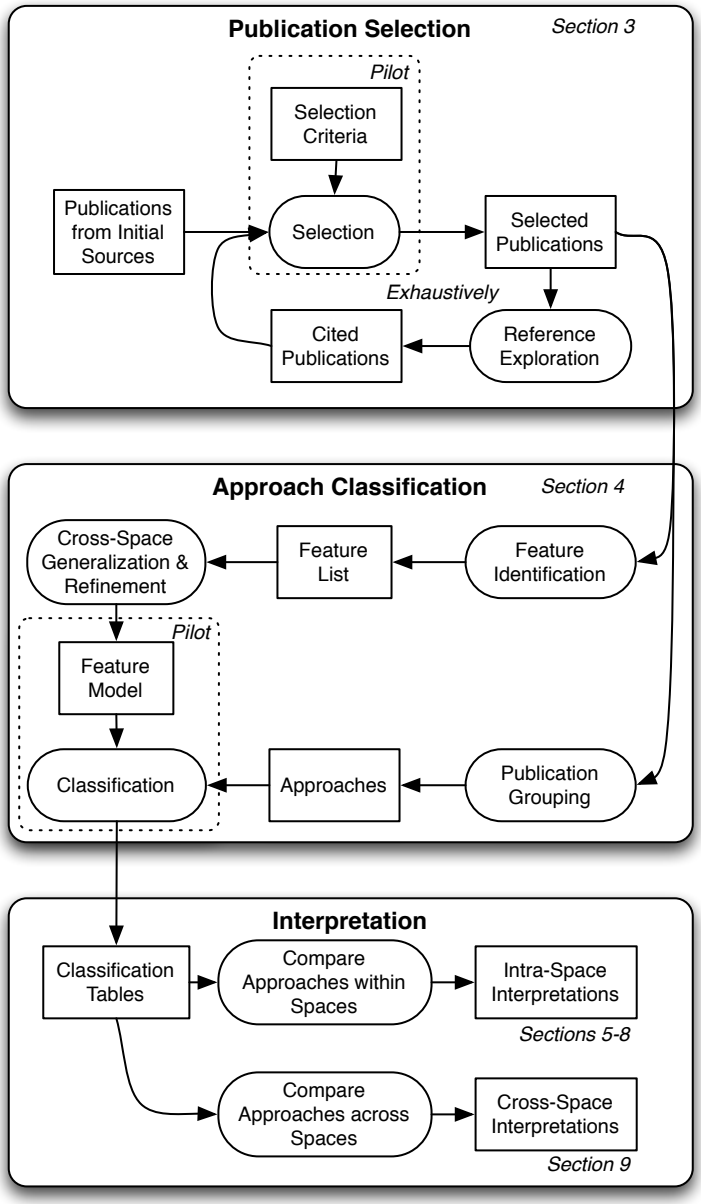
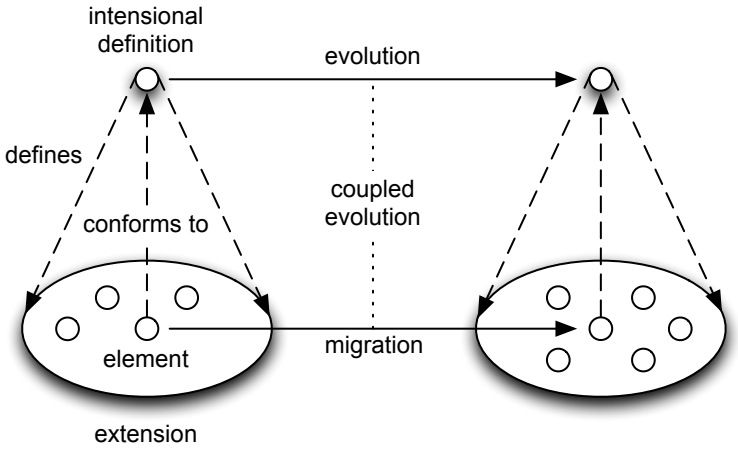


Figure 2.1 Survey methodology outline



	intensional definition	extension	element
dataware	schema	--	database
grammarware	grammar	language	word / sentence
XMLware	schema	format / language	document
modelware	metamodel	language	model

Figure 2.2 Cross-space terminology

across technological spaces. Thus, we switch to terms adopted from linguistics [van Sterkenburg, 2003] whenever more than one technological space is involved. We outline these terms and their relations to existing terminology from different technological spaces in the remainder of the section and in Figure 2.2. The figure shows the terms from linguistics schematically and in the top row of the table. The following rows of the table mention the matching term in each of the technological spaces that we will be addressing.

Various *technological spaces* deal with *intensional definitions* of possibly infinite sets. Depending on the technological space, different terms have been established for such a definition, its *extension*—i.e., the defined set—and the *elements* of the set. Grammarware and modelware specify *languages* by *grammars* respectively *metamodels*. In grammarware, the elements of a language are either called *word*s or *sentence*s. In modelware, these elements are called *model*s. XMLware and dataware rely on *schemas* to define sets of *document*s respectively *database*s. While dataware provides no term for a set of databases, a set of documents is either called *format* or *language* in XMLware. For an element of the extension of a definition, we say the element *conforms to* the definition.

Intensional definitions are subject to *evolution*, triggering the need for element *migration*. This migration is often called *co-evolution*, since it depends on the evolution of the intensional definition. *Coupled evolution* addresses

the automation of element migration based on its dependency on evolution of intensional definitions. Coupled evolution is an example for *coupled software transformations* where multiple software artifacts must be transformed in such a way that they remain consistent with each other [Lämmel, 2004, Visser, 2008b].

Figure 2.2 summarizes the various terms used in the different technological spaces. It is important to not mistake the columns in the figure for metalevels. As the figure above the table shows, the terminology applies to a pair of metalevels — any pair of metalevels. The intensional definition resides at the higher level of such pair, the various elements and the extension reside at the lower level of such pair. Yet, when considering three meta levels (e.g. model, metamodel and meta-metamodel), the terminology can be applied to the lower two adjacent metalevels (to model and metamodel), but also to the upper two adjacent metalevels (to metamodel and meta-metamodel).

For example, the dataware space distinguishes three levels: Data at the lowest level, schema definitions at the second level, and data definition languages at the top level. Typically, data definition languages employ grammarware technology and are defined by a grammar. Such grammars reside in a metalevel above schema definitions. In a similar way, we find grammars for grammar definition languages residing in another metalevel above. Thus, we find grammars at different metalevels.

All grammars, schema definitions, and metamodels are intensional definitions of possibly infinite sets—regardless of their metalevel.

2.3 PUBLICATION SELECTION

A systematic literature survey requires a thorough publication search strategy to cover all research conducted within the scope of the survey [Kitchenham and Charters, 2007]. Unambiguous selection criteria are needed to refine the set of found publications on relevance. Figure 2.1 outlines our process for publication selection in the topmost box. In this section, we discuss the selection criteria, the disambiguation of the selection criteria by a pilot study, and the application of the criteria in a search strategy to yield a complete set of publications.

2.3.1 *Selection Criteria*

The survey covers published literature, with the exclusion of workshop publications and technical reports. We set out the scope of the survey by means of a set of inclusion and exclusion criteria, presented below. Publications falling within the relevant technological space, yet rejected based on the selection

criteria, are recorded along with the reason for rejection.[1]

This survey focuses on *coupled evolution* of an intensional definition and its elements. 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 determines element migration. Manual migration of individual elements falls outside the scope of the survey due to a lack of coupling to evolution. Tool-supported manual construction of an executable migration specification falls within the scope of the survey.

We exclude work focused on comparison of intensional definitions, since these do not discuss a coupling of migration to evolution. Such comparison includes work on change detection, model comparison, difference calculation and difference representation. We also exclude 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 exclude work on views on elements, when these were not explicitly called in to prevent or aid coupled evolution.

The space of ontology evolution is considered out of scope of the survey, since it currently does not take into account element migration. API evolution is considered out of scope, since the extension is not completely defined by the intensional definition.

2.3.2 *Pilot Study*

To ensure unambiguously defined criteria, we perform a first pilot study: We randomly selected 25 potentially relevant publications from our set of initial sources (discussed below). Subsequently, each of the three author of this survey independently applied the selection criteria to each of the publications, yielding three independent sets of selected publications. We compare the resulting sets.

Out of the 25 publications, there appears to be disagreement in two cases and the selection criteria appear to be hard to apply to a third. Consequently, we have improved the criteria: To resolve the first disagreement, we have added the restriction on subsequent versions in evolution to show clear resemblance to exclude migrations between independent intensional definitions. Due to the second disagreement, we have excluded manual migration, but included manually written migration specifications. To resolve the difficulty to apply the criteria, we have included publications on views only if these

[1] `http://swerl.tudelft.nl/twiki/pub/Main/SanderVermolen/ce_survey_excl.pdf`

views are explicitly used to support coupled evolution, and have excluded publications addressing views which may support coupled evolution, but are not actually used to this extent by its authors.

2.3.3 *Search Strategy*

The rigor of the search process is a distinguishing factor for systematic literature surveys versus traditional surveys [Kitchenham and Charters, 2007]. Following an iterative process, we have set a search strategy and follow it throughout the survey. The search strategy comprises two stages: A selection of relevant publications from a large set of conferences and journals (the initial sources), and by exhaustive recursion, following relevant references of all publications included in the survey. Figure 2.1 outlines the search strategy graphically in the topmost box.

As a starting point of the survey, we comprise the set of relevant conferences and journals shown in Figure 2.3. By studying all editions of each of these journals and all occurrences of each of these conferences, we select relevant publications by application of the selection criteria. The set of conferences and journals is not intended to be a complete set containing all relevant literature. It merely provides an initial set of publications.

To complement the initial sources, for each publication, we include all cited publications relevant to the survey. By applying reference inclusion recursively, we expand the survey outside the scope of the initial sources. By applying the recursive reference inclusion exhaustively, we complete the set of selected publications.

We deliberately do not use keyword searches to find initial sources. Due to the differences in terminology within or across technological spaces, completeness is hard to achieve. Moreover, Brereton et al. [2007] recently observed that "current software engineering search engines are not designed to support systematic literature reviews"; this result was confirmed by Staples and Niazi [2007].

2.3.4 *Selection Results*

Exhaustive application of the search strategy yielded a total of 86 publications. Figure 2.4 shows the number of publications for each technological space as well as for each of the last decades. Coupled evolution appears to be 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 relatively new topic in the modelware space, this is where coupled evolution currently draws most attention.

Acronym	Full Name	Years	# P.
Conferences			
BNCOD	British National Conf. on Databases	1981 - 2009	505
CAiSE	Int. Conf. on Advanced Information Systems Engineering	1989 - 2009	1,418
CIKM	Int. Conf. on Information and Knowledge Management	1992 - 2009	2,489
CSMR	Europ. Conf. on Software Maintenance and Reengineering	1997 - 2009	518
ECMFA	Europ. Conf. on Modeling Foundations and Applications	2005 - 2009	155
ECOOP	Europ. Conf. on Object-Oriented Programming	1987 - 2009	692
EDOC	Int. "Enterprise Computing Conference"	2000 - 2009	474
ER	Int. Conf. on Conceptual Modeling	1979 - 2009	1,893
GTTSE	Generative and Transformational Techniques in Softw. Eng.	2005 - 2007	39
ICDE	Int. Conf. on Data Engineering	1988 - 2010	3,441
ICMT	Int. Conf. on Model Transformation	2008 - 2009	57
ICSE	Int. Conf. on Software Engineering	1976 - 2009	3,338
ICSM	Int. Conf. on Software Maintenance	1993 - 2009	1,094
MODELS	Int. Conf. on Model Driven Eng. Languages and Systems	1997 - 2009	495
OOPSLA	Object-Oriented Progr., Systems, Languages & Applications	1986 - 2009	1,823
SLE	Int. Conf. on Software Language Engineering	2008 - 2009	47
VLDB	Int. Conf. on Very Large Databases	1975 - 2009	2,525
WCRE	Working Conf. on Reverse Engineering	1993 - 2009	606
Journals			
JSME	Journal of Software Maintenance and Evolution	1989 - 2010	266
JVLC	Journal of Visual Languages and Computing	1993 - 2010	500
KAIS	Knowledge and Information Systems	1999 - 2010	502
SIGMOD	ACM's Special Interest Group on Management of Data	1977 - 2009	1,552
SIGPLAN	ACM's Special Interest Group on Programming Languages	1987 - 2010	1,453
SoSyM	Software and Systems Modeling	2002 - 2010	237
TKDE	IEEE Transactions on Knowledge and Data Engineering	1989 - 2010	2,084
TOPLAS	ACM Transactions on Programming Languages and Systems	1979 - 2010	920
TOSEM	ACM Transactions on Software Eng. and Methodology	1992 - 2010	267
TSE	IEEE Transactions on Software Engineering	1975 - 2010	2,972
VLDBJ	Journal on Very Large Databases	1992 - 2010	492

Figure 2.3 Initial publication sources

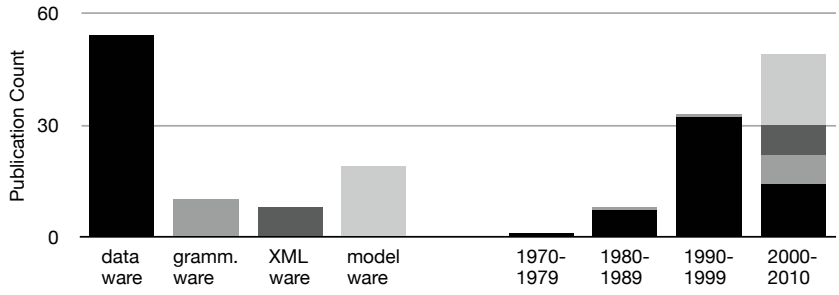


Figure 2.4 Selected publications

2.4 APPROACH CLASSIFICATION

To be able to compare approaches from different technological spaces, we need a scheme according to which we can classify all approaches. We use a feature model to represent this classification scheme, as it allows us to define the features of the different approaches as well as how they can be composed. Figure 2.1 outlines our process for classifying the approaches. In this section, we present the grouping of publications to approaches, the derivation of the feature model, the resulting feature model, its disambiguation in a pilot study, and its application to approaches.

2.4.1 *Grouping Publications to Approaches*

Different publications frequently address the same approach. Consequently, they generally offer similar characteristics. To prevent duplicating classifications, we focus on approaches rather than individual publications. We group several publications into one approach if they address the same tool, tool set, or methodology and offer the same characteristics. They generally also share some or all of the authors. The result is a list of approaches which are shown in Figures 2.6 to 2.10.

2.4.2 *Deriving the Feature Model*

Figure 2.1 shows how we derived the feature model from the selected publications: we identified features in publications from different technological spaces, generalized them across technological spaces, and structured them into a feature model.

First, we studied all the selected publications and extracted properties that characterize the presented approaches. The result after studying all the publications is a (large) list of used features. Next, we identified and combined similar features, thus condensing the feature list. Additionally, we generalized features to make them applicable to all technological spaces. Repeated com-

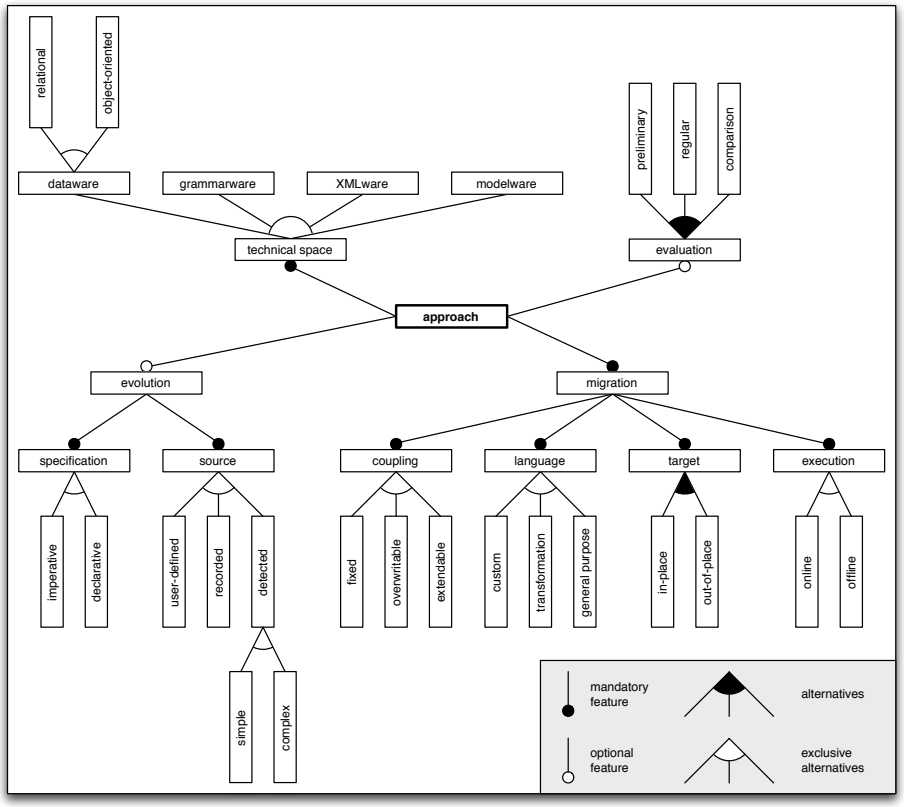


Figure 2.5 Feature model for the classification of coupled evolution approaches

bination and generalization yielded a list of consolidated features. Finally, we grouped alternative features into composite features, yielding a feature model. At the topmost level, we grouped the features according to the terminology introduced in Section 2.2. The resulting feature model is shown in Figure 2.5 and explained in the following section.

To ensure that the feature model is sufficiently unambiguous when applying it to approaches, we performed a pilot study as explained in Section 2.4.4.

2.4.3 *Resulting Feature Model*

Figure 2.5 presents the feature model to classify approaches from different technological spaces. We now discuss all the features in detail.

Technological Space. An approach is typically restricted to a particular technological space. We cover the technological spaces of *dataware*, *grammarware*, *XMLware*, and *modelware*. In dataware, we distinguish approaches which address *relational* and *object-oriented* database management systems.

EVOLUTION. When a developer edits and thus evolves an intentional definition, for example a metamodel, he applies changes (edits) to create a new metamodel version. When changes are done, it is the result of the changes — the new metamodel version — which is saved, not the changes themselves. The evolution of a metamodel, or more generally, an intensional definition, is thereby implicitly recorded in the original and the evolved version of the definition.

However, many coupled evolution approaches are based on explicit evolution specifications. They need the applied changes, rather than the result of these changes. Thus the changes need to be made explicit in a change specification, known as an *evolution specification*. We distinguish two styles of such specifications: *Imperative* specifications describe the evolution by a sequence of applications of *change operators*. Change operators are representations of edits applied to the intensional definition, such as renaming a class, moving an attribute from one class to another, or merging two classes in a metamodel. In contrast, *declarative* specifications model the evolution by a set of differences between the original and evolved version of a definition. They do not record the changes that a developer has applied in the evolution and thus do not record how the evolution took place. Rather, they record the effect of the evolution.

In most development environments, evolutions are stored implicitly by storing versions in a versioning system. The explicit evolution specification then needs to be derived from these versions in a process known as evolution *detection*. We distinguish two kinds of detections: First, detections which are only able to detect *simple* changes. Simple changes are atomic and can thus not be decomposed further, they typically include additions, deletions and renamings. Examples are adding an inheritance link, deleting a class, or renaming an attribute. For some approaches, simple changes include moves as well, when these are considered atomic. Second, detections which can also detect *complex* changes. Complex changes can be decomposed into (more than one) simple changes. They capture part of the intention that the developer had with his edit. For example, the merge of two classes can in simple changes be recorded implicitly as a several attribute additions, and a class removal, whereas in complex changes we can record the intention by specifying the class merge explicitly.

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.

MIGRATION. In contrast to evolution, migration is always specified explicitly. Modelware uses model transformations, dataware uses database migrations and grammarware uses program transformations. The migration has a direct relation to the evolution it is constructed for. For example, when a class is deleted in evolution, any objects of that class need to be deleted in migration. The relation between evolution and migration, is used to couple

particular migrations to particular evolutions, or evolution patterns. For example, the deletion of class x in evolution can be coupled to the removal of all x objects in migration. By coupling a suitable migration to an evolution (pattern), the migration can be reused when the same evolution needs to be addressed, or in the case of an evolution pattern, when a similar evolution needs to be addressed. A combination of evolution pattern to a suitable migration, provides a reusable evolution step, which we call a *coupled operator*. A coupled operator transforms both the intensional definition and a conforming element, thus preserving the conformance relation between the two.

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 migration specifications. With an *extendable* coupling, the user can add completely new couplings between elements of evolution and migration specifications.

Approaches with overwritable coupling need to provide a language to specify the custom migration. 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 technological space addressed by a coupled evolution approach. Common examples are SQL or OQL for dataware, XLT or XQuery for xmlware and ATL or QVT for modelware. Another way is to add migration support to a *general-purpose programming language* (GPL) in form of an API or an internal domain-specific language (DSL).

Migration might be performed either *in-place* or *out-of-place*. In the first case, the target of the migration is the original element itself which is modified during migration. In the second case, the target is a new migrated element which is created during migration. The original element is preserved. Creating new elements is only possible when the element-size allows. Thus out-of-place migrations are mostly applied to smaller elements, such as models and in-place migrations are mostly applied to larger elements, such as databases.

Furthermore, the migration might be executed *offline* where applications cannot use some of the elements during the migration, or *online* where applications can still use all elements and where the usage of an element by an application triggers lazy migration. Online migrations are needed when the element is in constant need. Databases, may serve live applications and need to stay online, thus needing an online migration. On the other hand, models are typically only used as input to transformations such as compilation and can thus be transformed offline while not in use.

EVALUATION. Evaluation is crucial for the validation of coupled evolution approaches. It is not a characteristic of the approach itself, but rather of the development of the approach and the publications on the approach. Approaches might receive no evaluation at all. They might receive only evaluation of *preliminary* nature, e.g. by toy examples. Other approaches include

regular evaluation on industrial or open-source systems of medium to large scale. Some authors provide a *comparison* of their approach with existing approaches.

2.4.4 *Pilot Study*

Using the feature model, we classified all approaches by reviewing their respective publications. To ensure that different reviewers classify an approach according to the same features, we conducted a second pilot study. We randomly selected five approaches from different technological spaces and classified them independently of one another. We compared the classifications with respect to inter-rater agreement.

We found a high agreement for all features except for the following issues: First, it was difficult to distinguish user-defined from recorded evolution source, because user-defined evolution can be considered a variant of recording using a regular editor. To alleviate this issue, we decided that in case of recording, an explicit representation of changes is directly recorded by the editor. Second, the difference between GPL and custom language was not unambiguously defined, due to internal languages and APIs. We refined the definition of the features to classify internal languages and APIs as GPLs, since the host language is a GPL. Third, it was difficult to distinguish in-place from out-of-place migration for database views. However, we classified them as out-of-place, since the original data is not changed in response to the calculation of views.

2.4.5 *Classification Results*

In the following sections, we organize approaches by technological spaces. Each section covers one technological space. A table at the beginning of a section summarizes the approaches for a technological space and their classification according to the feature model. Each approach and its classification is then discussed in more detail throughout the section. Some publications do not present or discuss a particular approach for coupled evolution, but still contribute to the research on coupled evolution in a technological space. These publications cannot be classified as approaches, but we discuss them separately at the end of each section.

2.5 DATAWARE

Dataware deals with data stored in databases (the elements). The structure of this data is described by a schema or data model (the intensional definition). A data model defines a collection of databases which can be processed by

an application (the extension). As any other type of application, database applications evolve over time. As data models are an intrinsic component of database applications, application changes that result in changes to the data model are relatively frequent [Sjøberg, 1992]. Consequently, modifying database application without support for evolution of the data model is a troublesome task.

As an example, consider a case study by Sjøberg [1993]. He measured evolution and its impact in a health management system, comprising 150k lines of code. 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 6000 code locations.

Schema evolution has been a field of study for several decades, yielding a substantial body of research. 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 topic of database schema evolution, *Roddick* [1992] presents an annotated bibliography. The bibliography also categorizes publications along the evolving formalism into evolution of relational data models and object-oriented data models as well as miscellaneous works. The relational papers present temporal extensions for the relational model to deal with schema evolution, discuss the implementation of schema changes and propose to use schema evolution for schema integration. The object-oriented papers present taxonomies of operations for schema evolution, propose versioning approaches and discuss different techniques for converting data, including immediate and lazy conversion. The miscellaneous papers present extensions of existing languages to support schema evolution and discuss the impact of domain evolution on schema evolution. Relevant publications from the bibliography are included in this survey.

2.5.1 *Technological Space Specifics*

There exists a long line of development and innovation on dataware—both in academia and industry—yielding a broad range of schema evolution traces and (large) databases. On the one hand, database migration, which is the transformation of both the data and data structure (schema) inside a database, faces wide-spread adoption. On the other hand, a coupling of such migrations to the evolution of the schema to allow reuse, receives much research, yet less adoption.

Dataware generally deals with few large (or very large) elements, applying the following restrictions on coupled evolution:

- Manual editing of elements, manual validation of elements and manual intervention during migration is hard or impossible.
- Performance of migration is an important factor.
- Schema evolution and data sets are mostly distributed across machines.
- Evolution of a schema frequently requires migration of multiple elements, implying the need for a distributable migration.
- Loss of data generally has significant consequences, as it usually takes time to restore backups of the data and the migrated data may already be changed.

Database management systems frequently have internal support for versioning of data [Roddick, 1995] and views on data [Halevy, 2001]. Additionally, databases are generally at the heart of running software, which cannot be stopped for migration, setting demands on the availability of the data set.

2.5.2 *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 implicitly within records, yet their consistency is generally ensured by the database system. Figure 2.6 lists the approaches from relational dataware together with their classification according to the feature model presented in Section 2.4. We distinguish a manual specification approach and operator-based approaches.

Manual Specification Approaches. When a user constructs a database migration manually and preserves the coupling to evolution, we speak of manual specification approaches, referring to the manual specification of the migration. *Ronström* [2000] presents an approach for online schema evolution and migration of a telecom database. 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.

Operator-based Approaches. Operator-based approaches specify coupled evolution as a sequence of coupled operators. Coupled operators encapsulating both schema evolution and database migration. When applied, they preserve the conformance between schema and database.

Shneiderman and Thomas [1982] propose an architecture for the coupled evolution of relational schemas and databases as well as applications and programs. They present 15 coupled operators for schema transformation and discuss their effect on databases in terms of a relational algebra. Though based

	Evolution						Migration										Eval.		
	Spec.		Source				Coupl.			Lang.			Tgt.		Exec.				
	imperative	declarative	user-defined	recorded	detec. simple	detec. complex	fixed	overwritable	extendable	custom	TL	GPL	in-place	out-of-place	online	offline	preliminary	regular	comparison
Manual specification																			
Ronström	●		●					●				●		●	●				
Operator-based																			
Shneiderman	●		●				●				●			●		●			
Ambler SQL	●		●				●				●[1]		●			●		●	
PRISM	●		●					●			●[1]		●		●			●	

[1] SQL

Figure 2.6 Classification of the relational dataware approaches

on previous practical experiences with their own schema definition language, the approach is completely theoretical. Nevertheless, deducing from the fact that it is often cited by other coupled evolution approaches, it provided inspiration much inspiration. It is, amongst others, cited by [Sprinkle, 2003], [Lerner, 2000] and [Curino et al., 2008a].

Ambler and Sadalage [2006] propose an agile and evolutionary design of a relational database. 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 operators, tools to evaluate schema change effects, translation of old queries, automatic data migration, and documentation of intervened changes [Curino et al., 2009, 2008a,b]. Migration predictability is achieved by characterizing the extent of information preservation in response to schema changes, and by automating data conversion.

Additional Publications. Some publications do not discuss approaches to coupled evolution, yet are still relevant to the domain of coupled evolution. These publications improve the understanding of coupled evolution in the technological space, present approach-independent case studies, or discuss causes and consequences of coupled evolution. We discuss these publications per technological space. In relational dataware, there are four such publications.

Sockut and Goldberg [1979] introduce basic concepts of database reorganization. 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 and Heiler [1991] argue that, similar to database integration, domain evolution can create problems of semantic heterogeneity—i.e. clashes of implicit semantics. These are similar to those encountered in database integration and similar solutions apply.

Roddick [1995] discusses schema versioning issues. 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 new schema and later recorded data is viewable through previous schema; And schema modifications should be expressed in algebraic operations for formal verification.

2.5.3 *Object-oriented Dataware*

In object-oriented dataware, schemas are defined using classes, which can inherit from other classes and which define attributes or associations to other classes. A database consists of objects which are instances of these classes.

Casais [1995], Benatallah [1999], as well as Rashid and Sawyer [2005] present categories of approaches for object-oriented data-ware which we have combined to group the many approaches in this technological space. Operator-based and matching approaches modify the schema and database in-place and specify the schema evolution either imperatively or declaratively. Versioning and view-based approaches allow 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. Figure 2.7 enumerates the approaches from object-oriented dataware together with their classification.

Operator-based Approaches. Operator-based approaches specify schema evolution imperatively as a sequence of schema modification operators. An operator application not only adapts the schema, but also triggers in-place migration of the database to restore a consistent state.

Banerjee et al. [1987a, 1987b] present the semantics of a fixed set of primitive operators for ORION. The operators 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 Stein* [1987] , for Sherpa by *Nguyen and Rieu* [1989] and for F2 by *Al-Jadir and Léonard* [1998] . While GemStone supports only offline migration, Sherpa automatically propagates changes of a class to its instances. To improve implementation and performance, F2 splits objects into multiple

		Evolution						Migration										Eval.		
		Spec.		Source				Coupl.			Lang.			Tgt.		Exec.				
		imperative	declarative	user-defined	recorded	detec. simple	detec. complex	fixed	overwritable	extendable	custom	TL	GPL	in-place	out-of-place	online	offline	preliminary	regular	comparison
Operator-based																				
Banerjee	ORION	•		•				•						•		•				
Penney	GemStone	•		•				•						•			•			
Ngyuen	Sherpa	•		•				•						•		•				
Al-Jadir	F2	•		•				•						•			•		•	
Ferrandina	O_2	•		•					•				•[1]	•		•				
SERF	PSE	•		•						•		•[2]		•			•			
Schema matching																				
OTGen			•		•				•			•		•			•			
TESS			•				•		•			•		•			•		•	
Draheim			•			•			•				•[3]	•			•		•	
Class versioning																				
Skarra	ENCORE		•	•				•					•		•	•				
Monk	CLOSQL		•	•					•			•			•	•				
SADES	Jasmine		•	•						•		•			•	•				•
Schema versioning																				
Kim	ORION	•		•				•							•	•				
Andany	Farandole	•		•				•							•	•				
Clamen			•	•					•			•			•	•				
Lautemann	COAST	•		•					•			•			•	•				
Bouneffa	GORM	•		•					•			•			•	•				
View-based																				
Tresch	COCOON		•	•					•			•[4]			•	•				
EVER			•	•					•			•[5]			•	•				
Brèche	O_2		•	•					•			•[6]			•	•				
TSE	GemStone		•	•					•			•[7]			•	•		•		
Hybrid																				
Benatallah		•		•					•			•[2]		•	•	•				

[1] C++ [2] OQL [3] Java [4] COOL [5] EVER [6] VDL [7] MultiView

Figure 2.7 Classification of the object-oriented dataware approaches

objects (multi-objects), distributing inherited attributes to objects specific to the class they were inherited from.

In addition to high-level operators that are composed of primitive operators [Zicari, 1991, Brèche, 1996], *Ferrandina et al.* present operators to redefine and remove a class as a whole in O_2 [Ferrandina et al., 1995, Brèche and Wörner, 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 operator [Claypool et al., 1998, 2000]. However, the flexibility comes at the price that the migration can no longer be performed online.

Schema Matching Approaches. Schema matching approaches derive the in-place migration based on a difference between schema versions that is either recorded or detected. The difference 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 commands to initialize variables, perform context-dependent changes, move data, create objects and share information among objects.

TESS (Type Evolution Software System) derives migration rules by detecting changes between two schema versions [Lerner, 1997, 2000]. The detection is based on a comparison algorithm with three stages that compare classes by their name, use sites or structurally. TESS allows 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, showing that it performs well in case of strong naming and structural similarities.

Draheim et al.propose a matching approach for object-oriented schemas that are mapped to relational schemas via object-relational mapping [Draheim et al., 2004, Bordbar et al., 2005]. For simple changes, they generate SQL statements for updating the relational schema and Java code for data migration which can be customized. The migration is performed offline by cloning the data and in-place by turning off the constraints during migration. The approach has been evaluated in an industrial system for collecting distributed measurement data.

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

Monk and Sommerville [1992, 1993] propose to use update and backdate functions on classes to allow for more flexible migration. 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 online to any other version.

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

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

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

Clamen [1994] proposes to specify evolution declaratively by relating different schema versions which can also be used for schema integration. For each schema version, an interface is provided. Attributes can be shared between, independent of, derived from, or dependent on other interfaces. Whenever an attribute value is modified, dependent attributes in other interfaces need to be updated.

Lautemann [1996, 1997] proposes an approach which specifies the evolution imperatively. 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 functions. *Bouneffa and Boudjlida* [1995] presents a comparable approach, in which each object-schema version combination is represented by a facet. User-defined mapping functions map objects from one facet into another.

VIEW-BASED APPROACHES. 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 online.

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

Liu et al., Liu et al. [1993, 1994] enhance the graphical constructs used in Entity Relationship diagrams, and develop *EVER* (EVolutionary ER diagrams). EVER diagrams can be translated into relational or object-oriented database schemas. For each schema version, a consistent, updatable view is maintained. 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 a view-based simulation of capacity-increasing changes [Ra and Rundensteiner, 1995b, 1997, Crestana-Jensen et al., 2000]. Schema changes mapped onto views are 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].

HYBRID APPROACHES. *Benatallah* [1999] proposes a hybrid approach that combines schema versioning and schema modification. When a schema change operator 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.

ADDITIONAL PUBLICATIONS. In object-oriented dataware, five publications do not address approaches, but focus on coupled evolution from different perspectives.

Casais [1995] presents a survey of techniques to manage class evolution in object-oriented systems. On the class level, he distinguishes: tailoring which creates subclasses; surgery which uses change primitives; versioning; and

reorganization which performs more complex changes. On the object level, Casais distinguishes: change avoidance, conversion, and filtering.

Meyer [1996] addresses schema evolution from the perspective of applications which need to persist objects. He discusses different forms of persistency, i.e., files, relational databases and object-oriented databases. Migrations are performed online but need to be implemented manually without any coupling to the evolution.

Pons and Keller [1997] propose to organize operators in a multi-level catalog in which operators from higher levels are implemented using operators from lower levels. The catalog shows which modifications can be performed to the schema, starting from the primitives a database system provides.

Li [1999] identifies important issues in research on object-oriented schema evolution. 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 [2008] present a cross-space generalization of coupled evolution and propose to generate a domain-specific transformation language for the metalevel . The generalized solution is applied by a coupled evolution tool set for object-oriented schemas and relational databases.

2.5.4 *Intra-Space Interpretations*

Schema evolution is generally user-defined to achieve flexibility. This is the case for 23 out of the 26 approaches we identified. More recent approaches generally focus on in-place and preferably lazy migrations, to cope with the trend of increasing database size.

Due to the risks of migrating large data sets, research on dataware has a stronger focus on preventing loss of data. Additionally, there is a more explicit focus on the impact of schema evolution on other artifacts than the database—mostly queries. Also, attention goes to the process of applying coupled evolution [Roddick, 1995].

Originating in database support for versioning, an extension to schema and class versioning is found in 8 out of the 26 approaches. Continuing this extension, research has focused on supporting different software and different versions of the same software in parallel on a single data set, thereby satisfying availability requirements on the data set. Continuous support for multiple versions involves support for reading and writing to the same data from different perspectives of different schema versions, as well as support for forward and backward migrations to convert data from one perspective to another [Monk and Sommerville, 1993]. A field of research within dataware focuses on using data views for schema evolution, comprising 4 out of the 26 approaches we found. Views neither change nor version the data, but allow reading and generally writing of data from different version perspectives.

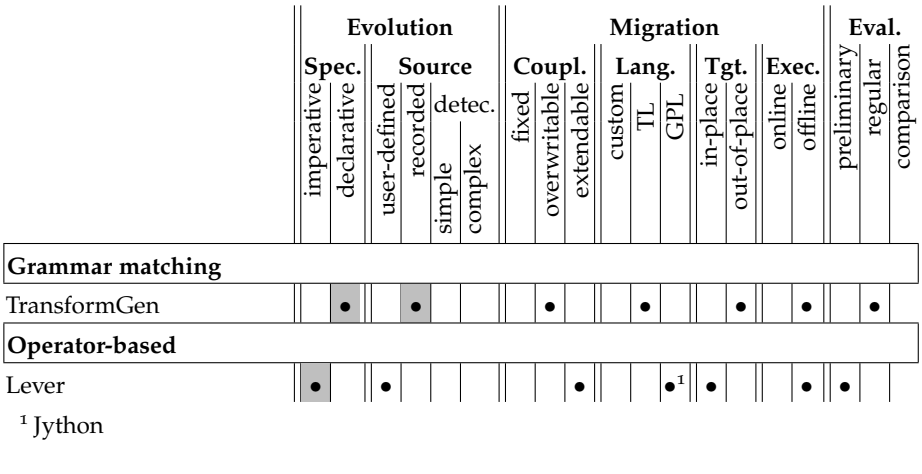


Figure 2.8 Classification of the grammarware approaches

From the rich history of dataware, one would expect that evaluation of approaches is most thorough. This is clearly not the case. Only 6 out of 26 approaches received some form of non-preliminary evaluation.

2.6 GRAMMARWARE

Grammarware deals with grammars (the intensional definition), which define a language (the extension). An element of a language is referred to as a word, sentence, or in the context of a programming language, a program.

In contrast to the dataware space, migration is less common in the grammarware space. Main programming languages (such as Java or C++) try to avoid the need for migration. New versions of such languages typically include older versions of the same language. Although this may degrade the quality of the language, and thus the quality of the programs, it appears sustainable for slowly evolving languages. Domain specific languages on the other hand are less stable and need more extensive evolution support.

2.6.1 *Technological Space Specifics*

Grammarware is the traditional technological space of programming and programming languages. In comparison to databases, a program is typically spread over several textual artifacts, each of it tending to be rather small than big. Programs are edited manually or transformed automatically into other programs or other software artifacts. There is a long tradition in program transformation, typically performed out-of-place. Even typical in-place transformations like refactorings are often performed by out-of-place trans-

formation systems. In contrast to other technological spaces, there is a strong emphasis on language semantics in grammarware. If a language evolves syntactically, migration is required to preserve the semantics of programs. Typically, this is hard to achieve and thus programming language evolution tries to avoid the need for migration by maintaining backward compatibility.

2.6.2 *Approaches*

There are a few approaches which address migration explicitly by coupled evolution. As is shown in Figure 2.8, we group them into the two categories of grammar matching and operator-based approaches. Though originally proposed for modelware approaches [Rose et al., 2009], these categories fit here as well.

Grammar Matching Approaches. *TransformGen* comprises a grammar matching approach that infers the migration specification between two grammar versions from a recorded grammar history [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 migration specification 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.

Operator-based Approaches. *Lever* is an operator-based approach that provides a suite of operators coupling grammar evolution with word migration [Jürgens and Pizka, 2006, Pizka and Jürgens, 2007b,a]. Furthermore, it supports the migration of compilers. Lever comes with three DSLs embedded in a scripting language: One for specifying grammar evolution, one for specifying word migration, and another one offering abstractions on top of the other two for defining coupled operators. It has been evaluated using a fictitious evolution of a catalog description language.

Additional Publications. In grammarware, language evolution is considered an interesting problem on its own. Lämmel [2001] presents an operator suite just for grammar evolution. A similar operator suite is used in a lightweight verification method to maintain the correspondence between grammar versions [Lämmel and Zaytsev, 2009a], e.g.between different releases of the Java Language Standard [Lämmel and Zaytsev, 2009b].

Overbey and Johnson [2009] discuss a side effect when programming languages like Fortran and Java evolve but migration is avoided. In this case, old programs use outdated constructs instead of newer ones introduced later. For example, the introduction of assert statements, more advanced for loops or generics to the Java language, rendered older (generally more elaborate) constructions that had a similar goal outdated. They envision a refactoring-based

solution to the problem where refactorings replace the old constructs with the new and better ones.

2.6.3 *Intra-Space Interpretations*

Since languages are more stable than metamodels and schema, there are only two approaches for coupled evolution in grammarware. Furthermore, migration is often avoided by backward compatible evolution. Most research on language evolution focuses on understanding language evolution in grammarware by formalizing the effect of different evolution operators. One of the approaches for coupled evolution is also based on such operators, and the other is based on declarative evolution specifications. Since programs are development time artifacts which are most of the time unused, there is no need for online migration in grammarware. Thus, both approaches perform migrations offline. Using a general-purpose language for the migration is somehow surprising, since grammarware offers many program transformation languages. But these languages typically support only out-of-place transformations, while the Lever approach focuses on in-place migrations. Both approaches allow for user-defined extensions of couplings and are evaluated.

As opposed to general purpose languages, domain specific languages tend to evolve faster and more extensively. The rapidly increasing use of domain specific languages implies an increasing need for coupled evolution in the grammarware domain. Similar to what we see in the modelware domain over the past ten years, research on coupled evolution in the grammarware domain is therefore likely to expand in the upcoming years.

2.7 XMLWARE

In XMLware an XML schema (intensional definition) defines a language or format of XML documents (element). Schemas are expressed in schema languages like *DTD* [200, 2008] or *XML Schema* (XSD) [Walmsley, 2001], both recommended by the World Wide Web Consortium. Similar to databases in the dataware space, XML documents are often used by applications. The structure of the documents therefore needs to evolve when requirements or the context of the application change.

2.7.1 *Technological Space Specifics*

We can distinguish two kinds of XML documents: user and application documents. User documents are edited manually by a user and are automatically processed or transformed into other software artifacts. Examples are HTML documents, or XML configuration files. Like programs in grammar-

		Evolution							Migration											Eval.		
		Spec.		Source					Coupl.			Lang.			Tgt.		Exec.					
		imperative	declarative	user-defined	recorded	detec. simple	detec. complex		fixed	overwritable	extendable	custom	TL	GPL	in-place	out-of-place	online	offline		preliminary	regular	comparison
Manual specification																						
Tan	XSD		●	●						●		●				●		●				
Operator-based																						
XEM	DTD	●		●					●					●	●			●				
Lämmel	DTD	●		●					●				●[1]			●		●				
XEvolution	XSD	●		●						●			●[2]		●			●				

[1] XSLT [2] XQuery

Figure 2.9 Classification of the XMLware approaches

ware, these kinds of documents are unused most of the time. Application documents are employed by applications to store or retrieve data. Examples are XML databases, or XML-based log files. In contrast to the dataware space, XML documents are more portable than databases, but due to their relatively inefficient format XML documents are typically not accessed as frequently.

XML documents vary largely in size. Large documents like genome descriptions can reach a size which makes it impossible to entirely parse them. Furthermore, the XMLware space provides mature language and tool support for transforming XML documents. These transformations are typically performed out-of-place.

2.7.2 *Approaches*

Figure 2.9 shows the XMLware approaches as well as their classification. We distinguish manual specification and operator-based approaches. Again, these categories where originally proposed for modelware approaches [Rose et al., 2009] but fit here as well.

Manual Specification Approaches. *Tan and Goh* [2005] present a manual specification approach which requires the user to manually specify the migration of documents from one schema version to another. They propose an extension to XML Schema that allows to declaratively specify the differences to previous versions directly in the schema. Additions, removals, moves, and renames of elements and attributes are supported. The information is then used for migrating documents between different schema versions.

Operator-based Approaches. Operator-based approaches provide a set of reusable coupled operators, which are for xmlware mostly different from the operators found in dataware approaches. The user specifies schema evolution imperatively by a sequence of operator applications. Since the operators work at the schema level as well as at the document level, such a sequence specifies both schema evolution and document migration.

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

Lämmel and Lohmann [2001] suggest transformation operators for DTD schemas from which specifications for document migration are induced. The effect of the operators at the schema level are described as informal text, whereas the migrations are specified by XSLT. The operators preserve the well-formedness of both DTD schemas and XML documents. Moreover, the operators 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 [Guerrini et al., 2007]. Like XEM, it provides a complete, minimal and sound suite of primitive operators. At the schema level, the operators 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 DSL provides means for the specification of custom migrations [Guerrini and Mesiti, 2008]. This DSL extends the standardised XQuery Update.

2.7.3 *Intra-Space Interpretations*

There are few approaches for coupled evolution in XMLware. Main stream schema evolution seems to be either restricted to backward compatibility in order to avoid migration or to go hand in hand with manually written migration specifications. 2 of the 4 approaches address XSD as a schema definition language; The other two approaches cover DTD. Operator-based approaches dominate the technological space of XMLware, comprising a total of 3 out of 4. Only 2 of the 4 approaches allow user-defined extensions of the migration specification by overwriting. Though transformation is a central concept in XMLware, only two approaches reuse standard XMLware transformation technology for the migration. Because XML documents are typically not accessed frequently at runtime, there is not much need for online migration in XMLware. All approaches support only offline migration. Moreover, they have in common that they require user-defined specification of the evolu-

tion and are not evaluated. 2 of the 4 approaches support in-place migration needed for large XML documents.

2.8 MODELWARE

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

Evolution of metamodels is common and can have large impact on existing models. Additionally, because of the relative small model size, lack of need for constant availability of models and well defined model structure (rich metamodels), there is much potential for automation of migration upon evolution.

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

Another study on automated coupled evolution was performed by Herrmannsdoerfer et al. [2008]. They present a classification of changes with respect to their potential for automation. Applying the classification to the evolution of two industrial metamodels indicated a high potential for automation: For over 80 percent of the changes, the corresponding migration could be given automatically.

The combination of frequent evolution and high potential for automation triggered a large body of research and high number of publications on the topic.

2.8.1 *Technological Space Specifics*

Specification and execution of model migration is always separated in time, since there might be several models for a metamodel, possibly distributed over several machines. In addition to the metamodel, modeling languages often have an explicit semantics, and thus a migration specification is required to preserve the semantics of all the models. Models are usually rather small, since modeling is about abstracting away unnecessary details, and thus migration performance is usually not an issue.

The transformation of models from one metamodel to another is a well-established research area in modelware. However, existing model transformation languages are not particularly suited for specifying migration [Sprinkle, 2003]. Exogenous transformation languages [Mens and Van Gorp, 2006] target completely different source and target metamodels and thus require to specify the complete mapping from the source to the target metamodel, leading to a lot of identity rules. Endogenous transformation languages [Mens and Van Gorp, 2006] focus on the transformation of a model within the same metamodel and thus require the metamodel to stay the same, which is not the case for metamodel evolution. Exogenous transformation languages usually perform transformation out-of-place, and endogenous languages in-place.

Another research area related to model migration is model matching and differencing, which can be employed to compare different metamodel versions.

2.8.2 *Approaches*

Rose et al. [2009] compare different approaches to automate model migration in response to metamodel evolution. They identify three categories of approaches: manual specification, metamodel matching, and operator-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. Figure 2.10 lists all the modelware approaches and groups them according to three categories.

Manual Specification Approaches. Model migrations are specified manually through custom model transformation languages in manual specification approaches. Thereby, specific model migration constructs reduce the effort for building a migration specification.

Sprinkle et al.[Sprinkle, 2003, Sprinkle and Karsai, 2004] introduce a visual language to declaratively specify the differences between two versions of a GME-based metamodel. The Model Change Language *MCL* is another visual migration language targeting GME [Narayanan et al., 2009, Balasubramanian et al., 2009]. With both languages, the user does not only specify the metamodel differences, but defines a model migration based on them. This overwrites the default copying behavior. 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., 2010]. Here, only the model migration is specified. Differences between metamodel versions are not made explicit. Instead, Flock automatically copies

		Evolution						Migration										Eval.		
		Spec.		Source				Coupl.			Lang.			Tgt.		Exec.				
		imperative	declarative	user-defined	recorded	detec. simple	detec. complex	fixed	overwritable	extendable	custom	TL	GPL	in-place	out-of-place	online	offline	preliminary	regular	comparison
Manual specification																				
Sprinkle	GME		•	•					•		•				•		•	•		
MCL	GME		•	•					•		•				•		•			
Flock	Ecore								•		•				•		•			•
Metamodel matching																				
Gruschko	Ecore		•			•			•			•[1]			•		•			
Geest	MS DSL		•			•			•				•[2]		•		•		•	
Cicchetti	Ecore		•				•	•				•[3]			•		•			
AML	Ecore		•				•			•		•[3]			•		•		•	
Operator-based																				
Hößler	MOF	•		•				•							•		•			
Wachsmuth	MOF	•		•				•				•[4]			•		•			
COPE	Ecore	•			•					•			•[5]	•			•		•	

[1] ETL [2] C# [3] ATL [4] QVT [5] Groovy

Figure 2.10 Classification of the modelware approaches

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 Petri net example from Wachsmuth [2007b], Flock has been compared to migration specifications in model transformation languages ATL and Ecore2Ecore as well as to an operator-based approach with COPE which we introduce later.

Metamodel Matching Approaches. In metamodel matching approaches, the differences between two metamodel versions are automatically detected. 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 et al., 2007, Becker et al., 2007]. They propose automatic migration steps for resolvable changes and envision to support the user in overwriting the migration specification for unresolvable changes. The approach is only prototypically implemented and thus not yet evaluated.

G. de Geest and S. D. Vermolen and A. van Deursen and E. Visser [2008] apply a similar approach in the context of Microsoft DSL Tools. 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. [2008] also detect complex changes. 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, they characterize dependencies between complex changes [Cicchetti et al., 2009].

The Atlas Matching Language *AML* allows the user to parametrize 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 Petri net example from Wachsmuth [2007b], and on the Java metamodel from NetBeans.

Operator-based Approaches. Operator-based approaches provide—similar to corresponding grammarware and XMLware approaches—a set of reusable coupled operators that work at the metamodel level as well as at the model level. The operator sets often show much resemblance to the operator sets found in approaches for evolution of object-oriented databases.

Hößler et al. [2005] formalize a fixed suite of reusable coupled operators. The completely theoretical approach is based on a generic instance model supporting versioning and is neither implemented nor evaluated.

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

COPE adds tool support for the evolution of Ecore-based metamodels to EMF [Herrmannsdoerfer et al., 2009]. It provides an extendable suite of coupled operators. In addition to user-defined evolution specifications, COPE supports recording of operator applications. The operators are specified in a DSL embedded into a general-purpose language. COPE is the only modelware approach performing in-place migration. Its evaluation by reverse engineering the evolution of the Palladio Component Model and the Graphical Modeling Framework [Herrmannsdoerfer et al., 2009, 2010a] proved the applicability of operator-based approaches in modelware.

Additional Publications. There is one remaining publications which does not propose an approach, but which analyzes evolution operator suites.

Based on literature and several case studies, Herrmannsdoerfer et al. [2010b] derive a library of coupled operators for evolution of EMOF-like metamodels (Chapter 3). The library is complete with respect to practical usage, which is validated against several case studies. They classify the operators and present and apply coupled operator features, such as language preservation, model preservation and bidirectionality, which are further discussed in Chapter 3.

2.8.3 *Intra-Space Interpretations*

Most approaches target MOF or its implementations, in total 7 out of the 10 identified approaches. MOF is a well-recognized standard in modelware. Approaches targeting the same modeling framework can be easily compared with each other, leading to evaluations by comparison.

6 out of the 10 approaches use declarative evolution specifications, since they define or make use of declarative model transformation languages. In modelware, there is only one recording approach which is probably more complex to implement, i.e., most approaches focus on specifying the model migration after the metamodel evolution has been performed.

To be able to specify semantics-preserving model migrations, most approaches allow at least to overwrite the migration specification 7 out of the 10 in total. But only two approaches can be extended by reusable couplings. Most of the approaches reuse or refine existing model transformation languages (7 out of 10): Manual specification approaches tailor model transformation languages to migration (3 approaches), metamodel matching approaches synthesize difference specifications (3 approaches), and Wachsmuth's operator-based approach specifies operators in QVT (1 approach). Only one approach performs in-place migration, since exogenous transformation languages that are required for metamodel evolution do not support in-place migration. None of the 10 approaches can perform migration online—probably since models are design-time artifacts, thus being stored most of the time and not in use.

Finally, 5 out of the 10 approaches are evaluated, 4 of which at least on a regular level, thus exceeding a preliminary state.

2.9 INTER-SPACE INTERPRETATIONS

In the previous sections, we addressed the various technological spaces individually. In this section, we interpret approach classifications across technological space boundaries, yielding avenues for future research. We address common and uncommon features, discuss portability of techniques across space boundaries, and analyze correlations between features.

2.9.1 *Common and Uncommon Features*

To derive avenues for future research, we assessed which features received much attention and which features received little attention. To this end, we counted occurrences of each feature individually and compared these feature counts within each composite feature, thus obtaining a cross-space interpretation. In the following, we discuss the most common and most uncommon features and derive avenues for future research.

Most approaches (32 out of 42) choose to have evolution defined by the developer. Although this offers a user-verified and thereby most likely correct evolution trace, it also requires a developer to first apply needed evolution by editing the intensional definition and subsequently specify the same evolution to be used as input to coupled evolution. Most approaches acknowledge the redundancy in these steps, yet refrain from countermeasures such as detection or recording due to their complexity or practical limitations. Detection and recording are often mentioned as directions of future work. Yet, only 6 approaches actually support detection and merely 3 approaches support recording. The lack of means for obtaining evolution hampers the usability of coupled evolution approaches. Detection, recording, or other approaches to obtaining evolution need to be addressed in future research, to allow coupled evolution approaches to be used in real-life development.

Most approaches (22 out of 35 that are overwritable) choose to use a transformation language for specifying migration. 4 others choose to define a custom language, which generally resembles a transformation language, but offers language constructs particularly useful for migration, such as automatic identity migration of element parts that do not require adaptation. Nevertheless, still 9 approaches choose to use a general purpose language. Although this choice is usually not explicitly motivated, it suggests a lack of a suitable transformation languages within the technological space at hand. Transformation languages ease the construction of migrations and thus of coupled evolution. Yet they also ease overwriting existing coupled operators and thereby coupled evolution usability. Future research should fill the gap that many coupled evolution approaches face.

The type of migration is primarily determined by the element under migration. Modelware, XMLware and grammarware all deal with elements that do not require continuous availability, and therefore the approaches all favor the simpler offline migration. In dataware, 18 out of the 26 approaches choose to support online migration, as their database systems support live software. Along the same lines, the small element size in modelware, XMLware and grammarware enables a choice for out-of-place migrations for 12 out of the 16 approaches. In dataware, the database size generally enforces in-place migrations. Out-of-place migration is only used if the migration is online and the approach either uses versioning or views, which allow the migration to be executed lazily or be postponed until needed.

Finally, only 9 out of the 42 approaches present a regular evaluation. Additionally, 2 approaches compare to related approaches and 3 approaches provide preliminary (toy example) results. The remaining 28 approaches do not present evaluation at all. On the one hand, this opens evaluation and approach comparisons (possibly within a technological space) as directions of future research. On the other hand, it emphasizes the lack of evaluation cases and benchmarks, thereby proposing development of such.

2.9.2 *Feature Portability*

With the feature model from Section 2.4, we are able to classify approaches from the different technological spaces along the same criteria. Furthermore, we found approaches from all technological spaces to fit into categories which were originally proposed by Rose et al. [2009] for the modelware space. These categories are *manual specification*, *matching* approaches, and *operator-based* approaches. The unique feature of manual specification approaches is a *custom* migration language for *overwriting* a default migration manually. For matching approaches, it is a *declarative* evolution specification which is either *recorded* or *detected*. The unique feature of operator-based approaches is an *imperative* evolution specification as a sequence of operator applications. Notably, the dataware space offers a wider range of evolution techniques beyond these categories. Some of these techniques are specific to databases or schemas, yet others can be ported to different spaces. In this section, we discuss such portable techniques and features.

Element views offer a different presentation of the element, yet leave the element unchanged. Views are an integral part of many database systems and are commonly used to continuously support different versions of a schema [Liu et al., 1993, 1994, Ra and Rundensteiner, 1995a,b, 1997, Crestana-Jensen et al., 2000], or to support coupled evolution without having to change or recreate the original database [Tresch and Scholl, 1993, Brèche et al., 1995]. In modelware, grammarware and XMLware, (editable) views are not widely used, hence their application to coupled evolution is lacking. Nevertheless, the solutions that views can offer may prove equally beneficial in other spaces, posing views for coupled evolution as a direction for future research.

In addition to the features present in the feature model, dataware approaches can also be classified along the type of versioning they use. Versioning can be class-based [Skarra and Zdonik, 1986, Monk and Sommerville, 1992, Rashid and Sawyer, 2000], schema-based [Kim and Chou, 1988, Andany et al., 1991, Clamen, 1994, Lautemann, 1997, Bouneffa and Boudjlida, 1995], view-based or hybrid (a combination of the previous) [Benatallah, 1999]. Both data and schema versioning are common. As versioning primarily applies to dataware approaches at present, we decided to exclude it from the feature model. Yet, although versioning is not explicitly used in coupled evolution approaches in modelware or grammarware, there is generally an implicit and

frequently manual approach to versioning. Furthermore, there is a body of research covering versioning in the modelware space [Altmanninger et al., 2009], and its usage in the dataware space suggests direct applicability in the other spaces.

Similar to databases, models may be used for live systems. Coupled evolution of live systems requires in-place migrations. Similarly, the size of a model or a program may enforce in-place transformations. Yet, few techniques exist to support in-place transformations in modelware or grammarware. In-place transformations and their application in coupled evolution provides a direction for future research.

2.9.3 *Feature Correlations*

Correlations between features can help to identify feature combinations that are often used together or combinations that are rarely used together. Often used combinations may suggest that the features can be easily combined, and that these combinations should be exploited, when implementing an approach for a new technological space. In contrast, rarely used combinations may indicate that the features are hard to combine, and thus hint at possible avenues for future research.

To identify correlations between two features, the presence of a feature in an approach can be modeled as a binary variable. To measure correlations between two binary variables, we use the Phi correlation test [Hilderman and Peckham, 2007]. We have applied the Phi correlation test to all combinations of two features from different composite features. We excluded features within the same composite feature, as they are largely orthogonal to each other and thus by definition show a strong negative correlation. We also excluded the technological spaces as features, since the presence of certain features within technological spaces is already covered by the intra-space interpretations. In the remaining combinations, we found no two features that exhibit strong correlation. Figure 2.11 gives an overview over the weak correlations that we identified. In the following, we discuss the positive and negative correlations together, as they often imply each other.

Since declarative approaches define the evolution as a mapping between the versions of the intensional definition, they often detect the evolution specification, allow to overwrite the migration specification, and perform the migration out-of-place. However, they rarely allow the users to define the specification or restrict them to fixed couplings, and seldom perform the migration in-place. To improve declarative approaches, predefined couplings provide a means to reuse recurring migration specifications across intensional definitions, user-defined evolution specifications enable the definition of more correct migrations, and in-place migration increases migration performance—especially for large elements.

weak positive (+)			**weak negative** (–)		
user-defined	online	.48	user-defined	regular	.53
imperative	user-defined	.47	user-defined	offline	.48
imperative	fixed	.47	imperative	overwritable	.48
imperative	in-place	.45	declarative	fixed	.44
detected	regular	.45	imperative	detected	.43
declarative	detected	.45	declarative	in-place	.42
declarative	overwritable	.44	imperative	out-of-place	.41
declarative	out-of-place	.38	declarative	user-defined	.39
detected	offline	.35	detected	online	.35
offline	regular	.34	imperative	custom	.34
custom	comparison	.31	GPL	out-of-place	.34
GPL	in-place	.31	out-of-place	regular	.34
in-place	regular	.31	fixed	TL	.34
recorded	regular	.31	online	regular	.34

Figure 2.11 Weak feature correlations ordered by strength

In contrast, since imperative approaches define the evolution as a sequence of operators, they often require the user to define the evolution specification, only support fixed sets of couplings, and perform the migration in-place. However, they rarely detect the evolution, allow to overwrite the migration, perform the migration out-of-place, and implement a custom migration language. Whereas out-of-place migration does not provide a clear advantage, detecting the operation sequence automates the migration definition, overwritable migration specifications enable more expressive specifications, and custom migration languages better support the verification of migration specifications.

Approaches that detect the evolution specification often perform migration offline and rarely online. In contrast, approaches with user-defined specifications often perform migration online and rarely offline. However, online migration is desired in situations in which the elements are used at the same time when the migration is performed. Approaches that extend a GPL to specify migrations often perform migration in-place and rarely out-of-place. It seems that embedded languages can be more easily extended to perform in-place transformation. Approaches that are restricted to fixed couplings rarely use an existing transformation language. Apparently, existing transformation languages provide appropriate means to specify expressive migrations, but do not provide a way to reuse migration specifications.

A regular evaluation is often conducted for approaches that are detected or recorded, perform the migration in-place or offline, but rarely for approaches that allow the user to define the evolution or perform the migration out-of-place or online. We conjecture that such approaches currently do not provide adequate tool support for applying them in practice. Therefore we recom-

mend further research into tool support for user-defined, in-place and offline coupled evolution approaches.

2.10 EVALUATION

Finally, we evaluate the survey results in light of the methodology of the literature survey by discussing potential threats to the validity of our results. We structure the threats according to the main phases of the survey methodology as depicted in Figure 2.1.

2.10.1 *Publication Selection*

When selecting publications, we might miss publications important for the survey due to a multitude of reasons.

The initial sources may not be complete enough to find all important publications: we might miss old or new publications, publications within a technological space, or a complete technological space. We mitigated the first two issues by exhaustive citation browsing, which however only helps to complete publications within a single technological space. To address the last issue, we also took cross technological space publications into account in which experts tie the ideas of different technological spaces together. Since there are however not many of these publications, our survey may be biased.

The selection criteria may not be appropriate for unambiguously finding all important publications. They may be too strict—e.g.we excluded schema matching which can arguably also be applied for schema evolution. To mitigate this issue, we clearly excluded publications which do not directly support the migration of elements in response to the evolution of the intensional definition. Moreover, the selection criteria may yield different results when applied by different people. To ensure their unambiguity, we conducted a pilot study as explained in Section 2.3.2. While the study confirmed the already high degree of unambiguity of the selection criteria, it also helped us to further improve them.

Nevertheless, although we carefully selected the scope, some of the topics outside the scope of this survey may be of interest to coupled evolution in the context of a conformance relation. In particular, we excluded schema matching (and schema integration) from the scope since in these approaches, schemas do not have to show clear resemblance. Yet, schema matching can thereby be considered to solve a more complex problem. This may yield different solutions, yet the solutions may still be of interest to coupled evolution. Additionally, we excluded the domain of ontology evolution, as it does not take element migration into account and API evolution as the extension is not completely defined by the intensional definition. Although approaches

in these domains may thereby not directly be applicable to coupled evolution of artefacts related by conformance, indirectly they could provide inspiration to new approaches. Further research is needed to include these additional domains into the survey and into the classification.

2.10.2 *Approach Classification*

When classifying approaches according to the feature model, we might misclassify approaches due to a number of reasons.

We might have built a feature model that exhibits non-orthogonal features within the same composite feature. To mitigate this threat, we checked correlations between the features contained in each composite feature. Since all these correlations have been strong negative, we concluded that the features are orthogonal to each other.

To prevent duplicated classifications, we decided to classify approaches and not publications. However, this may lead to an unbalance in representation, since an approach with one publication gets as much attention as approaches with more than one publication. To avoid this issue, we covered approaches with more than one publication more extensively in the text. Moreover, selected publications may either address multiple approaches (e.g.surveys), or may not address a concrete approach (e.g.empirical studies). We decided to not leave these publications out, but mentioned them in the "Additional Publications" section within the appropriate technological space.

The application of the feature model may yield different results, when applied by different people. To ensure the unambiguity of the features, we conducted a pilot study as explained in Section 2.4.4. The study showed that most of the approaches could be classified unambiguously, but also helped to remove a few unambiguities in the feature model.

2.10.3 *Interpretation*

The interpretations of the classification within or across technological spaces may be biased. To avoid the bias in the interpretations, we derived most of the interpretations using a systematic approach. We derived intra-space interpretations from the numbers of approaches for a certain feature within a technological space. Similarly, we derived cross-space interpretations by exploiting the feature model common to all technological spaces: we used common and uncommon features, as well as correlations between features.

2.11 CONCLUSION

In this chapter, we have reported on a systematic literature survey on coupled evolution in the technological spaces of dataware, grammarware, XMLware and modelware. While our initial focus was on eighteen conferences and eleven journals, exhaustive reference browsing to include publications from other venues yielded a research body that is comprised of 86 relevant publications of up to 40 years old.

Through a detailed reading of this research body, we identified 42 different approaches to coupled evolution and derived a feature model that is independent of technological spaces and was consequently used to characterize these approaches systematically. We have characterized the approaches on the basis of four main features: technological space, evolution, migration, and evaluation. The resulting classification is useful as a reference work for researchers in the field of coupled evolution, and helps them to identify both related work and avenues of future research in different technological spaces.

In advance, we posed five research questions pertaining to: the identification of a space-independent feature model; the application of the feature model to the various approaches, identification of feature and approach relations; and the distillation of directions of future research.

We identified a set of space-independent features and organized them in the feature model shown in Figure 2.5. We characterized the surveyed approaches along the feature model, shown in Figures 2.6 to 2.10. We identified positive and negative correlations between the features, provided by Figure 2.11. And finally, we provided an interpretation of approach classifications across technological space boundaries and pointed out avenues for future research in Section 2.9.

Based on this interpretation, we have learned three significant lessons:

First, we were able to identify space-independent categories of approaches, namely manual specification approaches, matching approaches, and operator-based approaches. Each of these three categories can be characterized by unique features from the feature model, namely a custom migration language, a declarative evolution specification, and an imperative evolution specification respectively. We identified two more categories which are still unique to the dataware space but can give rise to new approaches in the other technological spaces, namely versioning approaches and view-based approaches (Section 2.9.2).

Second, we observed that most approaches require the user to define the evolution. Though detection and recording of evolution are often mentioned as possible directions for future work, only few approaches actually follow these directions (Section 2.9.1). Considering most approaches recognize the need for detection or recording, it opens a much-needed direction of research.

Third, we have learned that most approaches lack significant evaluation. This holds particularly for the many approaches with user-defined evolu-

tions (Section 2.9.1 and 2.9.3). It implies a need for more thorough evaluation, but also a need for open case studies or benchmarks.

In summary, the work described in this chapter makes the following contributions:

- A selection of key publications on coupled evolution in different technological spaces, based on explicit selection criteria.
- An overview of coupled evolution approaches discussed in these publications.
- A feature model that can be used to characterize coupled evolution approaches from different technological spaces.
- An actual characterization of the presented approaches in terms of the features in this model.
- An intra-space interpretation of the features found in each technological space.
- An inter-space interpretation of these features.
- A series of recommendations on future research directions based on this interpretation.

ACKNOWLEDGMENTS

At the Technische Universität München, this research was funded by by the German Federal Ministry of Education and Research (BMBF), grants "SPES 2020, 01IS08045A" and "Quamoco, 01IS08023B". At Delft University of Technology, this research was supported by NWO/JACQUARD, project 638.001.610, MoDSE: Model-Driven Software Evolution. We thank Lennart Kats, Andreas Vogelsang and Stefan Wagner for providing feedback to improve this chapter.

A Catalog of Coupled Operators

ABSTRACT

Modeling languages and thus their metamodels are subject to change. When a metamodel evolves, existing models may no longer conform to it. Manual migration of these models in response to metamodel evolution is tedious and error-prone. To significantly automate model migration, operator-based approaches provide reusable coupled operators that encapsulate both metamodel evolution and model migration. The success of an operator-based approach highly depends on the library of reusable coupled operators it provides. In this chapter, we thus present an extensive catalog of coupled operators that is based both on a literature survey as well as real-life case studies. The catalog is organized according to a number of criteria to ease assessing the impact on models as well as selecting the right operator for a metamodel change at hand.

3

3.1 INTRODUCTION

Just as any other type of software, modeling languages are subject to evolution due to changing requirements and technological progress [Favre, 2005]. A modeling language is adapted to the changed requirements by evolving its metamodel. Due to *metamodel evolution*, existing models may no longer conform to the evolved metamodel and thus need to be migrated to reestablish conformance to the evolved metamodel. Avoiding *model migration* by downwards-compatible metamodel changes is often a poor solution, since it reduces the quality of the metamodel and thus the modeling language [Casais, 1995]. Manual migration of models is tedious and error-prone, and hence model migration needs to be automated. In *coupled evolution* of metamodels and models, the association of a model migration to a metamodel evolution is managed automatically. There are two major coupled evolution approaches: difference-based and operator-based approaches.

Difference-based approaches use a declarative evolution specification, generally referred to as difference model [Cicchetti et al., 2008, Garcés et al., 2009]. The difference model is mapped onto a model migration. The model migration can be specified declaratively as well as imperatively.

Operator-based approaches specify metamodel evolution by a sequence of operator applications [Wachsmuth, 2007b, Herrmannsdoerfer et al., 2009]. Each operator application can be coupled to a model migration separately. Operator-

based approaches generally provide a set of reusable coupled operators which work at the metamodel level as well as at the model level. At the metamodel level, a coupled operator defines a metamodel transformation capturing a common evolution. At the model level, a coupled operator defines a model transformation capturing the corresponding migration. Application of a coupled operator to a metamodel and a conforming model preserves model conformance.

In both operator-based and difference-based approaches, evolution can be specified manually [Wachsmuth, 2007b], can be recorded [Herrmannsdoerfer et al., 2009], or can be detected automatically, as discussed in Chapter 5. When recording, the user is restricted to a recording editor. Using automated detection, the building process can be completely automated, but may lead to an incorrect model migration.

In this chapter, we follow an operator-based approach to automate building a model migration for EMOF-like metamodels [Object Management Group, 2006].

Problem. The success of an operator-based approach highly depends on the library of reusable coupled operators it provides [Rose et al., 2009]. The library of an operator-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 evolution scenarios. However, the higher the number of coupled operators, the more difficult it is to find a coupled operator in the library. Consequently, a library should also be organized in a way that it is easy to select the right coupled operator for the change at hand.

Contribution. To provide guidance for building a library, we present an extensive catalog of coupled operators in this chapter. To ensure completeness, the coupled operators in this catalog are either motivated from the literature or from case studies that we performed. However, we do not target theoretical completeness—to capture all possible migrations—but rather practical completeness—to capture migrations that likely happen in practice. To ease usability, the catalog is organized according to a number of criteria. The criteria do not only allow to select the right coupled operator from the catalog, but also to assess the impact of the coupled operator on the modeling language and its models. For difference-based approaches, the catalog serves as a set of composite changes that such an approach needs to be able to handle.

Outline. The chapter is structured as follows: Section 3.2 presents the EMOF-like metamodeling formalism on which the set of coupled operators is based. Section 3.3 introduces the papers and case studies from which the coupled operators originate. Section 3.4 defines different classification criteria for coupled operators. Section 3.5 lists and groups the coupled operators of the catalog. Section 3.6 discusses the catalog, and Section 3.7 concludes the chapter.

3.2 METAMODELING FORMALISM

Metamodels can be expressed in various metamodeling formalisms. Well-known examples are the Meta Object Facility (MOF) [Object Management Group, 2006], the metamodeling standard proposed by the Object Management Group (OMG) and Ecore [Steinberg et al., 2009], the metamodeling formalism underlying the Eclipse Modeling Framework (EMF). In this chapter, we focus only on the core metamodeling constructs that are interesting for coupled evolution of metamodels and models. We leave out annotations, derived features, and operations, since these cannot be instantiated in models. An operator catalog will need additional operators addressing these metamodeling constructs in order to reach full compatibility with Ecore or MOF.

3.2.1 *Metamodel*

Figure 3.1 gives a textual definition of the metamodeling formalism used in this chapter. A metamodel is organized into *Package*s which can themselves be composed of *sub packages*. Each package defines a number of *Type*s which can be either primitive (*PrimitiveType*) or complex (*Class*). Primitive types are either *DataType*s like Boolean, Integer and String or *Enumeration*s of *literal*s. Classes consist of a number of *features*. They can have *super types* to inherit features and might be *abstract*, i.e. are not allowed to have objects. The *name* of a feature needs to be unique among all features of a class, including inherited ones. A *Feature* has a multiplicity (*lower bound* and *upper bound*) and is either an *Attribute* or a *Reference*. An attribute is a feature with a primitive *type*, whereas a reference is a feature with a complex *type*. An attribute can serve as an *id*entifier for objects of a class, i.e. the values of this attribute must be unique among all objects. A reference may be *composite* and two references can be combined to form a bidirectional association by making them *opposite* of each other.

3.2.2 *Model*

At the model level, instances of classes are called *objects*, instances of primitive data types are called *values*, instances of features are called *slots*, and instances of references are called *links*. The set of all links of composite references forms a containment structure, which needs to be tree-shaped and span all objects in a model.

3.2.3 *Notational Conventions*

Throughout the chapter, we use the textual notation from Figure 3.1 for metamodels. In this notation, features are represented by their name followed by a separator, their type, and an optional multiplicity. The separator indicates the

```
abstract class NamedElement {
   name        :: String (1..1)
}

class Package : NamedElement {
   subPackages ◊
      Package (0..*)
   types       ◊
      Type (0..*)
}

abstract class Type : NamedElement
{}

abstract class PrimitiveType : Type
{}

class DataType : PrimitiveType
{}

class Enumeration : PrimitiveType {
   literals    ◊
      Literal (0..*)
}

class Literal : NamedElement
{}

class Class : Type {
   isAbstract  :: Boolean
   superTypes  → Class (0..*)
   features    ◊
      Feature (0..*)
}

abstract class Feature
 : NamedElement {
   lowerBound  :: Integer
   upperBound  :: Integer
   type        → Type
}

class Attribute : Feature {
   isId        :: Boolean
}

class Reference : Feature {
   isComposite :: Boolean
   opposite    → Reference
}
```

Figure 3.1 Metamodeling formalism providing core metamodeling concepts. The used integers are signed integers, such that -1 can refer to a missing lower or upper bound.

kind of a feature. We use :: for attributes, → for ordinary references, and ◊ for composite references.

3.3 ORIGINS OF COUPLED OPERATORS

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

3.3.1 *Literature*

First, coupled operators originate from the literature on the evolution of metamodels as well as object-oriented database schemas and code.

Wachsmuth [2007b] first proposes an operator-based approach for *metamodel* evolution and classifies a set of operators according to the preservation of metamodel expressiveness and existing models. Gruschko et al. envision a difference-based 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. [2008] list a set of composite changes which they are able to detect using their difference-based approach.

Banerjee et al. [1987b] 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. Brèche [1996] introduces a set of high-level operators for schema evolution in the object-oriented system O_2 and shows how to implement them in terms of primitive operators. Pons and Keller [1997] propose a three-level catalog of operators for object-oriented schema evolution which groups operators according to their complexity. Claypool et al. [2000] list a number of primitives for the adaptation of relationships in object-oriented systems.

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

3.3.2 *Case Studies*

Second, coupled operators originate from a number of case studies that we have performed. Figure 3.2 gives an overview of these case studies. It mentions the tool that was used in a case study, the name of the evolving metamodel, and whether the evolution was obtained in a forward or reverse engineering process. In forward engineering, the tool is used to aid and possibly record evolution as it happens, whereas in reverse engineering, the tool is used to reconstruct evolution after it occurred. To provide evidence that the case studies are considerable in size, the table also shows the number of different kinds of metamodel elements at the end of the evolution as well as the number of operator applications needed to perform the evolution.

Herrmannsdoerfer et al. [2008] performed a case study on the evolution of two industrial metamodels to show that most of the changes can be captured by reusable coupled operators: 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.

Based on the requirements derived from this study, Herrmannsdoerfer et al. [2009] implemented the operator-based tool *COPE*[1] which records change histories on metamodels of the Eclipse Modeling Framework (EMF). To demonstrate its applicability, COPE has been used to reverse engineer the operator history of a number of metamodels: Palladio Component Model (PCM) for the specification of software architectures [Herrmannsdoerfer et al., 2009] and Graphical Modeling Framework (GMF) for the model-based development of diagram editors [Herrmannsdoerfer et al., 2010a]. Currently, COPE is applied

[1] COPE web site, `http://cope.in.tum.de`

Tool	Case	Kind	Packages	Classes	Attributes	References	Data Types	Enumerations	Literals	Operator Applications
[H08]	FLUID	reverse	8	155	95	155	0	1	10	223
	TAF-Gen		15	97	81	114	1	13	76	134
COPE	PCM [H09]	reverse	19	99	18	135	0	4	19	101
	GMF [H10a]		4	252	379	278	0	27	166	737
	Unicase	forward	17	77	88	161	0	11	49	58
	Quamoco		1	22	14	35	0	1	2	423
Acoda	BugZilla	reverse	–	51	208	64	–	–	–	237
	Researchr		–	125	380	278	–	6	31	64
	YellowGrass	forward	–	12	33	21	–	0	0	30

Figure 3.2 Statistics for case studies. [H08] abbreviates [Herrmannsdoerfer et al., 2008]; [H09] abbreviates [Herrmannsdoerfer et al., 2009]; and [H10a] abbreviates [Herrmannsdoerfer et al., 2010a]

to forward engineer the operator history of a number of metamodels: Unicase[2] for UML modeling and project management and Quamoco[3] for modeling the quality of software products.

We implemented the operator-based tool *Acoda*[4] (Chapters 4 and 5), which detects operator histories on object-oriented data models. To demonstrate its applicability, Acoda has been used to reverse engineer the operator history of the data model behind BugZilla which is a well-known tool for bug tracking and the operator history behind Researchr[5], a web application for maintaining scientific publication meta data. Currently, Acoda is applied to forward engineer the operator-based evolution of YellowGrass[6], a web application for tag-based issue tracking.

The crossed-out cells in Figure 3.2 indicate that the metamodeling constructs are currently not supported by the used data modeling formalism.

[2]Unicase web site, `http://unicase.org`
[3]Quamoco web site, `http://www.quamoco.de`
[4]Acoda web site, `http://swerl.tudelft.nl/bin/view/Acoda`
[5]Researchr web site, `http://researchr.org`
[6]YellowGrass web site, `http://yellowgrass.org`

3.4 CLASSIFICATION OF COUPLED OPERATORS

Coupled operators 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, 2007b].

3.4.1 *Language Preservation*

A metamodel is an intensional definition of a language. Its extension is a set of conforming models. When an operator is applied to a metamodel, this has an impact on its extension and thus on the expressiveness of the language. We distinguish different classes of operators according to this impact [Lämmel, 2001, Wachsmuth, 2007b]: An operator is a *refactoring* if there exists always a bijective mapping between extensions of the original and the evolved metamodel. An operator is a *constructor* if there exists always an injective mapping from the extension of the original metamodel to the extension of the evolved metamodel. An operator is a *destructor* if there exists always a surjective mapping from the extension of the original metamodel to the extension of the evolved metamodel.

3.4.2 *Model Preservation*

Model preservation properties indicate when migration is needed. An operator is *model-preserving* if all models conforming to an original metamodel also conform to the evolved metamodel. Thus, model-preserving operators do not require migration. An operator is *model-migrating* if models conforming to an original metamodel might need to be migrated in order to conform to the evolved metamodel. It is *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). In contrast, an *unsafely model-migrating* operator might yield the same model when migrating two different models.

Classification of operators w.r.t. model preservation is related to the classification with respect to language preservation: Refactorings and constructors are either model-preserving or safely model-migrating operators. Destructors are unsafely model-migrating operators. Furthermore, the classification is related to a classification of changes known from difference-based approaches [Becker et al., 2007, Burger and Gruschko, 2010]: model-preserving operators perform *non-breaking changes*, whereas model-migrating operators perform *breaking, resolvable changes*. However, there is no correspondence for *breaking, non-resolvable changes*, since coupled operators always provide a migration to resolve the breaking change.

3.4.3 *Bidirectionality*

Another property we are interested in is the reversibility of evolution. Bidirectionality properties indicate that an operator can be safely undone on the language or model level. An operator is *self-inverse* iff a second application of the operator—possibly with other parameters—always yields the original metamodel. An operator is the *inverse* of another operator iff there is always a sequential composition of both operators which is a refactoring. Finally, an operator is a *safe inverse* of another operator iff there is always a sequential composition of both operators which is model-preserving.

3.5 CATALOG OF COUPLED OPERATORS

Metamodel adaptation requires migration of instances to reestablish conformance. Similar metamodel adaptations frequently require similar instance migrations. Metamodel adaptation operators can thereby be coupled to an instance migration operator. Such coupled operators ensure instance conformance on metamodel adaptation. Similar metamodel operators may require different types of instance migration operators and may thereby occur in different coupled operators. Coupled operators prove reusable in practice.

In this section, we present a catalog of 61 coupled operators that we consider complete for practical application. As discussed in Section 3.3, we included all coupled operators found in nine related papers as well as all coupled operators identified by performing nine real-life case studies. In the following, we explain the coupled operators in groups which help users to navigate the catalog. We start with *primitive* operators 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 *complex* operators. These can be decomposed into a sequence of primitive operators which has the same effect at the metamodel level but not neccessarily at the model level. We group complex operators according to the metamodeling techniques they address—distinguishing specialization and generalization, delegation, and inheritance operators—as well as their semantics—distinguishing replacement, and merge and split operators.

Each group is discussed separately in the subsequent sections. For each group, a table provides an overview over all operators in the group. Using the classifications from Section 3.4, the table classifies each coupled operator 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 operator by referring to its number.

#	Operator Name	Class.			MM			OODB				OOC		Ho8		COPE				Acoda		
		language preservation	model preservation	inverse	[Wachsmuth, 2007b]	[Becker et al., 2007]	[Cicchetti et al., 2008]	[Banerjee et al., 1987b]	[Brèche, 1996]	[Pons and Keller, 1997]	[Claypool et al., 2000]	[Fowler, 1999]	[Dig and Johnson, 2006]	FLUID	TAF-Gen	[Herrmannsdoerfer et al., 2009]	[Herrmannsdoerfer et al., 2010a]	Unicase	Quamoco	BugZilla	Researchr	YellowGrass
1	Create Package	r	p	2s		x									x	x						
2	Delete Package	r	p	1s		x									x							
3	Create Class	c	p	4s	x	x	x	x						x	x	x	x	x	x	x	x	x
4	Delete Class	d	u	3u	x	x	x	x	x				x	x		x		x	x	x	x	
5	Create Attribute	c	s	7s	x	x	x	x						x	x	x	x	x	x	x	x	x
6	Create Reference	c	s	7s	x	x	x	x						x	x	x	x	x	x	x	x	x
7	Delete Feature	d	u	5/6u	x	x	x	x						x	x	x	x	x	x	x	x	
8	Create Opposite Ref.	d	u	9u		x					x	x					x	x	x		x	x
9	Delete Opposite Ref.	c	p	8s		x					x	x			x							
10	Create Data Type	r	p	11s		x																
11	Delete Data Type	r	p	10s		x													x			
12	Create Enum	r	p	13s		x									x	x	x	x	x			
13	Delete Enum	r	p	11s		x												x				
14	Create Literal	c	p	15s		x												x				
15	Merge Literal	d	u	14u		x										x						

Figure 3.3 Structural Primitives

Finally, each paper and case study has a column in each table. An x in such a column denotes occurrence of the operator in the corresponding paper or case study. Papers are referred to by citation, while case studies are referred to by the name of the case. For each coupled operator, we discuss its semantics in terms of metamodel evolution and model migration.

3.5.1 *Structural Primitives*

Structural primitive operators (Figure 3.3) modify the structure of a metamodel, i.e. create or delete metamodel elements. Creation operators are parameterized by the specification of a new metamodel element, and deletion operators 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 slots using default values or default value computations.

Deleting metamodel elements, such as classes and references, 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 operators 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 operators discussed in subsequent sections 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 operators which may have been instantiated in the model (with the exception of *Delete Opposite Reference*) are unsafely model-migrating due to loss of information. Deletion provides a safe inverse to its associated creation operator. 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.

Creating and deleting references which have an opposite are different from other creation and deletion operators. *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.

Create / Delete Data Type and *Create / Delete Enumeration* are refactorings, as restrictions on these operators 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. In migration, occurrences of merged literals are replaced by the single target literal. Merging a literal provides a safe inverse to *Create Literal*.

A number of literals `l1`, ..., `ln` defined in the same enumeration can be merged into a single literal `l1`. In migration, these literals are all replaced by `l1`. Merging literals provides a safe inverse to creating literals. Vice versa provides an unsafe inverse.

3.5.2 *Non-structural Primitives*

Non-structural primitive operators (Figure 3.4) modify a single, existing metamodel element, i.e. change properties of a metamodel element. All non-structural operators take the affected metamodel element, their subject, as parameter.

#	Operator Name	Class.			MM			OODB					OOC		Ho8		COPE				Acoda		
		language preservation	model preservation	inverse	[Wachsmuth, 2007b]	[Becker et al., 2007]	[Cicchetti et al., 2008]	[Banerjee et al., 1987b]	[Brèche, 1996]	[Pons and Keller, 1997]	[Claypool et al., 2000]	[Fowler, 1999]	[Dig and Johnson, 2006]	FLUID	TAF-Gen	[Herrmannsdoerfer et al., 2009]	[Herrmannsdoerfer et al., 2010a]	Unicase	Quamoco	BugZilla	Researchr	YellowGrass
1	Rename	r	s	1s	x	x	x	x				x	x	x	x	x	x	x	x	x	x	x
2	Change Package	r	s	2s		x								x	x	x		x				
3	Make Class Abstract	d	u	4u		x								x		x			x			
4	Drop Class Abstract	c	p	3s		x											x		x			
5	Add Super Type	c	p	6s		x		x							x	x	x	x	x		x	x
6	Remove Super Type	d	u	5u		x		x							x	x	x	x	x			
7	Make Attr. Identifier	d	u	8u		x								x							x	
8	Drop Attr. Identifier	c	p	7s		x								x			x					
9	Make Ref. Composite	d	u	10u		x						x		x	x		x		x			
10	Switch Ref. Composite	c	s	9s		x		x				x		x	x				x			
11	Make Ref. Opposite	d	u	12u		x					x				x						x	x
12	Drop Ref. Opposite	c	p	11s		x					x						x		x			

Figure 3.4 Non-structural Primitives

Change Package can be applied to both package and type. Additionally, the value-changing operators *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.

Packages, types, features and literals can be renamed. *Rename* is safely model-migrating (when parametrized by an unused name) 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 super class 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 Super Type* `S` from a class `C` implies removing slots of features inher-

ited from S. Additionally, references targeting type S, referring to objects of type C, need to be removed. To prevent breaking multiplicity restrictions, *Remove Super Type* is restricted to types S which are not targeted by mandatory references—neither directly, nor through inheritance. The operator is unsafely model-migrating and can be unsafely inverted by *Add Super Type*. A special type of super type declaration removal is removing a superfluous super type declaration. A super type declaration is superfluous when its features are already inherited through other super type declarations. Superfluous super type removal is strictly instance preserving and language preserving. It has a safe inverse in adding the removed declaration.

Attributes defined as identifier need to be unique. *Make Attribute Identifier* requires a migration which ensures uniqueness of the attribute's values and is thus unsafely model-migrating. *Drop Attribute Identifier* is model-preserving and does not require migration.

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 link set satisfy the added multiplicity restriction. The operator 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 operator, reference b is no longer composite. *Switch Reference Composite* provides an unsafe inverse to *Make Reference Composite*.

3.5.3 *Specialization / Generalization Operators*

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 (Figure 3.5). All specialization and generalization operators 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).

#	Operator Name	Class.			MM			OODB				OOC		Ho8		COPE				Acoda		
		language preservation	model preservation	inverse	[Wachsmuth, 2007b]	[Becker et al., 2007]	[Cicchetti et al., 2008]	[Banerjee et al., 1987b]	[Brèche, 1996]	[Pons and Keller, 1997]	[Claypool et al., 2000]	[Fowler, 1999]	[Dig and Johnson, 2006]	FLUID	TAF-Gen	[Herrmannsdoerfer et al., 2009]	[Herrmannsdoerfer et al., 2010a]	Unicase	Quamoco	BugZilla	Researchr	YellowGrass
1	Generalize Attribute	c	p	2s	x	x	x							x	x	x	x		x	x	x	x
2	Specialize Attribute	d	u	1u	x	x	x							x		x	x		x			
3	Generalize Reference	c	p	4s	x	x	x							x	x		x		x		x	x
4	Specialize Reference	d	u	3u	x	x	x							x	x	x	x		x			
5	Specialize Comp. Ref.	d	u	3u										x		x			x			
6	Generalize Super Type	d	u	7u		x										x					x	
7	Specialize Super Type	c	s	6s		x				x				x	x	x	x		x			

Figure 3.5 Specialization / Generalization Operators

Generalization of feature types 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 slots 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 safe inverse of super type specialization which is an unsafe inverse vice versa.

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 the new attribute type. Specializing type conversions are surjective. *Generalize Attribute* extends the attribute's multiplicity or generalizes the attribute's type. Generalizing 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.

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.

Super type declarations are commonly adapted, while refining a metamodel. Consider the following example, in which classes `A`, `B` and `C` are part of a linear inheritance structure and remain unadapted:

```
class A { }
class B : A {
   f :: Integer (1..1)
}

class C : A
{ }
```

```
class A {}

class B : A {
   f :: Integer (1..1)
}

class C : B
{ }
```

From left to right, *Specialize Super Type* 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 creation of slots by the migration. In general, super type specialization requires addition of feature slots which are declared mandatory by the new super type. From right to left, *Generalize Super Type* 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`. Slots 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.

3.5.4 *Inheritance Operators*

Inheritance operators (Figure 3.6) move features along the inheritance hierarchy. Most of them are well-known from refactoring object-oriented code [Fowler, 1999]. 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.

Pull up Feature is a constructor which moves a feature that occurs in all subclasses of a class to the class itself. The operator is a constructor since instances of the class can now convey additional information. It is in general instance-preserving modulo variation. For migration, slots for the pulled up

# Operator Name	Class.			MM			OODB				OOC		Ho8		COPE				Acoda		
	language preservation	model preservation	inverse	[Wachsmuth, 2007b]	[Becker et al., 2007]	[Cicchetti et al., 2008]	[Banerjee et al., 1987b]	[Brèche, 1996]	[Pons and Keller, 1997]	[Claypool et al., 2000]	[Fowler, 1999]	[Dig and Johnson, 2006]	FLUID	TAF-Gen	[Herrmannsdoerfer et al., 2009]	[Herrmannsdoerfer et al., 2010a]	Unicase	Quamoco	BugZilla	Researchr	YellowGrass
1 Pull up Feature	c	p	2s	x		x					x	x	x	x		x		x			
2 Push down Feature	d	u	1u	x		x					x	x		x		x					
3 Extract Super Class	c	p	4s	x		x		x	x		x	x	x	x		x		x		x	x
4 Inline Super Class	d	u	3u	x		x			x		x	x		x		x	x	x			
5 Fold Super Class	c	s	6s										x					x			
6 Unfold Super Class	d	u	5u													x		x			
7 Extract Sub Class	c	s	8s					x	x		x					x		x			
8 Inline Sub Class	d	u	7u						x		x			x				x			

Figure 3.6 Inheritance Operators

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. It is the safe inverse of *Pull up Feature* while the latter is only a unsafe inverse of the first. This makes it a destructor since instances of the class can convey less information. It is only partially instance-preserving. While objects of the subclasses stay unaltered, slots for the original feature must be removed from objects of the class itself. As an alternative, objects of the class might be converted into objects of subclasses.

Extract Super Class is a constructor which introduces a new class, makes it the super class of a set of classes, and pulls up one or more features from these classes. In general, it is a constructor because the new class adds expressiveness to the language. It is strictly instance-preserving. The corresponding destructor *Inline Super Class* is the safe inverse for it. 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 super class in a previous step. The operator is a destructor since the removed class restricts expressiveness of the language. It is partially instance-preserving modulo variation. Objects of the class need to be migrated to objects of the subclasses. This might require the addition of slots for features of the subclasses. In general, this migration is irreversible which makes *Extract Super Class* only an unsafe inverse of *Inline Super Class*.

The constructor *Fold Super Class* is related to *Extract Super Class*. Here, the

new super class 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 operator then removes these features and adds instead an inheritance relation to the intended super class. In the same way, the destructor *Unfold Super Class* is related to *Inline Super Class*. This operator copies all features of a super class into a subclass and removes the inheritance relation between both classes. Here is an example for both operators:

```
class A {
   f1 :: Integer
}

class B : A {
   f2 :: Integer
}

class C {
   f1 :: Integer
   f2 :: Integer
   f3 :: Integer
}
```

```
class A {
   f1 :: Integer
}

class B : A {
   f2 :: Integer
}

class C : B {
   f3 :: Integer
}
```

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

The constructor *Extract Subclass* introduces a new class, makes it the subclass of another, and pushes down one or more features from this class. In general, it is a constructor because the new class adds expressiveness to the language. Objects of the original class must be converted to objects of the new class, in order not to loose instantiations of pushed down features. The corresponding destructor *Inline Subclass* pulls up all features from a subclass into its non-abstract super class and deletes the subclass afterwards. References in other classes pointing to the class are not allowed but can be generalized to a super class in a previous step. The operator is a destructor since the removed class restricts expressiveness of the language. Objects of the subclass need to be migrated to objects of the super class. *Extract Subclass* is an unsafe inverse of *Inline Subclass*.

3.5.5 *Delegation Operators*

Delegation operators (Figure 3.7) 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 Class moves features to a new delegate class and adds a composite reference to the new class together with an opposite reference. It takes the original class, the set of features, the name for the delegate class, and the name

# Operator Name	Class.			MM			OODB				OOC		Ho8		COPE				Acoda		
	language preservation	model preservation	inverse	[Wachsmuth, 2007b]	[Becker et al., 2007]	[Cicchetti et al., 2008]	[Banerjee et al., 1987b]	[Brèche, 1996]	[Pons and Keller, 1997]	[Claypool et al., 2000]	[Fowler, 1999]	[Dig and Johnson, 2006]	FLUID	TAF-Gen	[Herrmannsdoerfer et al., 2009]	[Herrmannsdoerfer et al., 2010a]	Unicase	Quamoco	BugZilla	Researchr	YellowGrass
1 Extract Class	r	s	2s	x		x		x	x		x	x	x	x		x		x		x	x
2 Inline Class	r	s	1s	x		x			x		x	x		x				x		x	
3 Fold Class	r	s	4s										x	x	x						
4 Unfold Class	r	s	3s																		
5 Move Feature over Ref.	c	s	6s	x		x			x		x	x		x		x			x	x	
6 Collect Feature over Ref.	d	u	5u													x		x			

Figure 3.7 Delegation Operators

for the containment reference as arguments. This refactoring is language-preserving and instance-preserving modulo variation. During migration, an object of the delegate class is created for each object of the original class, slots for the moved features are moved to the new delegate object, and a link to the delegate object is created. The corresponding *Inline Class* is a safe inverse for *Extract Class* and vice versa. Like its counterpart, it is a refactoring which is model-preserving modulo migration. On the metamodel level, it 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, slots 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.

Fold and *Unfold Class* are quite similar to *Extract* and *Inline Class*. Both operators are model-preserving modulo migration as well as safe inverses of each other. The only difference is that the delegate class exists already and thus is not created or deleted. The names of these operators have been adopted from grammar adaptation [Lämmel, 2001]. The following example illustrates the difference:

```
class A {
   a1 :: Integer
   a2 :: Boolean
   r1 → B (1..1)
   r2 → B (0..*)
}

class B
{ }

class C {
   a1 :: Integer
   r1 → B (1..1)
}
```

```
class A {
   c ◇ C (1..1)
   d ◇ D (1..1) opposite a
}

class B
{ }

class C {
   a1 :: Integer
   r1 → B (1..1)
}

class D {
   a2 :: Boolean
   r2 → B (0..*)
   a  → A (1..1) opposite d
}
```

From left to right, the features `a1` and `r1` of class `A` are folded to a composite reference `A.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 `A.c` is unfolded which keeps `C` intact while `A.d` is inlined which removes `D`.

Move Feature along Reference is a constructor which moves a feature over a single-valued reference to a target class. This operator is a constructor since objects of the target class which are not referenced by instances of the source class can now convey additional information. It is instance-preserving modulo variation. 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, slots with default values must be added. The destructor *Collect Feature over Reference* is a safe inverse of the last operator. 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, or remains such if it already was. 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, slots of the original feature are removed. Here is an example for both operators:

```
class A {
   f1 :: Integer (1..*)
   r1 → B (1..1)
   r2 → C (0..*)
}

class B
{ }

class C {
   f2 :: Integer (1..1)
}
```

```
class A {
   f2 :: Integer (0..*)
   r1 → B (1..1)
   r2 → C (0..*)
}

class B {
   f1 :: Integer (1..*)
}

class C
{ }
```

# Operator Name	Class.			MM			OODB					OOC		Ho8	COPE				Acoda		
	language preservation	model preservation	inverse	[Wachsmuth, 2007b]	[Becker et al., 2007]	[Cicchetti et al., 2008]	[Banerjee et al., 1987b]	[Brèche, 1996]	[Pons and Keller, 1997]	[Claypool et al., 2000]	[Fowler, 1999]	[Dig and Johnson, 2006]	FLUID	TAF-Gen	[Herrmannsdoerfer et al., 2009]	[Herrmannsdoerfer et al., 2010a]	Unicase	Quamoco	BugZilla	Researchr	YellowGrass
1 Subclasses to Enum.	r	s	2s								x										
2 Enum. to Subclasses	r	s	1s								x						x				
3 Reference to Class	r	s	4s	x										x				x			
4 Class to Reference	r	s	3s	x									x								
5 Inheritance to Delegation	r	s	6s						x		x		x	x		x					
6 Delegation to Inheritance	r	s	5s						x		x										
7 Reference to Identifier	c	s	8s										x								
8 Identifier to Reference	d	u	7u										x			x		x		x	

Figure 3.8 Replacement Operators

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

3.5.6 *Replacement Operators*

Replacement operators (Figure 3.8) replace one metamodeling construct by another, equivalent construct. Thus replacement operators typically are refactorings and safely model-migrating. With the exception of the last two operators, an operator to replace the first construct by a second always comes with a safe inverse to replace the second by the first, and vice versa.

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 super class, 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. The following example demonstrates both directions:

```
class C {
   ...
}

class S1 : C
{ }

class S2 : C
{ }
```

```
class C {
   e :: E
   ...
}

enum E {
   s1, s2
}
```

From left to right, *Subclasses to Enumeration* replaces the subclasses `S1` and `S2` of class `C` by the new attribute `C.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 `C.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`.

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. The following example demonstrates both directions:

```
class S {
   r → T (1..*)
   ...
}
```

```
class S {
   r ◇ R (1..*)
   ...
}

class R {
   s → S (1..1) opposite r
   t → T (1..1)
}
```

From left to right, *Reference to Class* retargets the reference `S.r` to a new reference class `R`. Source and target of the original reference can be accessed via references `R.s` and `R.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 `S.r` directly to the target class `T`.

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 slots of the features inherited from the superclass are extracted to a separate object of the super class. 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. The following example demonstrates both directions:

```
class C : S {
   ...
}
```

```
class C {
   s ◇ S (1..1)
   ...
}
```

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 slots of the features inherited from the super class `S` are extracted to a separate object of the super class. From right to left, *Delegation to Inheritance* removes the reference `C.s` and makes `S` a super class of `C`.

To decouple a reference, it can be replaced by an indirect reference using an identifier, and vice versa. *Reference to Identifier* deletes the reference and creates an attribute in the source class whose value refers to an id 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 operators.

3.5.7 *Merge / Split Operators*

Merge operators (Figure 3.9) merge several metamodel elements of the same type into a single element, whereas split operators split a metamodel element into several elements of the same type. Consequently, merge operators typically are destructors and split operators constructors. In general, each merge operator has an inverse split operator. Split operators are more difficult to define, as they may require metamodel-specific information about how to split values. There are different merge and split operators for the different metamodeling constructs.

# Operator Name	Class.			MM			OODB				OOC		Ho8	COPE				Acoda		
	language preservation	model preservation	inverse	[Wachsmuth, 2007b]	[Becker et al., 2007]	[Cicchetti et al., 2008]	[Banerjee et al., 1987b]	[Brèche, 1996]	[Pons and Keller, 1997]	[Claypool et al., 2000]	[Fowler, 1999]	[Dig and Johnson, 2006]	FLUID	TAF-Gen	[Herrmannsdoerfer et al., 2009]	[Herrmannsdoerfer et al., 2010a]	Unicase	Quamoco	BugZilla	Researchr	YellowGrass
1 Merge Features	d	u							x				x					x			
2 Split Reference by Type	r	s	1s															x			
3 Merge Classes	d	u	4u					x	x				x		x	x		x			
4 Split Class	c	p	3s																		
5 Merge Enumerations	d	u														x					

Figure 3.9 Merge / Split Operators

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 *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 instantiating the original reference is moved to the corresponding target reference according to its type. If we require that the type of the reference is abstract, this operator is a refactoring and has *Merge Features* as a safe inverse.

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 operator is a destructor. In a model, objects of the sibling classes are migrated to the new class. *Split Class* is an unsafe 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.

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 operator 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.

3.6 DISCUSSION

3.6.1 *Completeness*

At the metamodel level, an operator catalog is complete if any source metamodel can be evolved to any target metamodel. This kind of completeness is achieved by the catalog presented in the chapter. An extreme strategy would be the following [Banerjee et al., 1987b]: 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.

Completeness is much harder to achieve, when we take the model level into account. Here, an operator catalog 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 catalog needs to provide a full-fledged model transformation language based on operators. A first useful restriction is Turing completeness. But reaching for this kind of completeness comes at the price of usability. Given an existing operator, one can always think of a slightly different operator having the same effect on the metamodel level but a slightly different migration. But the higher the number of coupled operators, the more difficult it is to find an operator in the catalog. And with many similar operators, 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. When we started our case studies, we found the set of operators from the literature rather incomplete. For each case study, we added frequently reoccurring operators to the catalog. The number of operators we added to the catalog declined with every new case study, thus approaching a stable catalog. Our latest studies revealed no new operators. Although, most case studies showed some operators which were only applied once. They were never reused in other case studies. Therefore, we decided not to include them in the catalog.

We expect similar special cases in practical applications where only a few evolution steps can not be modeled by operators from the catalog. These

cases can effectively be handled by providing a means for overwriting a coupling [Herrmannsdoerfer et al., 2009]: The user can specify metamodel evolution by an operator application but overwrites the model migration for this particular application. This way, theoretical completeness can still be achieved.

3.6.2 *Metamodeling Formalism*

In this chapter, we focus only on core metamodeling constructs that are interesting for coupled evolution of metamodels and models. But 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 metamodeling constructs like interfaces, operations, derived features, volatile features, or annotations. An operator catalog will need additional operators addressing these metamodeling constructs to reach full compatibility with Ecore or MOF. Such additional operators are relevant for practical completeness.

In the GMF case study [Herrmannsdoerfer et al., 2010a], we found 25% of the applied operators to address changes in the API. Most of these operators do not require migration. The only exceptions were annotations containing constraints. An operator catalog accounting for constraints needs to deal with two kinds of adaptations: First, the constraints need to be co-evolved when the metamodel evolves. Operators need to provide this co-evolution in addition to model migration. Second, evolving constraints might invalidate existing models and thus require model migration. Here, new coupled operators for the evolution of constraints are needed.

Things become more complicated when it comes to CMOF [Object Management Group, 2006]. Concepts like package merge, feature subsetting, and visibility affect the semantics of operators in the chapter and additional operators are needed to deal with these concepts. For example, we would need four different kinds of *Rename* due to the package merge: 1) Renaming an element which is not 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*).

3.6.3 *Tool Support*

In operator-based tools, operators are usually made available to the user through an operator browser [Wachsmuth, 2007a, Herrmannsdoerfer et al., 2009]. Here, the organization of the catalog into groups can help to find an operator for a change at hand. The preservation properties can be used to

reason about the impact on language expressiveness and on existing models. In grammarware, similar operators have been successfully used in [Lämmel and Zaytsev, 2009b] to reason about relationships between different versions of the Java grammar. To make the user aware of the impact on models, it can be shown by a traffic light in the browser: green for model-preserving, yellow for safely and red for unsafely model-migrating. Additionally, the operator browser may have different modes for restricting the presented operators in order to guarantee language- and/or model-preservation properties. Bidirectionality can be used to invert an evolution that has been specified erroneously earlier. Recorded operator applications can be automatically undone with different levels of safety by applying the inverse operators. Tools that support evolution detection should evade of destructors in favor of refactorings to increase the preservation of information by the detected evolution.

Difference-based tools [Cicchetti et al., 2008, Garcés et al., 2009] need to be able to specify the mappings underlying the operators from the catalog. When they allow to specify complex mappings, they could introduce means to specify the mappings of the operators in a straightforward manner. Introducing such first class constructs reduces the effort for specifying the migration. For instance, the declarative language presented by Narayanan et al. [2009] provides patterns to specify recurrent mappings.

3.7 CONCLUSION

We presented a catalog of 61 operators for the coupled evolution of metamodels and models. These so-called coupled operators evolve a metamodel and in response are able to automatically migrate existing models. The catalog covers not only well-known operators from the literature, but also operators which have proven useful in a number of case studies we performed. The catalog is based on the widely used EMOF metamodeling formalism [Object Management Group, 2006] which was stripped of the constructs that cannot be instantiated in models. When a new construct is added to the metamodeling formalism, new operators have to be added to the catalog: Primitive operators to create, delete and modify the construct as well as complex operators to perform more intricate evolutions involving the construct. The catalog not only serves as a basis for operator-based tools, but also for difference-based tools. Operator-based tools need to provide an implementation of the presented operators. Difference-based tools need to be able to specify the mappings underlying the presented operators.

ACKNOWLEDGMENTS

At the Technische Universität München, this research was funded by by the German Federal Ministry of Education and Research (BMBF), grants "SPES 2020, 01IS08045A" and "Quamoco, 01IS08023B". At Delft University of Technology, this research was supported by NWO/JACQUARD, project 638.001.610, MoDSE: Model-Driven Software Evolution. We thank Lennart Kats, Andreas Vogelsang and Stefan Wagner for providing feedback to improve this chapter.

Generating Database Migrations for Evolving Web Applications

4

ABSTRACT

WebDSL is a domain-specific language for the implementation of dynamic web applications with a rich data model. It provides developers with object-oriented data modeling concepts but abstracts over implementation details for persisting application data in relational databases. When the underlying data model of an application evolves, persisted application data has to be migrated. While implementing migration at the database level breaks the abstractions provided by WebDSL, an implementation at the data model level requires to intermingle migration with application code. In this chapter, we present a domain-specific language for the coupled evolution of data models and application data. It allows to specify data model evolution as a separate concern at the data model level and can be compiled to migration code at the database level. Its linguistic integration with WebDSL enables static checks for evolution validity and correctness.

4.1 INTRODUCTION

WebDSL is a domain-specific language for the implementation of dynamic web applications with a rich data model [Visser, 2008a]. It provides developers with object-oriented data modeling concepts. These concepts abstract over implementation details for persistence. These details are added in a two-step compilation process. In the first step, the WebDSL compiler generates application code in an object-oriented general purpose programming language, which is Java. To achieve persistence, the generated code relies on the Hibernate framework. This framework realizes an object-relational mapping (ORM): Application data is kept in objects at runtime but is persisted in a relational database. In the second step, the generated application code is compiled and the persistence framework generates a relational database schema. When deploying the application, a relational database management system (RDBMS) generates an initial, empty database from this schema. The deployed application will interact with the RDMBS to store and to retrieve its data.

PROBLEM. As any other software, web applications and their data models evolve. An evolved application has to be recompiled and redeployed. During recompilation, the persistence framework generates a new database schema. Typically, the original database no longer complies with the new schema and original application data cannot be accessed from the evolved application anymore. During redeployment, the RDBMS instead generates a new initial database from the new schema. But original application data is a valuable asset. It needs to be migrated to co-evolve with the application and its data model.

Implementing migrations at the database level breaks the abstractions provided by WebDSL. Developers have to be aware of the persistence framework and its ORM to make sure that the migrated database complies with the new schema. They also have to be aware of the RDBMS to provide details such as character set definitions, collations, and storage engines.

To avoid breaking abstractions, migrations can be implemented at the data model level in WebDSL. Since the generated code will make extensive usage of the ORM, migration does not scale to large amounts of data and is typically performed lazily. The application migrates original data only when it needs to access this data. As a consequence, the original data model has to remain part of the evolving data model and application code is intermingled with migration code. Maintenance of data model, application code, and migration code becomes harder with every new evolution step.

CONTRIBUTION. In Chapter 3, we compiled an extensive catalog of coupled operators for the evolution of object-oriented data models. These operators couple common evolution steps at the data model level with their corresponding migrations at the data level. In this chapter, we focus on the implementation of these operators in Acoda, a tool for the coupled evolution of WebDSL data models and databases.

Acoda provides a domain-specific language for specifying data model evolution as a separate concern at the data model level. Its IDE offers static checks for evolution validity and correctness. While evolution validity ensures that an evolution can be applied to the original data model, evolution correctness secures that the evolution yields the evolved data model.

Acoda implements coupled operators as a mapping from evolution steps to migration code in SQL. In this chapter, we discuss this mapping for particular operators in detail, including complex operators that work along the inheritance hierarchy or over references. Thereby, we distinguish three kinds of migrations. First, *schema modifications* change only the database schema. Second, *conservative migrations* rearrange data without data loss. Third, *lossy migration* supports potential data loss on purpose.

OUTLINE. We briefly introduce WebDSL's data modeling concepts and its ORM in the next section. In Section 4.3, we discuss evolution specification. In

Sections 4.4 to 4.6, we address the generation of migration code for selected operators in detail. We conclude the chapter with a discussion in Section 4.8.

4.2 WEBDSL

WebDSL is a domain-specific language for the development of dynamic web applications that integrates data models, user interface models, actions, validation, access control, and workflow [Visser, 2008a]. The WebDSL compiler verifies the consistency of web applications and generates complete implementations in Java. In this section, we focus on WebDSL's data modeling concepts and the ORM underlying the generated Java code.

4.2.1 *Data modeling*

A data model definition in WebDSL features *entity* declarations, which comprise a name and a set of properties. An entity declaration might inherit from another entity declaration, indicated with `:`. Each *property* has a name and a type. We distinguish two kinds of properties: Value properties, indicated with `::`, and associations, indicated with $\rightarrow$. For value properties, WebDSL supports basic data types such as **`Bool`** and **`String`**, but also domain-specific types such as **`Email`**, **`Secret`**, and **`WikiText`**, which all provide additional functionality. Associations refer either to entities declared in the data model (*single-valued*) or to a **`Set`** or **`List`** thereof (*multi-valued*).

Figure 4.1 (left) shows a data model for a publication management application similar to Researchr[1]. It models publications written by authors and a special type of publication, namely the published volume. Additionally, users can register and create personal bibliographies, which are collections of publications.

4.2.2 *Object-relational Mapping*

WebDSL's data modeling concepts abstract over implementation details for persistence. These details are added by the WebDSL compiler which addresses Hibernate as a persistence framework. At runtime, application data is kept in objects which are stored persistently in a relational database. There is a database table for each hierarchy of entities, named after the root entity declaration in a hierarchy. Throughout the chapter, we will call these tables *hierarchy tables*. In the running example, there will be four tables `_User`, `_Bibliography`, `_Publication`, and `_Author`. Each of these tables has at least two columns: `id` stores object ids and acts as the primary key of the ta-

[1] Researchr is a web application for finding, collecting, sharing, and reviewing scientific publications: `http://researchr.org`.

```
entity Author {
   name      :: String
}

entity User {
   email     :: Email
   password  :: Secret
   public    :: Bool
}

entity Bibliography {
   owner
      → User (not null)
   publications
      → Set<Publication>
}

entity Publication {
   key       :: String
   title     :: String
   abstract  :: WikiText
   authors   → List<Author>
}

entity PublishedVolume
   : Publication {
   publisher :: String
}
```

```
entity Person {
   alias
      → Set<Alias> (not empty)
   email     :: Email
}

entity Alias {
   name      :: String (id)
}

entity User : Person {
   password  :: Secret
}

entity Bibliography {
   public    :: Bool
   owner
      → User (not null)
   publications
      → Set<Publication>
}

entity Publication {
   registrant → User
   key       :: String
   title     :: String
   abstract  :: WikiText
   authors   → List<Person>
}

entity PublishedVolume
   : Publication {
   editors   → List<Person>
   publisher :: String
}
```

Figure 4.1 Original and evolved data model for the running example

```
1 create Publication.registrant → User;
2 collect Bibliography.public over owner;

3 rename entity Author to Person;
4 create PublishedVolume.editors → List<Person>;
5 add super Person to User;
6 pull up Person.email;

7 extract
     entity Alias{ name::String }
     from Person
     as alias;
8 make Alias.name id;
9 generalize Person.alias to Set;
```

Figure 4.2 Evolution Model for the running example

ble while `DISCRIMINATOR` is used to distinguish which entity in the hierarchy is instantiated by an object. Object ids are implemented by universally unique identifiers (UUIDs) and are therefore database-wide (and beyond) unique.

Additional columns are added for each value property and for each single-valued association declared in one of the entities in an entity hierarchy. Since columns for single-valued associations will store the ID of a referred object, they act as implicit references. The implicit references are made explicit by a foreign key to the `id` column in the table corresponding to the type of an association. The RDBMS enforces foreign keys by preventing (or canceling) database operations that break integrity. The `_Bibliography` table will have three columns: `id` (primary key), `DISCRIMINATOR`, and `Bibliography_owner` (with a foreign key to `id` in `_User`).

Multi-valued associations are stored in separate *connection tables*. The names of these tables are composed from the names of the declaring entity, the association, and the association type. Each connection table has two columns to store pairs of object ids (referring and referred object). Both columns are foreign keys to the `id` column of the table corresponding to the declaring entity respectively the association type. A multi-valued association can either be a **`Set`** or a **`List`**. For sets, we place a primary key on the two columns, since we may not store a pair of objects twice. For lists, an additional index column is needed to persist order. Here we place a primary key on the combination of declaring entity reference and the index, since there can just be one reference per position in a list. For example, the association `Publication.authors` is stored in a connection table named `Publication_authors_Author` comprising three columns, namely: `_Publication_id` (with a foreign key to `id` in `_Publication`), `Publicationauthorindex`, and `authors_id` (with a foreign key to `id` in `_Author`), where the first two columns act as the primary key.

4.3 MODELING DATA MODEL EVOLUTION

Typically, the evolution of a data model is only implicitly defined by its original and evolved version. For example, the right part of Figure 4.1 shows an evolved version of the data model. In this section, we discuss means to model this evolution explicitly.

4.3.1 *Coupled Operators*

Informally, the example evolution follows three stages: First, bibliography management is extended by allowing users to submit new publications, hence they are linked to publications as `registrant` and can now individually set bibliography visibility. This requires adding an association from `User` to `Publication` and a `public` property to bibliographies. The latter is

collected from the `owner` of a bibliography as not to lose the user settings. Second, the system is refactored to support editors. This requires addition of an `editors` association, as well as renaming `Author` to a more general `Person` . Consequently, `User` can become a sub entity of `Person`, since they may also be editor or author of publications. The `email` of users is then generalized to be able to store email addresses for editors and authors. Third, the system is extended to support people (authors, editors, or users) to have different name aliases. Therefore, a person's `name` is extracted into a new entity, in which names are stored uniquely.

We can model evolution as a sequence of coupled operator applications. At the data model level, coupled operators capture common evolution steps. Thereby, they go beyond simple creations, changes, and deletions of entities and properties. For example, the evolution model from Figure 4.2 includes the collection of a property over an association, the pull-up of a property into a parent entity, and the extraction of an entity. Each of these operators couples the evolution step at the data model level with a corresponding migration. This allows us to compile evolution models into migration code for the database level.

4.3.2 *Linguistic Integration*

The language for evolution models is linguistically integrated with WebDSL. It reuses WebDSL's data modeling concepts and parts of their syntax definition. For example, constructs for property and entity creation reuse the syntax for properties and entities. An evolution model includes references to the original and evolved data model. Static checks ensure evolution validity and correctness with respect to these data models. For evolution validity, preconditions for operator applications are checked in the context of the original data model (Chapter 5). These preconditions secure that the evolution can be applied to the original data model. For evolution correctness, it is checked whether the evolution maps the original data model to the evolved data model.

4.3.3 *Migration*

To migrate the original database, each operator application in an evolution model is compiled to its corresponding migration code. Thereby, the compiler follows the same ORM as the WebDSL compiler, namely Hibernate. This ensures that migrating the original database and generating an initial schema for the evolved data model will result in the same database schema. Furthermore, the compiler is aware of the RDMBS and generates details such as character set definitions, collations, and storage engines[2].

[2]In the examples, we omit these details for readability.

In the following sections, we discuss database migration for selected coupled operators. Thereby, we distinguish three kinds of migrations. Operators such as property and entity creation only require *schema modification*. Their corresponding migrations affect the database schema but not the stored data. We discuss such operators in Section 4.4. Many other operators such as entity renaming, entity extraction, super addition, or cardinality generalization allow for *conservative data migration*. Their corresponding migrations affect both the database schema and the stored data. But the stored data is completely preserved during migration, no data is lost. We discuss such operators in Section 4.5. Only few operators such as property collection or property identification require *lossy migration*. Their corresponding migrations may not preserve the stored data completely. Some data may be lost intentionally during migration. We discuss such operators in Section 4.6.

4.4 SCHEMA MODIFICATION

Schema modifying migrations change the database schema, but leave the persistent data untouched. They generally allow for more information to be stored and are thereby most commonly needed while extending application functionality.

4.4.1 *Property Creation*

In Chapter 3, we identified two coupled operators for property creation: one for value properties and one for associations. But in WebDSL and its Hibernate configuration, single-valued associations and multi-valued associations are dealt with differently. The first is stored inside the containing entity's table, the second is stored in its own connection table. Thus, Acoda provides three different coupled operators for property creation: one for value properties, one for single-valued associations, and one for multi-valued associations.

When we create a new value property in a data model, the original database schema is missing a column for this property. To be precise, the hierarchy table corresponding to the entity containing the new property is missing a column. We need to create the missing column in order to migrate the original database.

For the creation of a new single-valued association, the migration is similar. Again, the hierarchy table corresponding to the containing entity is missing a column for storing ids of the associated entity. Additionally, a foreign key needs to be created to enforce validity. This foreign key needs to point to the `id` column of the hierarchy table corresponding to the associated entity.

EXAMPLE. Acoda generates the following migration for the creation of the association `registrant` in the running example:

```
1 create Publication.registrant → User;
```

```
ALTER TABLE _Publication
   ADD COLUMN `Publication_registrant`
      VARCHAR(32) default NULL,
   ADD CONSTRAINT f_Publication_registrant
      FOREIGN KEY `f_Publication_registrant`
         (Publication_registrant)
      REFERENCES _User (id);
```

It consists of a single SQL statement altering the `_Publication` table. First, it adds a new column `Publication_registrant` to store the association. Afterwards, it constrains this column with a foreign key to `id` in `_User`.

In contrast to single-valued references, multi-valued references are stored in separate connection tables. When we create a new multi-valued association in a data model, the original database schema is missing a table for this association. We need to create this table in order to migrate the original database.

EXAMPLE. Acoda generates the following migration for the creation of the `editors` association:

```
4 create PublishedVolume.editors → List<Person>;
```

```
CREATE TABLE `PublishedVolume_editors_Person` (
   `_PublishedVolume_id` VARCHAR(32) default NULL,
   `_editors_id` VARCHAR(32) default NULL,
   `PublishedVolumeeditorsindex` integer,
   PRIMARY KEY (`_PublishedVolume_id`,
      `PublishedVolumeeditorsindex`),
   INDEX `forward_lookup`
      (_PublishedVolume_id(14)),
   CONSTRAINT `f_PublishedVolume_editors_b`
      FOREIGN KEY `f_PublishedVolume_editors_b`
      (_PublishedVolume_id)
      REFERENCES _Publication (id),
   CONSTRAINT `f_PublishedVolume_editors_f`
      FOREIGN KEY `f_PublishedVolume_editors_f`
      (_editors_id)
      REFERENCES _Person (id)
);
```

It comprises a single SQL statement creating a table connecting records in `_PublishedVolume` to records in `_Person`. The table has three columns to store ids of published volumes, ids of persons, and list indices since order does matter. For a published volume and an index, the associated editor needs to be unique. Thus, the published volume and the index form the primary key of the table. Validity of the two columns which store ids is ensured by foreign keys. These point to the `id` columns in the connected tables. In order to support efficient use of the connection table, database indices are generated for the primary key, allowing efficient single editor lookup, and for the published volumes column, allowing efficient collection of the complete list of editors for a published volume.

4.4.2 *Entity Creation*

Similar to property creation, Acoda provides different coupled operators for the creation of a new entity: one for entities that do not extend another entity and one for entities that do.

When we create a new entity which does not extend another entity, the original database is missing a hierarchy table for this entity and connection tables for its multi-valued associations. We need to create these tables in order to migrate the original database. Since we explained the creation of connection tables already in the previous section, we only focus on the hierarchy table. Following Hibernate, this table needs to be named like the entity and needs to provide two columns `id` and `DISCRIMINATOR` as well as additional columns for value properties and single-valued associations. Columns for single-valued associations need to be constrained by foreign keys.

When we create a new entity which extends another entity, the original database is missing columns for its value properties and single-valued associations and connection tables for its multi-valued associations. The columns are missing in the hierarchy table of the extended entity. Thus, the migration is the same as for creating the properties of the new entity. Creating its value properties and single-valued associations will add the missing columns while creating its multi-valued associations will add the missing connection tables.

Figure 4.3 presents creation of an `entity A:B` with single- and multi-valued features `sf 1 .. sf n` and `mf 1 .. mf m` graphically. The figure above the dashed line shows the database before migration, the figure below the dashed line after migration. Each array denotes a table, each cell within an array a column. An open cell denotes columns which were already present before modification and remain unaltered. A solid double arrow denotes a uniqueness key, a dashed double arrow denotes a database index, and a single arrow denotes a foreign key. `A*` denotes the root entity in the hierarchy of entity `A`, `t(a)` denotes the target type of an association `a`. The id columns are always unique. We therefore omit the double solid uniqueness arrow on id columns.

4.5 CONSERVATIVE DATA MIGRATION

Conservative migrations are needed when the domain of an application shifts or expands. They change the schema and rearrange data but do not lose information. Conservative migrations are most common in practice, yet tedious and error-prone to write manually.

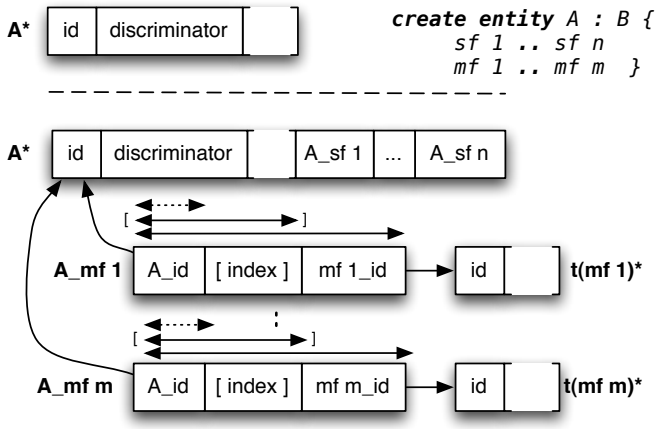


Figure 4.3 Database modification for entity creation

4.5.1 *Entity Renaming*

In schema generation, entity names influence table and column names in hierarchy and connection tables as well as associated foreign keys. When an entity is renamed, these names need to be updated and foreign keys need to be recreated. More specifically, renaming entity `A` requires the following schema modifications:

1. Drop foreign keys for single- and multi-valued associations in `A`
2. Drop foreign keys for single- and multi-valued associations of type `A`
3. Rename the hierarchy table for `A` if `A=A*`
4. Rename columns for value properties and single-valued associations in `A`
5. Rename connection tables for multi-valued associations in `A`
6. Rename columns for multi-valued associations in `A`
7. Rename columns for multi-valued associations of type `A`
8. Create foreign keys for single- and multi-valued associations in `A`
9. Create foreign keys for single- and multi-valued associations of type `A`

However, entity names are also used as discriminator, distinguishing between different entities in a hierarchy. An entity rename therefore needs to migrate the data inside the hierarchy table for `A`, by replacing the old entity name in the `DISCRIMINATOR` column by the new entity name.

EXAMPLE. Acoda generates the following migration for renaming `Author` to `Person`:

```
3 rename entity Author to Person;
```

```
ALTER TABLE Publication_authors_Author
   DROP FOREIGN KEY `f_Publication_authors_f`;
ALTER TABLE _Author
   RENAME _Person;
ALTER TABLE Publication_authors_Author
   RENAME Publication_authors_Person,
   ADD CONSTRAINT `f_Publication_authors_f`
      FOREIGN KEY `f_Publication_authors_f` (_authors_id)
      REFERENCES _Person (id),
UPDATE _Person
   SET DISCRIMINATOR = "Person"
   WHERE DISCRIMINATOR = "Author";
```

First, the foreign key for the `Publication.authors` association is dropped, after which the hierarchy table `_Author` can be renamed. Next, the connection table for the association is renamed and the dropped foreign key is recreated. Finally, the object discriminators are updated.

4.5.2 *Super Addition*

When the original application models two inheritance-unrelated entities, separate tables are used to store the inheritance trees of both. When the application evolves by adding a super entity joining the two inheritance trees, the target application only uses a single table to store both entities. To support the new application, the original tables and associated data needs to be merged. The schema modifications are presented graphically in Figure 4.4. The migration of adding super entity `A` to entity `B` comprises the following steps:

1. Expand the table for `A*` by all single-valued properties in the inheritance tree of `B` (`inh(B)`)
2. Create foreign keys for outward single-valued associations in `inh(B)`
3. Copy single-valued data from the old table for `B` to the table for `A*`
4. Drop foreign keys for outward (multi-valued and single-valued) associations in `inh(B)`
5. Create foreign keys for outward multi-valued associations in `inh(B)`
6. Drop foreign keys for inward associations to `inh(B)`
7. Create foreign keys for inward associations to `inh(B)`

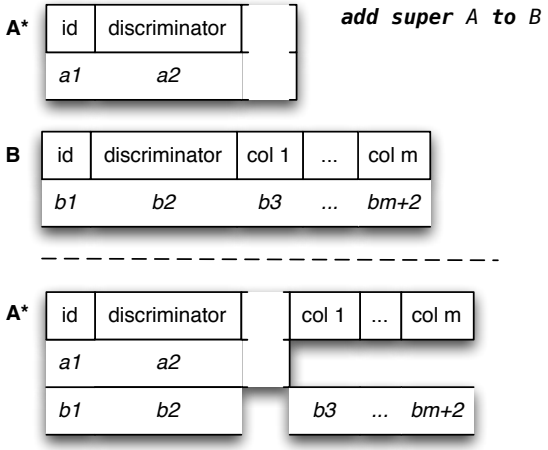


Figure 4.4 Database modification for super type addition

8. Drop the old table for `B`

Step 1 creates the new space to store data for `B` and its sub entities inside the table, which was originally only used for `A*` and its sub entities. Step 2 creates foreign keys pointing away from the table of `A*`. Step 3 prevents any loss of data. Steps 5 and 7 create foreign keys pointing to the table of `A*`, which work on the copied data. Steps 4 and 6 drop all foreign keys, that point to the table for `B`, to prevent breaking the database integrity. Finally, step 8 deletes the old data. The order of steps is crucial, it targets to maximize the number of constraints at any point in migration: Foreign keys for outward single-valued associations are added before copying data, since they point away from the table for `A*` and thereby also hold on an incomplete (or empty) set of `B` records. Foreign keys for outward multi-valued associations and inward associations are created after copying, since they point to the table for `A*` and therefore require a complete data set. The foreign keys are dropped before the data is dropped to prevent them from breaking and the foreign keys are dropped before they are recreated to prevent name clashes. Note that except for their foreign keys, any connection table associated to `inh(B)` remains unaltered.

EXAMPLE. In the running example, `Person` becomes super entity of `User`. Following the scheme outlined above: `User` has two single-valued properties `email` and `password`, which are added to the `Person` table in step 1. Both these properties are attributes, hence step 2 can be omitted. Next, the user data is copied from the `User` table to the `Person` table in step 3. Step 4 can again be omitted. User has one inward association `registrant` from `Publication`, whose new foreign key is added in step 5 and whose old foreign

key is dropped in step 7. Step 6 can again be omitted and step 8 drops the old user data. The steps are formalized in the following migration:

```
5 add super Person to User;
```

```
ALTER TABLE _Person
   ADD COLUMN `User_email` VARCHAR(255) DEFAULT '';
   ADD COLUMN `User_password` VARCHAR(255) DEFAULT '';
INSERT INTO _Person
   (id,DISCRIMINATOR, version_opt_lock,
      User_email, User_password)
   SELECT id,DISCRIMINATOR,
      version_opt_lock,_email,_password
   FROM _User;
ALTER TABLE _Publication
   DROP FOREIGN KEY `f_Publication_registrant`;
ALTER TABLE _Publication
   ADD CONSTRAINT `f_Publication_registrant`
   FOREIGN KEY `f_Publication_registrant`
   (Publication_registrant)
   REFERENCES _Person (id);
DROP TABLE _User;
```

4.5.3 *Entity Extraction*

To enrich a data model, an entity may need to be extracted from another entity. During entity extraction, a new entity is created using some or all of the properties of an existing source entity. A single-valued association is created to link objects of the two entities. An example entity extraction can be found in the running example, where `Alias` is extracted from `Person`, using a new association `alias`. We distinguish the following steps in a migration for extracting entity `B` from `A` as `a`:

1. Adapt the schema to store `B`
2. Add a column for `a` to the table for `A*`
3. Generate new identifiers in the column for `a`
4. Copy `a` as id and other single-valued columns in `B` from the table for `A*`
5. Create a foreign key for `a`
6. Drop the old columns in the table for `A*`
7. Move the data for multi-valued properties in `B` and update their ids using the mapping provided by `a`

Step 1 comprises a migration for creating an entity, as discussed in Section 4.4.2. Step 2 adds a column, but leaves out its foreign key. Step 3 generates new ids, which can be sequentially numbered, or as in our case UUIDs. Step

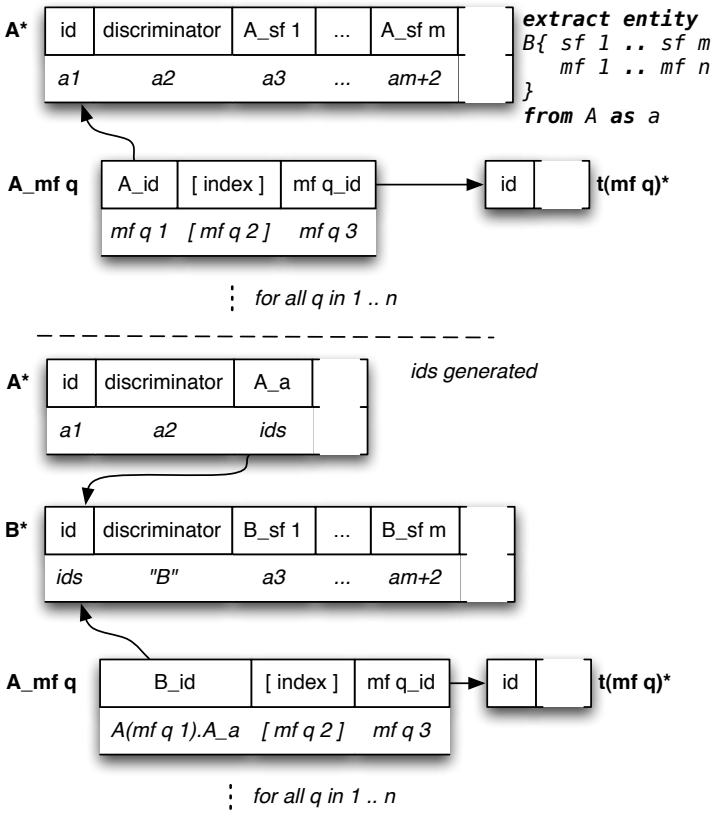


Figure 4.5 Database modification for entity extraction

4 then performs the extraction for all single-valued data, by copying their columns including the newly generated ids and a discriminator (`'B'`) to the new table. As the identifier duplication now validates the foreign key, it can be created in step 5. Step 6 then drops the old single-valued data from the table for `A*`. Finally, step 7 moves the multi-valued data to new connection tables, which were created in step 1. This data references `A` objects, whereas they should now be referencing `B` objects, therefore there links need to be updated using the mapping specified in the table for `A* (id, A_a)`. After moving the multi-valued data, the old connection tables are dropped. Note that step 4 moves each property across association `a` to `B`. We could therefore have reused the migration generation for moving properties, yet this would yield an inefficient migration. Copying all data in one pass over the table for `A*` is more efficient then separate passes for each of the single-valued properties in `B`.

Figure 4.5 shows the database before and after migration. The data set identifiers are generated (step 3 above) and the data set for `B_id` is obtained

by applying the mapping from `A` objects to `B` objects (step 7).

EXAMPLE. In the running example, we extract entity `Alias` and its `name` property from `Person`, while creating an association `alias`. To adapt the database, we generate the migration shown below. Step 7 above is not represented, since `Alias` has no multi-valued properties.

```
7 extract entity Alias{name::String} from Person as alias;
```

```
CREATE TABLE IF NOT EXISTS `_Alias` (
   `DISCRIMINATOR` VARCHAR(255) default '',
   `id` VARCHAR(32) default NULL,
   `Alias_name` VARCHAR(255) default '',
   PRIMARY KEY (`id`)
);
ALTER TABLE _Person
   ADD COLUMN `Person_alias` VARCHAR(32)
      default NULL;
UPDATE _Person
   SET Person_alias = UUID();
INSERT INTO _Alias
   SELECT "Alias", Person_alias, Person_name
   FROM _Person;
ALTER TABLE _Person
   ADD CONSTRAINT `f_Person_alias`
   FOREIGN KEY `f_Person_alias` (Person_alias)
   REFERENCES _Alias (id);
ALTER TABLE _Person
   DROP COLUMN Person_name;
```

4.5.4 *Maximum Cardinality Generalization*

During the lifetime of an application, attributes often get generalized to expand the application's functionality. One type of property generalization is increasing its maximum cardinality. Any multi-valued cardinality uses the same database structure, its exact number is irrelevant. However, a single-valued association is represented as a column, whereas a multi-valued association as a connection table. In the running example, a person's alias is stored within the `Person` table before step 7 and stored in a connection table afterwards. To support such generalization, we need to generate a migration, which first creates the connection table, then moves the data from the main table to the connection table and subsequently drops the old column.

EXAMPLE. For generalizing the maximum `alias` cardinality in the running example, we generate the migration below. The first statement creates a connection table as discussed in Section 4.4.1. The second statement inserts the old data into the new connection table. The last two statements drop the old column by first dropping the foreign key and then dropping the column itself.

```
9 generalize Person.alias to Set;
```

```
CREATE TABLE IF NOT EXISTS `Person_alias_Alias` (
   `_Person_id` VARCHAR(32)default NULL,
   `_alias_id` VARCHAR(32)default NULL,
   INDEX `forward_lookup` (_Person_id(14)),
   CONSTRAINT `f_Person_alias_b`
      FOREIGN KEY `f_Person_alias_b` (_Person_id)
      REFERENCES _Person (id),
   CONSTRAINT `f_Person_alias_f`
      FOREIGN KEY `f_Person_alias_f` (_alias_id)
      REFERENCES _Alias (id)
);
INSERT INTO Person_alias_Alias
   SELECT id, Person_alias
   FROM _Person
   WHERE Person_alias IS NOT NULL;
ALTER TABLE _Person
   DROP FOREIGN KEY f_Person_alias;
ALTER TABLE _Person
   DROP Person_alias;
```

4.5.5 *Property Pull-Up*

For pulling up a property, Acoda provides different migrations for value properties, single-valued associations, and multi-valued associations. Value properties as well as single-valued associations are stored inside the inheritance hierarchy table. A property is pulled up from each of the sibling entities inside the hierarchy. The pulled up property is stored in one database column. During migration, the values for the different sibling columns need to be combined. This is achieved by creating the new column `A_f`, copying the data sets of each of the sibling properties separately and dropping the sibling properties afterwards. Figure 4.6 presents single-valued pull up. The pulled up data (`a12` to `an2`) is merged to form a new column `A_f`. When associations are pulled up, the old foreign keys are dropped (arrows in figure) and a single foreign key is created along with the new column `A_f`.

EXAMPLE. In the running example, `email` is pulled up from `User` to `Person`. Email is a single-valued property and `User` has no sibling entities. Thus, for the example, we need to merge a single column with no foreign key, which amounts to creating a new column, copying the data and dropping the old column:

```
6 pull up Person.email;
```

```
ALTER TABLE _Person
   ADD COLUMN `Person_email`
   VARCHAR(255) DEFAULT '';
UPDATE _Person
   SET Person_email = User_email
   WHERE DISCRIMINATOR='User';
ALTER TABLE _Person
   DROP COLUMN `User_email`;
```

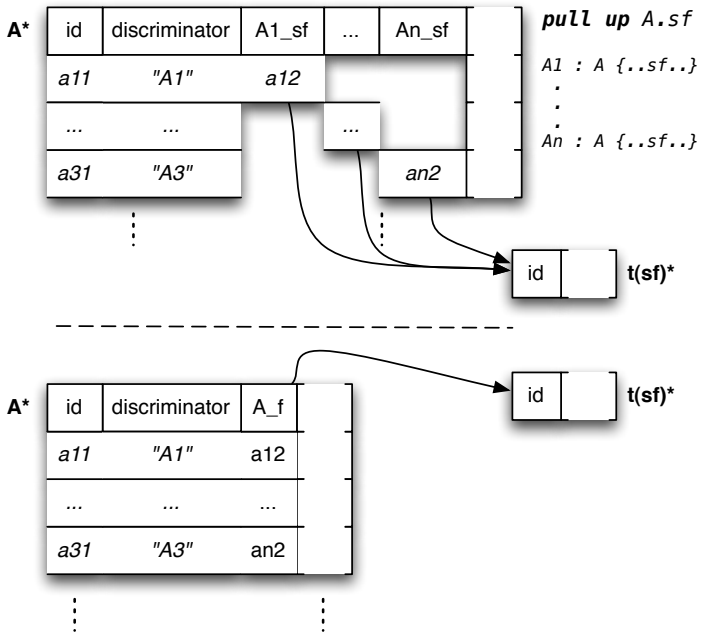


Figure 4.6 Database modification for single-valued property pull up

When pulling up a multi-valued association, the set of sibling associations is stored in a collection of connection tables. These need to be merged into a new table which has column names and foreign keys adapted to the new containing type. In contrast to single-valued associations, merging of multi-valued associations comprises a union of the sibling data sets and can thus be done in one SQL statement.

4.6 LOSSY MIGRATION

Although data loss is generally not desirable, when correcting design flaws it can often not be avoided. Additionally, in many cases a migration may in theory potentially cause data loss, yet in practice for many databases this will not actually be the case.

4.6.1 *Property Collection*

It is common for properties to be repositioned during the life-span of an application. They can be repositioned across an inheritance relation (e.g. `pull up`), but can also be repositioned across an association. In WebDSL, associations are directed. When repositioning a property in the direction of the association,

we speak of *moving* a property, when repositioning opposite to the association direction, we speak of *collecting* a property. When a property is repositioned across an association, we call the association a *bridge*.

There are two main reasons for collecting properties: First, the application may use numerous dereferences to access a property, in which case the dereference can be made permanent by collecting the property. Second, the property might no longer logically belong to the referred object but to the referring object. This is the case in our running example: We want to distinguish for each bibliography if it is public or not. In the original data model, the distinction is made only on a per user basis. Thus, the corresponding property `public` needs to be collected from `User` to `Bibliography`, using `owner` as a bridge.

Property collection may cause loss of data, since the bridge may not be surjective. There may be users who have set their public field but do not have a bibliography. To adapt a database to a collected single-valued property `f` in `A` from `B` across single-valued association `bridge`[3], we first create the new column to store `f`. Next, we join the tables for `A*` and `B*` (we compute their cross product) and filter the result on records where the bridge holds `(A.bridge = B.id)`. Then we copy the old column for `f` to the new column for `f` in the cross product result. Finally we drop the old column for `f`. If `f` is an association, its new foreign key needs to be created along with creating its new column and its old foreign key needs to be dropped before dropping the old column. Note that the database index on the (primary) id column of the table for `B` ensures that the table join can be computed efficiently.

Figure 4.7 shows the process graphically, in which the middle stage represents the intermediate signature during update. During migration, data is typically duplicated: A user can have multiple bibliographies, each of which gets the same public value.

EXAMPLE. To collect `public` from `User` to `Bibliography` in our running example, we generate the migration below. The first and last statement create and delete columns to store the `public` property. The second statement copies (and duplicates) the information by updating the `Bibliography` and `User` table joined together using `owner` as criterium.

```
2 collect Bibliography.public over owner;
```

```
ALTER TABLE _Bibliography
   ADD COLUMN `Bibliography_public` BIT(1)
   DEFAULT FALSE;
UPDATE _Bibliography target, _User source
   SET target.Bibliography_public = source.User_public
   WHERE target.Bibliography_owner = source.id;
ALTER TABLE _User
   DROP User_public;
```

[3]Note that `A` and `B` could be the same type

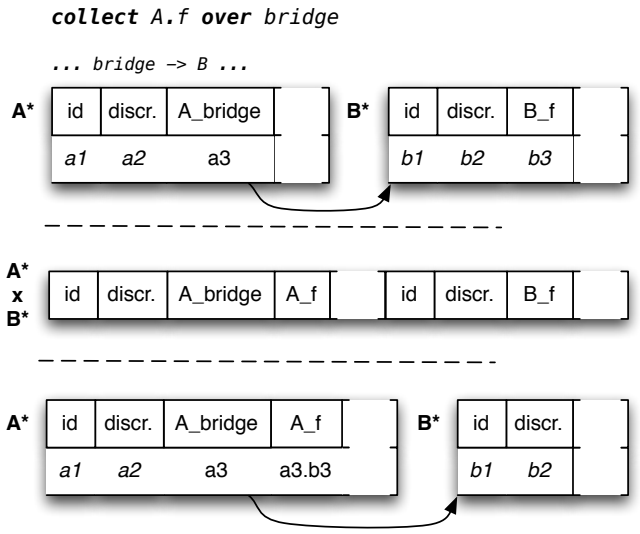


Figure 4.7 Database modification for property collection

There are different migrations for collecting multi-valued properties and for collecting properties across a multi-valued bridge. In both cases, the migration is extended by including connection tables. When collecting a property across a multi-valued bridge, we need to extend the join above by the connection table representing the bridge. When collecting a multi-valued property, the property is represented as a connection table and we can thereby make a new connection table by rewriting the connection table's reference to `B` into a reference to `A`, using the bridge. To apply the rewriting efficiently, a database index on the bridge needs to be generated first.

4.6.2 *Property Identification*

When a property is kept unique by the application, yet is not modeled as such, it can be made unique to ensure correctness of the application logic. Also, when data is stored redundantly, it can be compacted by enforcing uniqueness of redundant properties. The latter is the case in the running example, where multiple aliases with the same name exist after entity extraction. By making an alias' name unique, only one object would be needed per name.

Although the schema generated for the new application would match the original schema, the application logic assumes property uniqueness, whereas this is not guaranteed by the database. The original database may contain duplicate values. Migration needs to resolve these duplicates as to ensure uniqueness. There are two approaches to enforcing uniqueness: Either the identifying values are adapted to be unique, yet it is hard to provide a decent strategy to do so and in practice this is rarely desirable. Or the objects which

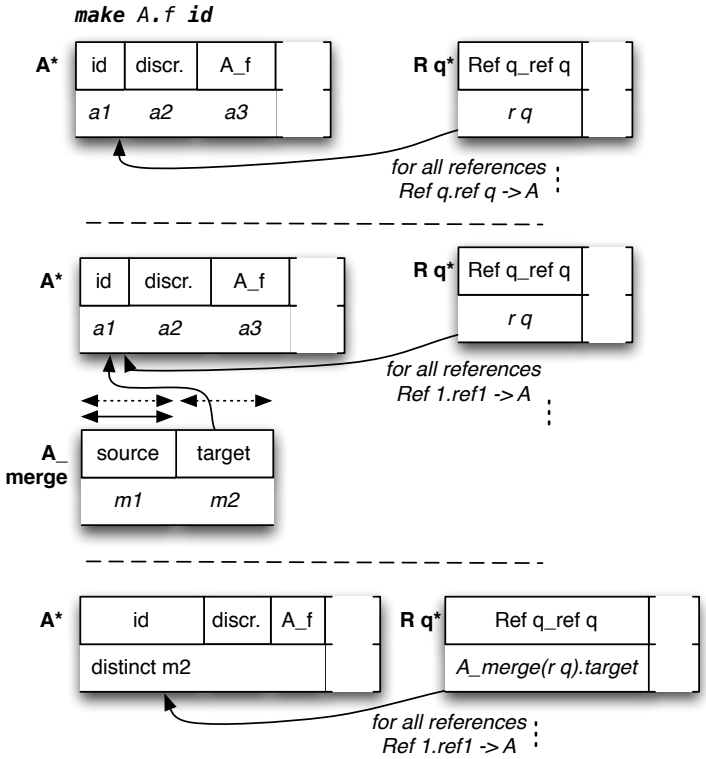


Figure 4.8 Database modification for attribute identification

contain duplicate values are merged. We use the latter. It may merge objects, which are not exactly the same, in which case information is lost.

Merging objects along an identifying property comprises two tasks: the objects need to be merged and all associations to these objects need to be updated to point to the merged objects. Both tasks make extensive use of a mapping from original objects to merged objects. As this mapping is computationally complex to derive, we compute it once and reuse the result. The schematical changes for making property `A.f` an identifier are shown in Figure 4.8. The top-most part shows the table for `A` and associations to this table, which may both be from single-valued associations (columns) as well as multi-valued associations (connection tables). The middle part shows the computed mapping from `A` object ids (source) to merged `A` object ids (target). Only the `target` column has a foreign key to `A*`. At the start of migration the source column also references `A` ids, yet after merge, `source` may point to no longer existing, merged objects. The bottom part shows the schema after migration.

For making `Alias.name` an identifier (step 8 in the running example), Acoda generates the following migration:

```
8 make Alias.name id;
```

```
CREATE TABLE Alias_merge
(  INDEX forward_lookup (source),
   INDEX reverse_lookup (target)   )
   CONSTRAINT `f_Alias_merge`
      FOREIGN KEY `f_Alias_merge`
      (target)
      REFERENCES _Alias (id)
   SELECT original.id AS source, target
   FROM
      _Alias AS original,
      (  SELECT min(id) AS target, Alias_name
         FROM _Alias
         GROUP BY Alias_name    ) AS merged
   WHERE original.Alias_name = merged.Alias_name;
UPDATE _Person AS ref, Alias_merge AS map
   SET ref.Person_alias = map.target
   WHERE ref.Person_alias = map.source;
DELETE FROM _Alias
   WHERE NOT EXISTS
   (  SELECT *
      FROM Alias_merge AS map
      WHERE map.target = id);
ALTER TABLE _Alias
   ADD CONSTRAINT `Alias_name_unique`
   UNIQUE (_name);
DROP TABLE MergeMap_Alias;
```

The first statement computes and stores the mapping form original aliases to merged aliases. It uses two indices for efficient lookup: a forward index to rewrite the associations and a backward index to update the alias table. The second statement updates the `alias` association from `Person`, which at this point in migration is still single-valued. The update joins the `Person` table and the map to update all associations efficiently. The third statement drops the old and redundant aliases, which can now safely be removed, since they are no longer in use. The fourth statement enforces uniqueness and the final statement removes the merge map.

4.7 IMPLEMENTATION

The presented evolution modeling language is implemented as a part of Acoda[4]. To seamlessly integrate into regular development, Acoda offers an Eclipse plugin developed using Spoofax/IMP [Kats et al., 2009]. It operates in cooperation with the (already available) WebDSL plugin, which provides WebDSL application editing and compilation services. Acoda offers additional functionality around evolving WebDSL data models, such as comparison of original and evolved data model to yield an evolution model (Chapter 5); editor support for editing evolution models (such as syntax highlighting, instant error marking and content completion); generation of SQL migration code; and

[4]http://swerl.tudelft.nl/bin/view/Acoda

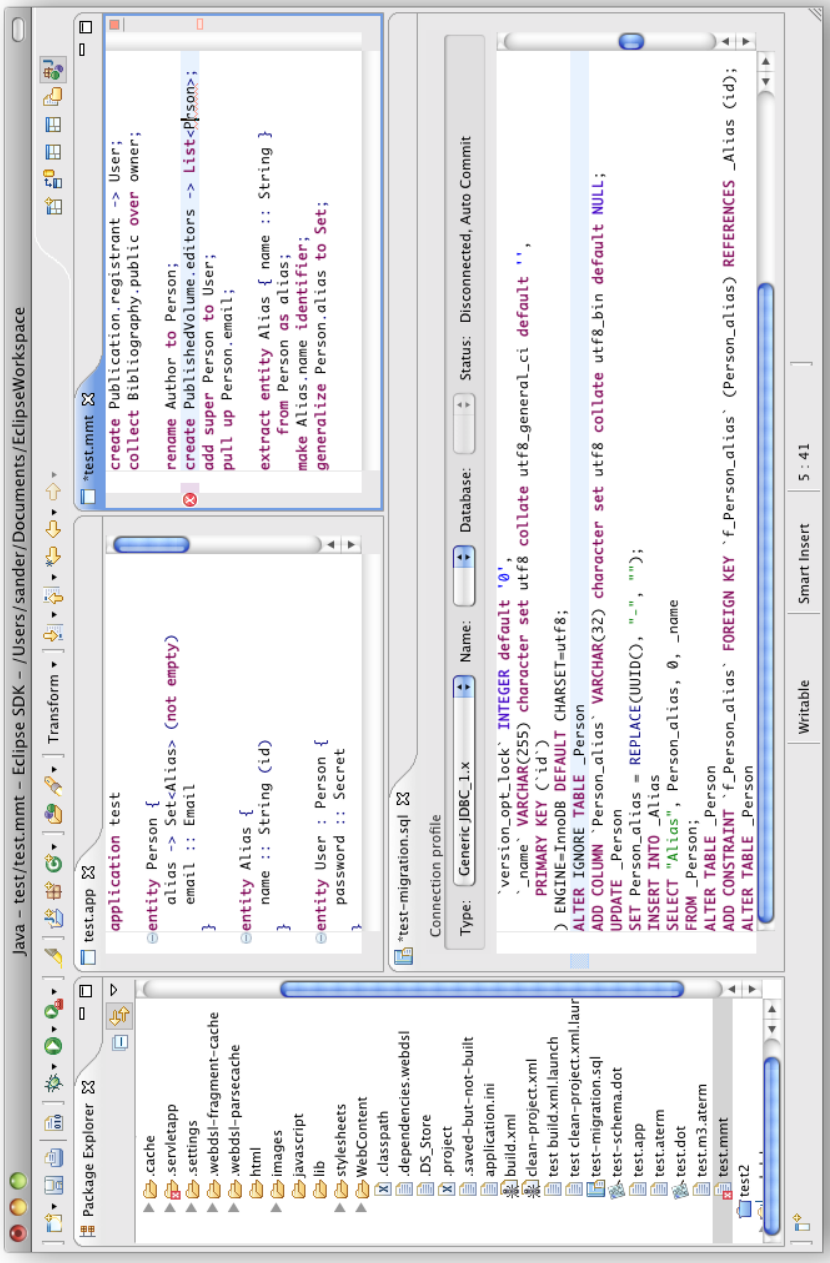


Figure 4.9 Screenshot of the Acoda Eclipse plugin. The left-most column shows the regular Eclipse project tree. The top-left editor displays a WebDSL data model. The top-right editor shows the evolution specification. The bottom editor shows the generated SQL migration.

application of migrations to a database. The plugin can be used in the context of agile development, in which it supports a short development - migration - deployment - testing loop. For migration of production databases, Acoda also offers a stand alone version, which can be run on-site or remotely.

Figure 4.9 shows a screenshot of the plugin. The left-most column shows the regular Eclipse project tree. The top-left editor displays a WebDSL data model (the running example). This editor is provided by the WebDSL plugin. The top-right editor shows the evolution specification used throughout the chapter, with a small typo to show evolution validity checking and corresponding error marking. This editor is provided by the Acoda plugin. The bottom editor shows the SQL migration generated by the plugin. Although this migration can be viewed and adapted by the developer, general practice is to apply the evolution specification directly, without examining SQL code. However, this still generates the SQL migration internally, which is then applied to the database.

4.8 DISCUSSION

4.8.1 *Related Work*

Migration generation is common in software development. Evolving data models require data migration, evolving DTDs require XML migration, and evolving schema require database migration. Furthermore, migration is not restricted to data modeling. It also occurs where meta-models evolve, where domain-specific languages evolve [Pizka and Jürgens, 2007b], and where grammars evolve. The coupled transformation problem is ubiquitous [Lämmel, 2004]. In this section, we relate our work to existing work on data model evolution and to work on coupled evolution in general.

Ruby on Rails offers support for migration of databases along an evolving web application[5]. The web applications use an ORM to persist data in a relational database. They offer support for versioning different databases running different versions of the same application. In contrast to our work, the Ruby on Rails migration support requires the developer to define database migrations himself in terms of the relational database. Ruby on Rails offers a set of SQL-like methods to alter a database, such as `create_table`, `add_column` and `remove_index`. They do not offer an evolution language at the application abstraction level.

In the area of data model evolution, most work focuses on evolving schema and migrating databases [Berdaguer et al., 2007, Gupta et al., 1993, Hainaut et al., 1994]. Schema describe structure of data storage, primarily focusing on storage techniques to improve query performance. Evolving schema requires the developer to be concerned with database implementation details.

[5] http://guides.rubyonrails.org/migrations.html

In our work, we bridge the ORM to allow the developer to define evolution in the application domain and abstract away from database details. From the application-level evolution specification, we generate schema evolution definitions (in SQL). We rely on the previous work on schema evolution to efficiently map the generated schema evolution onto a database migration. On the one hand, this allows the developer to reason in terms of application logic instead of database techniques. On the other hand, it allows us to introduce more advanced concepts into evolution specifications, such as inheritance, cardinalities, and associations.

Visser et al. formalize the more general coupled transformations [Cunha et al., 2006, Visser, 2008b, Alves et al., 2008a]: Not only conforming artifacts are considered (such as a database or XML document), also dependent artifact transformations are formalized (such as query and constraint migration). The formalization makes use of data refinement theory and uses Haskell for presentation. Visser et al. do not offer concrete migrations in addition to the presented formalization. Although they consider flattening hierarchies and present a formalization of such, they do not consider inheritance, or a complete ORM. In their concluding remarks, they point out that inheritance would be useful to include, to extend the scope to object-oriented data models.

Lämmel and Lohmann discuss migration of XML data along evolving DTDs [Lämmel and Lohmann, 2001]. They formalize the migration concepts and distinguish two groups of evolution: refactorings and structure-extending and -reducing evolutions. They discuss higher-level evolutions, such as folding and generalization. Lämmel and Lohmann do not offer a language for describing evolution.

Similar to the application models considered in our work, meta-models are defined in terms of high-level concepts, such as inheritance and cardinalities. Meta-model evolution languages cover a high level of abstraction and are similar to evolution steps on object-oriented data models [Cicchetti et al., 2008, Wachsmuth, 2007b, Herrmannsdoerfer et al., 2009, Hößler et al., 2005]. We therefore reused the evolution steps defined on meta-models, which are outlined in Chapter 3. In contrast to our work, in meta-modeling, there is a close relationship between the data set structure and the data definition: models closely follow the structure defined in their meta-model. The relational structure of a RDBMS, does not closely follow the object-oriented structure of an application-level model. Thus, where model migration does not need to cover the gap between defined structure and implemented structure, our work covers the mapping between object domain and relational domain: the ORM.

4.8.2 *Changing Persistence Implementation*

WebDSL abstracts over implementation details for persistence. The presented migration generation is aligned to Hibernate. But the WebDSL compiler might change some of the parameters for Hibernate's ORM or might even address

another persistence framework. Such changes would be transparent to evolution models, since Acoda abstracts over implementation details for migration and preserves WebDSL's data modeling abstractions. The Acoda compiler needs to reflect these changes and has to address the same ORM as the WebDSL compiler. These changes primarily amount to naming differences (of columns, tables, and keys) and a different type of inheritance representation (e.g. using separate tables for each entity, instead of hierarchy tables). To cope with naming differences, the naming in Acoda is pluggable and can be replaced by another naming scheme. To cope with a different inheritance representation, migration generations dealing with inheritance (e.g. the presented sub entity creation, property pull-up, and super addition) need to be adapted. Considering inheritance flattening is the more complex variant, adaptation will generally simplify migration generation.

4.8.3 *Performance & Uptime*

Databases may serve live web applications. Database migration may cause application downtime. Good performance of migration is important to limit downtime.

Acoda constructs migrations from database operations. Efficiency of their implementation depends on the used RDBMS. Nevertheless, we optimize the usage of database operations at two levels: First, we combine evolution operators at the data model level to form more complex operators with a more efficient migration at the database level. For example, a class creation and a feature addition can be combined into a single class creation. Second, we combine SQL operations in the generated migrations at the database level. Acoda compiles a sequence of evolution operators into a sequence of SQL statement sequences. These sequences may overlap. For example, two changes on a table (e.g. a rename and a column addition) may be generated for different evolution operators, yet can be combined into a single `ALTER TABLE` statement, thus significantly improving performance. The two kinds of optimizations target to generate the most efficient migration script.

Furthermore, the generated migrations attempt to shorten the time in which the database is inaccessible as much as possible. For example, the super addition postpones data deletion to the last step, even though it could have been applied earlier. This allows the database to stay online in read-only mode while the more computation intensive steps are executed (such as copying data). Additionally, migrations generally only target a part of the database, remaining application data stays accessible (both readable and writable). In practice however, most migrations are short and can be executed while the application is updated. They cause little or no additional downtime on regular-sized (WebDSL) databases.

4.9 CONCLUSION

The previous chapter (Chapter 3) focused on constructing an extensive catalog of operators for metamodel evolution. This chapter implemented this catalog for the evolution of web applications specified in WebDSL and the coupled migration of their databases. To specify evolution of WebDSL applications, it proposes an evolution DSL which shows close integration with the WebDSL language. The DSL receives an IDE to check evolution validity, ensuring the evolution can be applied, and evolution correctness, ensuring that applying the evolution yields the intended evolved WebDSL application. The chapter further describes the implementation of coupled operators, mapping from the evolution DSL to SQL migration scripts. It distinguishes schema modifications, which only adapt the database schema; conservative migrations, which rearrange data without data loss; and lossy migrations, which support intended loss of data.

The implementation of the coupled operators and the IDE are part of Acoda. Acoda is a tool set for evolving WebDSL applications. It has been used in the evolution of both case studies YellowGrass and Researchr, discussed in Appendices A and B respectively. The latter of these offered the basis for the running example of this chapter.

ACKNOWLEDGMENTS

This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*.

Reconstructing Complex Metamodel Evolution

5

ABSTRACT

Metamodel evolution requires model migration. To correctly migrate models, evolution needs to be made explicit. Manually describing evolution is error-prone and redundant. Metamodel matching offers a solution by automatically detecting evolution, but is only capable of detecting primitive evolution steps. In practice, primitive evolution steps are jointly applied to form a complex evolution step, which has the same effect on a metamodel as the sum of its parts, yet generally has a different effect in migration. Detection of complex evolution is therefore needed. In this chapter we present an approach to reconstruct complex evolution between two metamodel versions, using a matching result as input. It supports operator dependencies and mixed, overlapping and incorrectly ordered complex operator components. It also supports interference between operators, where the effect of one operator is partially or completely hidden from the target metamodel by other operators.

5.1 INTRODUCTION

Changing requirements, an increased knowledge of the domain and technological progress require metamodels to evolve [Favre, 2005]. Preventing metamodel evolution by downwards-compatible changes is often insufficient as it reduces metamodel quality [Casais, 1995]. Metamodel evolution may break conformance of existing models and thus requires model migration [Sprinkle, 2003]. To correctly migrate models, the evolution – implicitly applied by developers – needs to become explicit. Metamodel evolution can be specified manually by developers, yet this is error-prone, redundant and hard in larger projects. Instead, evolution needs to be detected automatically from the original and evolved metamodel versions.

The most-used solution for detecting evolution is matching [Sun and Rose, 2003]. Metamodel matching attempts to link elements from the original metamodel to elements from the target metamodel based on similarity. The result is a set of atomic differences highlighting what was created, what was deleted and what was changed.

PROBLEM. In practice, groups of atomic differences may be applied together to form complex evolution steps such as pulling features up an inheritance chain or extracting super classes (see Chapter 3). In model migration, a complex operator is different from its atomic changes. For example, pulling up a feature preserves information, whereas deleting and recreating it loses information. To correctly describe evolution, we therefore need to detect complex evolution steps.

There are three major problems in reconstructing complex evolution steps:

Dependency. While metamodel changes are unordered, evolution steps are generally applied sequentially and may depend on one another [Cicchetti et al., 2009]. Dependencies need to be respected by a mapping from metamodel changes to evolution steps.

Detection. To detect a complex evolution step, we must find several simple steps which make up this complex step. But these steps are likely to be separated, incorrectly ordered and mixed with parts of other complex evolution steps.

Interference. An evolution step can hide, change or partially undo the effect of another step. Multiple steps can completely mask a step. As such, some or all simple steps forming a more complex step may be missing, which impedes its detection.

EXAMPLE. Figure 5.1 shows two metamodel versions for a tag-based issue tracker. In the original metamodel on the left-hand side, each issue has a reporter, a title and some descriptive text. Projects are formed by a group of users and have a name and a set of issues. Users can comment on issues and tag issues. Additions and removals of tags are recorded, such that they can be reverted.

While evolving the issue tracker, tagging became the primary approach for organization. As such, it became apparent, that not only issues, but also projects should be taggable. Additionally, the metamodel structure had to be improved to allow users to more easily subscribe to events, as to send them email updates. The resulting metamodel is shown in Figure 5.1 (right). An Event entity was introduced, which comprises comments as well as tag events (tag additions and removals). Furthermore, projects obtained room for storing tags and events on these tags.

Matching the original and evolved metamodel yields the difference model presented in Figure 5.2. Two classes and seven features were added to the evolved metamodel (left column), eight features were subtracted (middle column), and three classes have an additional super type in the evolved metamodel (right column). We will use this difference model as a starting point to detect the complex evolution steps involved in the evolution of the original metamodel.

```
class Issue {
   title :: String
   description :: Text
   reporter → User
   project → Project
      opposite issues
   tags ♢ Tag (0..*)
}

class Project {
   name :: String
   issues → Issue (1..*)
      opposite project
   members → User (1..*)
}

class Tag {
   name :: String
}

class TagAddition {
   issue → Issue
   tag → Tag
   timestamp :: DateTime
}

class TagRemoval {
   issue → Issue
   tag → Tag
}

class Comment {
   issue → Issue
   timestamp :: DateTime
   content :: Text
   author → User
}

class User {
   ...
}
```

```
class Issue {
   title :: String
   description :: Text
   reporter → User
   project → Project
      opposite issues
   log ♢ Event (0..*)
      opposite issue
   tags ♢ Tag (0..*)
}

class Project {
   name :: String
   issues → Issue (1..*)
      opposite project
   members → User (1..*)
   log ♢ TagEvent (0..*)
   tags ♢ Tag (0..*)
}

class Tag {
   name :: String
}

class TagAddition : TagEvent
{ }

class TagRemoval : TagEvent
{ }

class Event {
   issue → Issue
      opposite log
   time :: DateTime
   actor → User
}

class TagEvent : Event {
   tag → Tag
}

class Comment : Event {
   content :: Text
}

class User {
   ...
}
```

Figure 5.1 Original and evolved metamodel for the running example

$$\bot \rightarrow \langle \texttt{Issue.log} \rangle \qquad \langle \texttt{TagAddition.tag} \rangle \rightarrow \bot$$
$$\bot \rightarrow \langle \texttt{Project.log} \rangle \qquad \langle \texttt{TagAddition.timestamp} \rangle \rightarrow \bot$$
$$\bot \rightarrow \langle \texttt{Project.tags} \rangle \qquad \langle \texttt{TagAddition.issue} \rangle \rightarrow \bot$$
$$\bot \rightarrow \langle \texttt{Event} \rangle \qquad \langle \texttt{TagRemoval.issue} \rangle \rightarrow \bot$$
$$\bot \rightarrow \langle \texttt{Event.issue} \rangle \qquad \langle \texttt{TagRemoval.tag} \rangle \rightarrow \bot$$
$$\bot \rightarrow \langle \texttt{Event.time} \rangle \qquad \langle \texttt{Comment.author} \rangle \rightarrow \bot$$
$$\bot \rightarrow \langle \texttt{Event.actor} \rangle \qquad \langle \texttt{Comment.issue} \rangle \rightarrow \bot$$
$$\bot \rightarrow \langle \texttt{TagEvent} \rangle \qquad \langle \texttt{Comment.timestamp} \rangle \rightarrow \bot$$
$$\bot \rightarrow \langle \texttt{TagEvent.tag} \rangle$$

$$\langle \texttt{TagAddition} \rangle \xrightarrow[\langle \texttt{TagEvent} \rangle]{+superTypes} \langle \texttt{TagAddition} \rangle$$

$$\langle \texttt{TagRemoval} \rangle \xrightarrow[\langle \texttt{TagEvent} \rangle]{+superTypes} \langle \texttt{TagRemoval} \rangle$$

$$\langle \texttt{Comment} \rangle \xrightarrow[\langle \texttt{Event} \rangle]{+superTypes} \langle \texttt{Comment} \rangle$$

Figure 5.2 Difference model for the running example

```
create feature TagRemoval.timestamp :: DateTime
extract super class TagEvent {issue, timestamp, tag}
      from TagAddition, TagRemoval

rename Comment.author to actor
create feature TagEvent.actor → User
extract super class Event {issue, timestamp, actor}
      from Comment, TagEvent
rename Event.timestamp to time

create feature Issue.log <> Event (0..*) opposite issue
create feature Project.log <> TagEvent (0..*)
create feature Project.tags <> Tag (0..*)
```

Figure 5.3 Evolution trace for the running example

The evolution of the metamodel can also be captured in an evolution trace as shown in Figure 5.3. At the metamodel level, the trace specifies the creation of five new features, the renaming of two other features, and the extraction of two new classes. At the model level, it specifies a corresponding migration. From the properties of the involved operators, we can conclude that the evolution is constructive and that we can safely migrate existing models without losing information.

In detecting the example evolution trace from the difference model, we face all three major problems in trace reconstruction several times. For example: The second step depends on the first step as it can only be applied if `TagRemoval` has a `timestamp`; Furthermore, the second step comprises several of the presented differences; And finally, the first step interferes with the second, since its effect is completely hidden from the difference model. The step needs to be reconstructed during detection.

Contribution & Outline. In this chapter, we provide an approach to reconstruct complex evolution traces from difference models automatically. It is based on the formalization of the core concepts involved, namely metamodels, difference models, and evolution traces (Section 5.2). First, we provide a mapping from changes in a difference model to primitive operators in an evolution trace. We solve the dependency problem by defining preconditions for all primitive operators. Based on these preconditions, we define a dependency relation between operators which allows us to order operators on dependency and to construct valid primitive evolution traces from a difference model (Section 5.3). Second, we show how to reorder primitive traces without breaking their validity and provide patterns for mapping sequences of primitive operators to complex operators. We solve the detection problem by reordering primitive traces to different normal forms in which the patterns can be detected easily (Section 5.4). Finally, we extend our method to detect also partial patterns in order to solve the interference problem (Section 5.5).

```
class MetaModel {
   classes      ◇ Class (0..*)
}

abstract class NamedElement {
   name         :: String (1..1)
}

abstract class Type
   : NamedElement {}

class Class : Type {
   isAbstract
:: Boolean (1..1)
   superTypes   → Class (0..*)
   features     ◇
Feature (0..*)
}

class DataType : Type {}

abstract class Feature
   : NamedElement {
   lowerBound
:: Integer (1..1)
   upperBound
:: Integer (1..1)
   type         → Type (1..1)
}

class Attribute : Feature {
   isId
:: Boolean (1..1)
}

class Reference : Feature {
   isComposite :: Boolean (1..1)
   opposite     → Reference
}
```

Figure 5.4 Metamodeling formalism providing core metamodeling concepts

5.2 MODELING METAMODEL EVOLUTION

5.2.1 *Metamodeling Formalism*

Metamodels can be expressed in various metamodeling formalisms. In this chapter, we focus only on the core metamodeling constructs that are interesting for coupled evolution of metamodels and models. We leave out packages, enumerations, annotations, derived features, and operations.

Figure 5.4 gives a textual definition of the metamodeling formalism used in this chapter. A metamodel defines a number of classes which consist of a number of features. Classes can have super types to inherit features and might be abstract. A feature has a multiplicity (lower and upper bound) and is either an attribute or a reference. An attribute is a feature with a primitive type, whereas a reference is a feature with a class type. We only support predefined primitive types like Boolean, Integer and String. An attribute can serve as an identifier for objects of a class. A reference may be composite and two references can be combined to form a bidirectional association by making them opposite of each other. In the textual notation, features are represented by their name followed by a separator, their type, and an optional multiplicity. The separator indicates the kind of a feature. We use `::` for attributes, $\rightarrow$ for ordinary references, and $\diamondsuit$ for composite references.

If we want to reason about properties of metamodels and their evolution, a textual representation is often not sufficient. Thus, we provide in Figure 5.5 a more formal representation of metamodels in terms of sets, functions, and predicates. In the upper left, we define instance sets for the metaclasses from Figure 5.4. In the upper right, we formalise most metafeatures from Figure 5.4

in terms of functions and predicates. Since super types and features of a class c form subsets of instance sets, we formalise them accordingly. In terms of these subsets, we define other interesting subsets, e.g., children, ancestors and descendants of c in the middle part. Typically, we refer to a class c by its name *cn* and to a feature f of class c by *cn.fn* where *cn* and *fn* are the names of c and f, respectively. To access classes and features referred by name, we define lookup functions in the last box. The formalization so far also captures invalid metamodels, such as metamodels with duplicate class names, or cycles in an inheritance hierarchy. Therefore, we define metamodel validity by a number of invariants in Figure 5.6.

5.2.2 *Difference Models*

Difference-based approaches to coupled evolution use a declarative evolution specification, generally referred to as the difference model [Cicchetti et al., 2008, Garcés et al., 2009]. This difference model can be mapped automatically onto a model migration. With an automated detection of the difference model, the process can be completely automated. Matching algorithms provide such a detection [Lopes et al., 2006, Falleri et al., 2008, Del Fabro and Valduriez, 2007, Kolovos et al., 2009, Xing and Stroulia, 2005, Brun and Pierantonio, 2008].

We do not rely on a particular matching algorithm and abstract over concrete representations of difference models. We model the difference between an original metamodel m_o and an evolved version m_e as a set $\Delta(m_o, m_e)$. The elements of this set are three different kinds of changes [Cicchetti et al., 2008]: *Additive changes* $\bot \rightarrow e$, where the evolved metamodel contains an element e which was not present in the original metamodel. *Subtractive changes* $e \rightarrow \bot$, where the evolved metamodel misses an element e which was present in the original metamodel. *Updative changes*, where the evolved metamodel contains an element e' which corresponds to an element e in the original metamodel and the value of a metafeature of e' is different from the value in e. We distinguish three kinds of updates: *Additions* $e \xrightarrow[v]{+mf} e'$, where the multi-valued metafeature mf of e' has an additional value v which was not present in e. *Removals* $e \xrightarrow[v]{-mf} e'$, where the multi-valued metafeature mf of e' is missing a value v which was present in e. *Substitutions* $e \xrightarrow{mf} e'$, where the single-valued metafeature mf of e' has a new value which is different from the value in e. A complete list of possible metamodel changes with respect to our metamodeling formalism is given in the left columns of Figures 5.7 and 5.8.

For validity of difference models, we have three requirements: First, the original and evolved metamodel need to be valid. Second, two changes should not link the same source element with different target elements or the same target element with different source elements. Element merges and

Instance sets

$$N := T \cup F \quad \text{(named elements)}$$
$$T := T_d \cup T_c \quad \text{(types)}$$
$$T_d \quad \text{(data types)}$$
$$T_c \quad \text{(classes)}$$
$$F := F_a \cup F_r \quad \text{(features)}$$
$$F_a \quad \text{(attributes)}$$
$$F_r \quad \text{(references)}$$

Functions and predicates

$$name : N \to String \quad \text{(names)}$$
$$lower : F \to Integer \quad \text{(lower bounds)}$$
$$upper : F \to Integer \quad \text{(upper bounds)}$$
$$type : F \to T \quad \text{(types)}$$
$$opposite : F_r \to F_r \quad \text{(opposite references)}$$
$$abstract : T_c \quad \text{(abstract classes)}$$
$$id : F_a \quad \text{(identifying attributes)}$$
$$composite : F_r \quad \text{(composite references)}$$

Instance subsets

$$C_p(c) \quad \text{(parents)}$$
$$C_c(c) := \{\, c' \in T_c \mid c \in C_p(c') \,\} \quad \text{(children)}$$
$$C_a(c) := C_p(c) \cup \bigcup_{c' \in C_p(c)} C_a(c') \quad \text{(ancestors)}$$
$$C_d(c) := C_c(c) \cup \bigcup_{c' \in C_c(c)} C_d(c') \quad \text{(descendants)}$$
$$C_h(c) := C_a(c) \cup C_d(c) \cup \{c\} \quad \text{(type hierarchy)}$$
$$F(c) \quad \text{(defined features)}$$
$$F_i(c) := F(c) \cup \bigcup_{c' \in C_a(c)} F(c') \quad \text{(defined and inherited features)}$$
$$F_a(c) := F_a \cap F(c) \quad \text{(attributes)}$$
$$F_r(c) := F_r \cap F(c) \quad \text{(references)}$$

Lookup functions

$$\langle cn \rangle := \begin{cases} c & \text{if } c \in T_c \wedge name(c) = cn \\ \bot & \text{else} \end{cases} \qquad \langle cn.fn \rangle := \begin{cases} f & \text{if } f \in F(\langle cn \rangle) \wedge name(f) = fn \\ \bot & \text{else} \end{cases}$$

Figure 5.5 Formal representation of metamodels in terms of sets, functions, and predicates

splits are represented as separate additions and removals and will be reconstructed during detection. Third, we expect changing features not to move between classes, i.e., the class containing a changed feature should be the same or a changed version of the class containing the original feature. We define these requirements formally in Figure 5.6. Note that $s(\delta)$ yields the source element of a change (left-hand side of an arrow) while $t(\delta)$ gives the target element (right-hand side).

Metamodel validity $\vdash m$

$$\forall c, c' \in T_c : name(c) = name(c') \Rightarrow c = c' \quad \textit{(unique class names)}$$
$$\forall c \in T_c : \forall f, f' \in F_i(c) : name(f) = name(f') \Rightarrow f = f' \quad \textit{(unique feature names)}$$
$$\forall c \in T_c : c \notin C_a(c) \quad \textit{(non-cyclic inheritance)}$$
$$\forall f \in F : lower(f) \leq_b upper(f) \wedge upper(f) >_b 0 \quad \textit{(correct bounds)}$$
$$\forall f \in F_a : type(f) \in T_d \quad \textit{(well-typed attributes)}$$
$$\forall f \in F_r : type(f) \in T_c \quad \textit{(well-typed references)}$$
$$\forall f, f' \in F_r : opposite(f) = f' \Leftrightarrow opposite(f') = f \quad \textit{(inverse reflectivity)}$$

Difference model validity $\vdash \Delta(m_o, m_e)$

$$\vdash m_o \wedge \vdash m_e \quad \textit{(source and target validity)}$$
$$\forall \delta, \delta' \in \Delta(m_o, m_e) : t(\delta) = t(\delta') \neq \perp \Rightarrow s(\delta) = s(\delta') \quad \textit{(unique sources)}$$
$$\forall \delta, \delta' \in \Delta(m_o, m_e) : s(\delta) = s(\delta') \neq \perp \Rightarrow t(\delta) = t(\delta') \quad \textit{(unique targets)}$$
$$\forall \delta, \delta' \in \Delta(m_o, m_e) : s(\delta) \in F(s(\delta')) \wedge t(\delta) \neq \perp \Rightarrow t(\delta) \in F(t(\delta')) \quad \textit{(non-moving features)}$$

Evolution trace validity $m_o, m_e \vdash O_1 \dots O_n$

$$\vdash m_o \quad \textit{(source validity)}$$
$$\forall i \in 1, \dots, n : \vdash O_1 \circ \dots \circ O_i(m_o) \quad \textit{(valid applications)}$$
$$O_1 \circ \dots \circ O_n(m_o) = m_e \quad \textit{(target validity)}$$

Figure 5.6 Validity of metamodels, difference models, and evolution traces

5.2.3 *Evolution Traces*

Operator-based approaches to coupled evolution provide a rich set of coupled operators which work at the metamodel level as well as at the model level (Chapter 3). At the metamodel level, a *coupled operator* defines a metamodel transformation capturing a common evolution step. At the model level, it defines a model transformation capturing the corresponding migration. Following the terminology from Chapter 3, we differentiate between primitive and complex operators. *Primitive operators* perform an atomic metamodel evolution step that can not be further subdivided. A list of primitive operators which is complete with respect to our metamodeling formalism is given in the left columns of Figures 5.10 to 5.13. *Complex operators* can be decomposed into a sequence of primitive operators which has the same effect at the metamodel level but typically not at the model level. For example, a feature pull-up can be decomposed into feature deletions in the subclasses followed by a feature creation in the parent class. At the model level, the feature deletions cause the deletion of values in instances of the subclasses while the feature creation requires the introduction of default values in instances of the parent class.

Thus, values for the feature in instances of the subclasses are replaced by default values. This is not an appropriate migration for a feature pull-up which instead requires the preservation of values in instances of the subclasses. We will define only a few complex operators. For an extensive catalog of operators, see Chapter 3.

Each operator has a number of formal parameters like class and feature names. Instantiating these parameters with actual arguments results in an *operator instance* O. This notation hides the actual arguments but is sufficient. We can now model the evolution of a metamodel as a sequence of such operator instances $O_1 \dots O_n$. We call this sequence an *evolution trace*. We distinguish *primitive traces* of only primitive operator instances from *complex traces*. There are three requirements for the validity of an evolution trace with respect to the original and the evolved metamodel. First, we require the original metamodel to be valid. Second, each operator instance should be applicable to the result of its predecessors and should yield a valid metamodel. Third, applying the complete trace should result in the evolved metamodel. Again, we capture these requirements formally in Figure 5.6.

5.3 RECONSTRUCTING PRIMITIVE EVOLUTION

This section shows how to reconstruct a correctly ordered, valid evolution trace from a difference model. First, we provide a mapping from metamodel changes to sequences of primitive operator instances. Second, we define a dependency relation between operator instances based on preconditions of these instances. This allows us to order primitive evolution traces on dependency resulting in valid primitive evolution traces.

5.3.1 *Mapping*

The mapping of changes onto sequences of operator instances is presented in Figures 5.7 and 5.8. The left column shows the metamodel differences. The right column shows the corresponding operator instances. The middle column shows conditions to select the right mapping and to instantiate parameters correctly. Note that we omit conditions of the form $xn = name(x)$. We assume such conditions implicitly whenever there is a pair of variables x and xn. This way, cn refers to the name of a class c, fn to the name of a feature f, and tn to the name of a type t. Figure 5.9 (left) shows the result of the mapping applied to the example difference model in Figure 5.2.

Metamodel Diff	Conditions	Primitive Operator Instances
$\bot \rightarrow c$	$c \in T_c$	**create class** *cn*
	abstract(c)	[**make** *cn* **abstract**]
		[**add super** scn_1 **to** *cn*
	$C_p(c) = \{sc_1, \ldots, sc_k\}$	⋮
		add super scn_k **to** *cn*]
$c \rightarrow \bot$	$c \in T_c$	**drop class** *cn*
$e \xrightarrow{name} e'$	$e \in T_c$	**rename** *en* **to** *en′*
	$e \in F(c)$	**rename** *cn* . *en* **to** *en′*
$c \xrightarrow{isAbstract} c'$	$\neg$*abstract*(c)	**make** *cn* **abstract**
	abstract(c)	**drop** *cn* **abstract**
$c \xrightarrow[sc]{+superTypes} c'$		**add super** *scn* **to** *cn*
$c \xrightarrow[sc]{-superTypes} c'$		**drop super** *scn* **from** *cn*
$\bot \rightarrow f$	$f \in F_a(c) \wedge t = type(f)$	**create feature** *cn* . *fn* **::** *tn*
	$l = lower(f) \wedge l >_b 0$	[**specialize lower** *cn* . *fn* **to** *l*]
	$u = upper(f) \wedge u >_b 1$	[**generalize upper** *cn* . *fn* **to** ub]
	id(f)	[**make** *cn* . *fn* **identifier**]
	$f \in F_r(c) \wedge t = type(f)$	**create feature** *cn* . *fn* $\rightarrow$ *tn*
	$l = lower(f) \wedge l >_b 0$	[**specialize lower** *cn* . *fn* **to** *l*]
	$u = upper(f) \wedge u >_b 1$	[**generalize upper** *cn* . *fn* **to** ub]
	composite(f)	[**make** *cn* . *fn* **composite**]
	$f' = opposite(f)$	[**make** *cn* . *fn* **inverse** *fn′*]
$f \rightarrow \bot$	$f \in F(c)$	**drop feature** *cn* . *fn*

Figure 5.7 Metamodel differences and corresponding sequences of primitive operator instances (1)

5.3.2 *Dependencies between Operator Instances*

Despite the atomicity of primitive operators, not all primitive evolution traces that are valid can be completely executed. Reconsider the left trace in Figure 5.9. Step 5 creates a reference to `TagEvent` at a point where no class `TagEvent` exists. Similarly, step 8 references a non-existent class `Tag` and step 24 attempts to create an inheritance chain with duplicate feature names. Operator instances cannot be applied to all metamodels: Features can only be created in classes that exist, classes can only be created if no equivalently named class is present and a class can only be dropped if it is not in use anywhere else. These restrictions either come directly from the meta-metamodel or from the invariants for valid metamodels. We can translate these restrictions into preconditions. An operator precondition O_{pre} ensures that an oper-

Metamodel Diff	Conditions	Primitive Operator Instances
$f \xrightarrow{lowerBound} f'$	$l = lower(f') \wedge l <_b lower(f)$	**generalize lower** *cn*.*fn* **to** *l*
	$l = lower(f') \wedge l >_b lower(f)$	**specialize lower** *cn*.*fn* **to** *l*
$f \xrightarrow{upperBound} f'$	$u = upper(f') \wedge u >_b upper(f)$	**generalize upper** *cn*.*fn* **to** ub
	$u = upper(f') \wedge u <_b upper(f)$	**specialize upper** *cn*.*fn* **to** ub
$f \xrightarrow{type} f'$	$f \in F(c)$ $f' \in F_a(c') \wedge t = type(f')$ $l = lower(f') \wedge l >_b 0$ $u = upper(f') \wedge u >_b 1$ $id(f')$	**drop feature** *cn*.*fn* **create feature** *cn'*.*fn'* :: *tn* [**specialize lower** *cn'*.*fn'* **to** *l*] [**generalize upper** *cn'*.*fn'* **to** ub] [**make** *cn'*.*fn'* **identifier**]
	$f \in F(c)$ $f' \in F_r(c') \wedge t = type(f')$ $l = lower(f') \wedge l >_b 0$ $u = upper(f') \wedge u >_b 1$ $composite(f')$ $f'' = opposite(f')$	**drop feature** *cn*.*fn* **create feature** *cn'*.*fn'* → *tn* [**specialize lower** *cn'*.*fn'* **to** *l*] [**generalize upper** *cn'*.*fn'* **to** ub] [**make** *cn'*.*fn'* **composite**] [**make** *cn'*.*fn'* **inverse** *fn''*]
$f \xrightarrow{isId} f'$	$\neg id(f)$	**make** *cn*.*fn* **identifier**
	$id(f)$	**drop** *cn*.*fn* **identifier**
$f \xrightarrow{isComposite} f'$	$\neg composite(f)$	**make** *cn*.*fn* **composite**
	$composite(f)$	**drop** *cn*.*fn* **composite**
$f \xrightarrow{opposite} f'$	$f' \in F_r(c) \wedge f'' = opposite(f') \neq \perp$	**make** *cn*.*fn* **inverse** *fn''*
	$f' \in F_r(c) \wedge opposite(f') = \perp$	**drop** *cn*.*fn* **inverse**

Figure 5.8 Metamodel differences and corresponding sequences of primitive operator instances (2)

```
1  create feature Issue.log
     <> Event
2  generalize upper Issue.log to -1
3  make Issue.log composite
4  make Issue.log inverse
     Event.issue
5  create feature Project.log
     <> TagEvent
6  generalize upper Project.log
     to -1
7  make Project.log composite
8  create feature Project.tags
     <> Tag
9  generalize upper Project.tags
     to -1
10 make Project.tags composite
11 add super TagEvent
     to TagAddition
12 drop feature TagAddition.issue
13 drop feature TagAddition.tag
14 drop feature
     TagAddition.timestamp
15 add super TagEvent to TagRemoval
16 drop feature TagRemoval.issue
17 drop feature TagRemoval.tag
18 create class Event
19 create feature Event.issue
     → Issue
20 create feature Event.time
     :: DateTime
21 create feature Event.actor
     → User
22 create class TagEvent : Event
23 create feature TagEvent.tag
     → Tag
24 add super Event to Comment
25 drop feature Comment.issue
26 drop feature Comment.timestamp
27 drop feature Comment.author
```

```
create feature Project.tags
   <> Tag
generalize upper Project.tags
   to -1
make Project.tags composite
drop feature TagAddition.issue
drop feature TagAddition.tag
drop feature
   TagAddition.timestamp
drop feature TagRemoval.issue
drop feature TagRemoval.tag
create class Event
create feature Issue.log
   <> Event
generalize upper Issue.log to -1
make Issue.log composite
create feature Event.issue
   → Issue
make Issue.log inverse
   Event.issue
create feature Event.time
   :: DateTime
create feature Event.actor
   → User
create class TagEvent : Event
create feature Project.log
   <> TagEvent
generalize upper Project.log
   to -1
make Project.log composite
add super TagEvent
   to TagAddition
add super TagEvent to TagRemoval
create feature TagEvent.tag
   → Tag
drop feature Comment.issue
drop feature Comment.timestamp
drop feature Comment.author
add super Event to Comment
```

Figure 5.9 Unordered and dependency-ordered primitives mapped from the difference model

ator instance O can be applied and that the application on a valid metamodel yields again a valid metamodel. Figures 5.10 to 5.13 give a complete overview of the preconditions for primitive operators.

One condition for the validity of a trace of operators is the validity of each intermediate metamodel. Since succeeding operator preconditions ensure this validity, we can redefine trace validity in terms of preconditions:

Evolution trace validity $m_o, m_e \vdash O_1 \dots O_n$

$$O_{1,pre} \wedge \forall_{i \in 2..n} : O_{i,pre}((O_1 \circ \dots \circ O_i - 1)(m)) \qquad (\textit{valid applications})$$

Applying operator instances enables or disables other operator instances. For example, the creation of a class c can enable the creation of a feature $c.f$. The class creation operator validates parts of the precondition of the feature creation operator. To model the effect of an operator instance on conditions,

Primitive Operator	Preconditions	Postconditions
create class *cn*	$\langle cn \rangle = \bot$	$\langle cn \rangle \neq \bot$ $F(\langle cn \rangle) = \emptyset$ $\neg targeted(\langle cn \rangle)$ $\neg abstract(\langle cn \rangle)$
drop class *cn*	$\langle cn \rangle \neq \bot$ $F(\langle cn \rangle) = \emptyset$ $\neg targeted(\langle cn \rangle)$	$\langle cn \rangle \neq \bot$
create feature *cn* . *fn* **::** *tn*	$\langle cn \rangle \neq \bot$ $\forall c' \in C_h(\langle cn \rangle) :$ $\quad \forall f' \in F(c') :$ $\quad\quad name(f') \neq fn$	$\langle cn.fn \rangle \neq \bot$ $\langle cn.fn \rangle \in F_a$
create feature *cn* . *fn* $\rightarrow$ *tn*	$\langle cn \rangle, \langle tn \rangle \neq \bot$ $\forall c' \in C_h(\langle cn \rangle) :$ $\quad \forall f' \in F(c') :$ $\quad\quad name(f') \neq fn$	$\langle cn.fn \rangle \neq \bot$ $\langle cn.fn \rangle \in F_r$ $type(\langle cn.fn \rangle) = \langle tn \rangle$ $\nexists f' :$ $\quad opposite(\langle cn.fn \rangle) = f'$ $\neg composite(\langle cn.fn \rangle)$ $\neg id(\langle cn.fn \rangle)$
drop feature *cn* . *fn*	$\langle cn.fn \rangle \neq \bot$	$\langle cn.fn \rangle = \bot$

Figure 5.10 Pre- and postconditions for structural primitive operators

Primitive Operator	Preconditions	Postconditions
rename class *cn* **to** *cn′*	$\langle cn \rangle \neq \bot$ $\langle cn' \rangle = \bot$	$\langle cn \rangle = \bot$ $\langle cn' \rangle \neq \bot$
rename feature *cn* . *fn* **to** *fn′*	$\langle cn.fn \rangle \neq \bot$ $\forall c' \in C_h(\langle cn \rangle) :$ $\quad \forall f' \in F(c') :$ $\quad\quad name(f') \neq fn'$	$\langle cn.fn \rangle = \bot$ $\langle cn.fn' \rangle \neq \bot$
make *cn* **abstract**	$\langle cn \rangle \neq \bot$ $\neg abstract(\langle cn \rangle)$	$abstract(\langle cn \rangle)$
drop *cn* **abstract**	$\langle cn \rangle \neq \bot$ $abstract(\langle cn \rangle)$	$\neg abstract(\langle cn \rangle)$
add super cn_{sup} **to** cn_{sub}	$\langle cn_{sup} \rangle, \langle cn_{sub} \rangle \neq \bot$ $\langle cn_{sup} \rangle \notin C_h(\langle cn_{sub} \rangle)$ $\forall c \in C_h(\langle cn_{sub} \rangle) :$ $\quad \forall f \in F(c) :$ $\quad\quad \langle cn_{sup}.name(f) \rangle$ $\quad\quad\quad = \bot$	$\langle cn_{sup} \rangle \in$ $\quad C_p(\langle cn_{sub} \rangle)$
drop super cn_{sup} **from** cn_{sub}	$\langle cn_{sub} \rangle, \langle cn_{sup} \rangle \neq \bot$ $\langle cn_{sup} \rangle \in C_p(\langle cn_{sub} \rangle)$	$\langle cn_{sup} \rangle \notin$ $\quad C_p(\langle cn_{sub} \rangle)$

Figure 5.11 Pre- and postconditions for non-structural primitive operators

Primitive Operator	Preconditions	Postconditions
generalize type *cn . fn* **to** *cn′*	$\langle cn.fn \rangle \neq \bot$ $\langle cn' \rangle \neq \bot$ $\langle cn' \rangle \in C_a(\langle cn \rangle)$	$type(\langle cn.fn \rangle)$ $= \langle cn' \rangle$
specialize type *cn . fn* **to** *cn′*	$\langle cn.fn \rangle \neq \bot$ $\langle cn' \rangle \neq \bot$ $\langle cn' \rangle \in C_d(\langle cn \rangle)$	$type(\langle cn.fn \rangle)$ $= \langle cn' \rangle$
generalize upper *cn . fn* **to** u	$\langle cn.fn \rangle \neq \bot$ $u >_B upper(\langle cn.fn \rangle)$	$upper(\langle cn.fn \rangle) = u$
generalize lower *cn . fn* **to** l	$\langle cn.fn \rangle \neq \bot$ $l < lower(\langle cn.fn \rangle)$	$lower(\langle cn.fn \rangle) = l$
specialize upper *cn . fn* **to** u	$\langle cn.fn \rangle \neq \bot$ $u <_B upper(\langle cn.fn \rangle)$ $u \geq_B lower(\langle cn.fn \rangle)$	$upper(\langle cn.fn \rangle) = u$
specialize lower *cn . fn* **to** l	$\langle cn.fn \rangle \neq \bot$ $l > lower(\langle cn.fn \rangle)$ $l \leq upper(\langle cn.fn \rangle)$	$lower(\langle cn.fn \rangle) = l$

Figure 5.12 Pre- and postconditions for generalization and specialization operators

Primitive Operator	Preconditions	Postconditions
make *cn . fn* **inverse** *cn′ . fn′*	$\langle cn.fn \rangle, \langle cn'.fn' \rangle \neq \bot$ $\nexists f$: $opposite(\langle cn.fn \rangle) = f \vee$ $opposite(\langle cn'.fn' \rangle) = f$	$opposite(\langle cn.fn \rangle)$ $= \langle cn'.fn' \rangle$
drop *cn . fn* **inverse**	$\langle cn.fn \rangle \neq \bot$ $\exists f'$: $opposite(\langle cn.fn \rangle) = f'$	$\nexists f'$: $opposite(\langle cn.fn \rangle) = f'$
make *cn . fn* **identifier**	$\langle cn.fn \rangle \neq \bot$ $\neg id(\langle cn.fn \rangle)$	$id(\langle cn.fn \rangle)$
drop *cn . fn* **identifier**	$\langle cn.fn \rangle \neq \bot$ $id(\langle cn.fn \rangle)$	$\neg id(\langle cn.fn \rangle)$
make *cn . fn* **composite**	$\langle cn.fn \rangle \neq \bot$ $\neg composite(\langle cn.fn \rangle)$	$composite(\langle cn.fn \rangle)$
drop *cn . fn* **composite**	$\langle cn.fn \rangle \neq \bot$ $composite(\langle cn.fn \rangle)$	$\neg composite(\langle cn.fn \rangle)$

Figure 5.13 Pre- and postconditions for feature annotation modifying operators

we use a backward transformation description as introduced by Kniesel and Koch [2004]. A backward description O_{bd} is a function that, given a condition C to be checked after applying an operator instance O, computes a semantically equivalent condition that can be checked before applying O: $O_{bd}(C)(m) \Leftrightarrow C(O(m))$. We define backward description functions for the primitive operators based on the postconditions specified in Figures 5.10 to 5.13: A backward description rewrites any clause in a condition C with *true*, when it is implied by the operator postcondition. Using these backward description functions, we can define enabling and disabling operator instances as dependencies: Operator instance O_2 depends on operator instance O_1, if the backward description of operator O_1 changes the precondition of O_2. Typically, operator instances are dependent if they affect or target the same metamodel element. Examples are creation and deletion of the same class, creation of a class and addition of a feature to this class, and creation of a class and of a reference to this class.

5.3.3 *Dependency Ordering*

To ensure trace validity, we need to ensure that the preconditions of all operator instances are enabled and thus all dependencies are satisfied. The dependency relation between operator instances is a partial order on these instances. To establish validity, we apply the partial dependency order to the trace and make the ordering complete by arbitrarily ordering independent operator instances. Figure 5.9 (right) shows the dependency-ordered trace of primitive operators for the running example.

5.4 RECONSTRUCTING COMPLEX EVOLUTION

This section shows how to reconstruct valid complex evolution traces from valid primitive traces. First, we provide patterns for mapping sequences of primitive operator instances to complex operator instances. Second, we discuss how to reorder evolution traces without breaking their validity. This allows us to reorder traces into different normal forms in which the patterns can be detected easily and be replaced by complex operator instances.

5.4.1 *Patterns*

A complex operator instance comprises a sequence of (less-complex) operator instances. We can use patterns on these sequences to detect complex operator instances. Figure 5.14 lists the decompositions and conditions for two complex operators working across inheritance. When read from left to right, it shows how to decompose a complex operator instance, when read from right

Complex Operator	Conditions	Equivalent Trace
pull up feature *cn* . *fn*	$C_c(\langle cn \rangle) = \{c_1, .., c_k\}$ $\langle cn_1.fn \rangle \equiv_F .. \equiv_F \langle cn_k.fn \rangle$	**drop feature** cn_1 . *fn* ... **drop feature** cn_k . *fn*
	$t = type(\langle cn_1.fn \rangle) \wedge t \in T_d$	**create feature** *cn* . *fn* :: *tn*
	$l = lower(\langle cn_1.fn \rangle) \wedge l >_b 0$	[**spec. lower** *cn* . *fn* **to** *l*]
	$u = upper(\langle cn_1.fn \rangle) \wedge u >_b 1$	[**gen. upper** *cn* . *fn* **to** ub]
	$id(\langle cn_1.fn \rangle)$	[**make** *cn* . *fn* **identifier**]
	$C_c(\langle cn \rangle) = \{c_1, .., c_k\}$ $\langle cn_1.fn \rangle \equiv_F .. \equiv_F \langle cn_k.fn \rangle$	**drop feature** cn_1 . *fn* ... **drop feature** cn_k . *fn*
	$t = type(\langle cn_1.fn \rangle) \wedge t \in T_c$	**create feature** *cn* . *fn* → *tn*
	$l = lower(\langle cn_1.fn \rangle) \wedge l >_b 0$	[**spec. lower** *cn* . *fn* **to** *l*]
	$u = upper(\langle cn_1.fn \rangle) \wedge u >_b 1$	[**gen. upper** *cn* . *fn* **to** ub]
	$composite(\langle cn_1.fn \rangle)$	[**make** *cn* . *fn* **composite**]
	$f' = opposite(\langle cn_1.fn \rangle)$	[**make** *cn* . *fn* **inverse** fn']
extract super class *cn* { fn_1 , ..., fn_j } **from** cn_1 , ..., cn_k	*true*	**create class** *cn* **add super** *cn* **to** cn_1 ... **add super** *cn* **to** cn_k **pull up feature** *cn* . fn_1 ... **pull up feature** *cn* . fn_j
fold super class *cn* **from** cn'	$F_i(\langle cn \rangle) = \{f_1, .., f_k\}$ $\forall i = 1..k : \langle cn'.fn_i \rangle \equiv_F f_i$	**drop feature** cn' . fn_1 ... **drop feature** cn' . fn_k **add super** *cn* **to** cn'

Figure 5.14 (De-)Composition patterns for complex operators

to left, it defines its detection pattern. Given a source metamodel m, we can recursively decompose an operator instance O into a sequence of primitive operator instances $\lfloor O \rfloor_m = P_1 \dots P_n$. As a precondition, a complex operator instance needs to fulfill the backward descriptions of the preconditions of these primitives. But typically this is not enough and an operator instance requires an additional precondition.We highlight these additional preconditions with a box in Figure 5.14.

5.4.2 *Reordering traces*

Figure 5.15 shows an excerpt of Figure 5.9 (right). It displays the extraction of super class `Event` from class `Comment`. Operator ordering is still determined by the dependency ordering from the previous section. To simplify the example, we changed the operator on `Comment.author` to work on `Comment.actor`. We will look at `author` and the complete trace in the next section. Consider applying the patterns from Figure 5.14. There is no consecutive sequence of operator instances satisfying any of the patterns. We could detect pulling up

```
1 create class Event
2 create feature Event.timestamp
    :: DateTime
3 create feature Event.actor
    → User
4 drop feature Comment.timestamp
5 drop feature Comment.actor
6 add super Event to Comment
```

```
1 create class Event
6 add super Event to Comment
5 drop feature Comment.actor
3 create feature Event.actor
    → User
4 drop feature Comment.timestamp
2 create feature Event.timestamp
    :: DateTime
```

Figure 5.15 Excerpt of dependency-ordered operators in Figure 5.9

feature `timestamp` in instances 2 and 4, yet where do we put the detected complex operator: at position 2 or at position 4?

Detection patterns typically cannot be applied directly. Instead, traces need to be reordered to find consecutive instances of a pattern. Dependency ordering is partial and therefore leaves room for swapping independent operators. In the example, we can swap 2 and 3 as they work on different features; 2 and 4 as they work on different types; 2 and 5 which also work on different types; 4 and 5 which work on different features; 3 and 5, which work on different types; and finally, we can repeatedly swap 6 to follow operator 1, as all features that are created are dropped from the inheritance chain first. The reordered trace is shown at the right of Figure 5.15. We can now apply the patterns for pulling up `timestamp` and `actor`. Subsequently, we see the pattern for class extraction emerge, which yields a super class extraction of `Event {timestamp, actor}` from `Comment` and `TagEvent`.

5.4.3 *Normal forms*

In the example, we carefully swapped operators. Not only did we avoid swapping dependent operators (as to preserve trace validity), we also chose swaps, which gave us a detectable pattern. In particular, we focused on obtaining a consecutive feature creation and drop, of features that only differ in position in the inheritance chain. A set of swap rules can bring an evolution trace into a format most suitable for detecting a pattern. In general, these rules obey the dependency relation. However, some dependent instances can still be swapped by adjusting their parameters. For example, **`rename class`** `A` **`to`** `B` and **`create feature`** `B.f...` can be swapped to: **`create feature`** `A.f...` and **`rename class`** `A` **`to`** `B`.

Repeated application of a set of swap rules will result in a normal form defined by this set. Each normal form targets to bring potential components of a pattern together and to satisfy the operator precondition. For example, to detect a feature pull up, we rely on feature similarity: Class creations and super additions get precedence over other operators. Feature creations, changes and drops are sorted on feature name, type and modifiers. Class drops and destructive updates on the inheritance chain go last. Different patterns need

different trace characteristics and thus different normal forms. But operators with similar kinds of patterns can share normal forms.

5.5 RECONSTRUCTING MASKED OPERATOR INSTANCES

In this section, we extend the detection to deal not only with complete but also partial patterns. First, we revisit the problem of operator interference and study its effects on detection. Second, we show how to complete partial patterns by the additions of operator instances in a validity preserving fashion. This allows us to detect operator instances which patterns are partially or even completely hidden by other instances.

5.5.1 *Masked Operators*

We reconsider the running example from Figure 5.3. During evolution, several features of the classes `TagAddition` and `TagRemoval` were extracted into a new super class `TagEvent`. In order to extract the feature `timestamp` it needs to be present in both `TagAddition` and `TagRemoval` . Yet, it is not. As a human, we deduce that `timestamp` must have been added in the process of extracting `TagEvent`. There is, however, no explicit record of such feature creation. Detection will therefore fail. Later in the evolution, when extracting the class `Event`, we seek to pull up a feature `actor`. The class `Comment`, which we are extracting from, only offers a feature `author`. Again as a human, we assume that `author` must have been renamed to `actor` (like we did in the previous section), yet this operation is not present in the original evolution trace. Similarly, we have to create the feature `actor` in `TagEvent` before extracting `Event` and rename the feature `timestamp` to `time` after extracting `Event` to yield the target metamodel. Each of these operations has no record in the difference set obtained from the matching algorithm.

When evolutions become more complex, individual evolution steps no longer need to have an explicit effect on the target metamodel and are therefore not explicit in the matching result. An operator instance can hide or even undo parts of the effect of another instance. This is a strong variant of dependency, which we call masking. A primitive operator P_1 masks another primitive operator P_2 when composition of the two can be captured in a third primitive operator P_3. More generally, we define masking for arbitrary instances as the presence of a mask in decompositions:

$$P_1 \; masks_m \; P_2 \Leftrightarrow \exists P_3 : (P_1 \circ P_2)(m) = P_3(m)$$
$$O_1 \; masks_m \; O_2 \Leftrightarrow \exists P_1 \in \lfloor O_1 \rfloor_m : \exists P_2 \in \lfloor O_2 \rfloor_m : P_1 \; masks \; P_2$$

Most operators can be masked by renaming. All operators are masked by their inverses, in which case O_3 is the identity operator. Extraction of class `TagEvent` in the running example masks extraction of class `Event`. Note that a trace obtained from a valid difference model will only contain masks that involve complex operators.

5.5.2 *Masked Detection Rules*

When a primitive masks another primitive, the effect is completely hidden from the target metamodel. There is no information (implicit or explicit) that could lead back to the masked operator. However, when a complex operator is masked, generally only part of the effect of the decomposition is hidden. Using the remaining information, the complex operator can be reconstructed. We detect masked complex operators by automatically filling the gaps caused by masks.

Detection of masked operator instances follows a trace rewriting approach similar to the original detection of complex operator instances: We try to rewrite a sequence of operator instances into another sequence which has the same effect on the metamodel. Instead of checking the operator precondition in a pattern, like we did in the previous section, we now *ensure* the precondition by deducing a suitable sequence to rewrite to. We now discuss how to derive a detection rule for a masked complex operator instance, e.g., for pulling up an attribute $cn_{sup}.fn$. Its decomposition is the following:

drop feature cn_{sub} `l` . fn ... **drop feature** cn_{sub} `i` . fn **create feature** cn_{sup} . fn	[**specialize lower** cn_{sup} . fn **to** `l`] [**generalize upper** cn_{sup} . fn **to** `u`] [**make** cn . fn **identifier**]

From the decomposition we choose a trigger, which tells us that there may have been a feature pull up. We choose one of the feature drops (number `x`). We use the trigger as a pattern on the left-hand side of a rewrite rule and assume on the right-hand side that there must have been a feature pull up:

drop feature cn_{sub} `x` . fn	... **pull up feature** cn_{sup} . fn ...

When the dots are left blank, application of the left-hand side to a metamodel does not have an equivalent effect as application of the right-hand side. Instead, we fill the dots, to establish equivalence. The left set of dots ensures that the pull up feature operator can be applied, i.e., its precondition is satisfied. The right set of dots ensures that application of the trace is equivalent to application of the left-hand side of the rewrite rule. Both sets of dots are filled

in using inverses of the operators found in the pattern. The left set of dots is replaced by inverses of each of the primitive operators whose precondition is not already satisfied. For pull up feature, we create features in all sibling classes if they do not exist yet and remove the target feature if it already exists. The right set of dots is replaced by inverses that neutralize the effect of the complex operator and bring the metamodel back to its original state. For pull up feature, we need to create all sibling features, which were present beforehand, as these were deleted during pull up and we need to drop the target feature if it was not present beforehand. The rewrite rule for detecting a masked feature pull up is (leaving out the operations on feature modifiers, for simplicity):

drop feature cn_{sub} k . *fn*	**create feature** cn_{sib} n1 . *fn* ... **create feature** cn_{sib} nj . *fn* [**drop feature** cn_{sup} . *fn*] **pull up feature** cn_{sup} . *fn* **create feature** cn_{sib} e1 . *fn* ... **create feature** cn_{sib} ek . *fn* [**drop feature** cn_{sup} . *fn*]

In which cn_{sup} is chosen arbitrarily from $C_p(cn_{sub}k)$, $cn_{sib}n$ is the set of all sibling classes which do not have a feature named *fn* and thus need to obtain the feature to pull it up. $cn_{sib}e$ is the set of all sibling classes which do have a feature named *fn* and thus need to be reequipped with *fn* to neutralize the effect of pulling it up. The feature drops are conditional. The first drop should be present if $\langle cn_{sup}.fn \rangle \neq \bot$ and the latter should be present if $\langle cn_{sup}.fn \rangle = \bot$. In addition to the pattern on the left-hand side of a rewrite rule for a masked complex operator O, a rewrite rule is also conditioned by the O_{cpre}. It is checked in addition to the trigger.

For feature pull up, the operator precondition O_{cpre} ensures presence of an inheritance chain between cn_{sub} and cn_{sup}. The metamodel invariants ensure feature names uniqueness across inheritance. The precondition of the trigger ensures *fn* exists in cn_{sub}. Therefore, *fn* cannot exist in cn_{sup}. The rewrite rule for feature pull up can thus be simplified by removing the top drop feature and always using the bottom drop feature.

Using the presented approach, we can derive masked detection rules for any complex operator. By definition, such rules expand the trace. To find a suitable evolution, we need to compact the trace again. Firstly, we can rewrite any pair of inverse operators to the identity function, as their effect on the metamodel is cancelled out and they are unlikely to have been part of the original evolution. Secondly, we combine a creation and deletion of two features, which only differ by name into a feature rename. This allows us to detect complex operators, which are masked by a rename, such as a pull up of feature f, followed by a rename of f to f'. Combining rules for inverses

```
1 drop feature                  → pull up feature TagEvent.issue
     TagAddition.issue            drop feature TagEvent.issue
                                  create feature TagRemoval.issue

2 create feature                → identity
     TagRemoval.issue
  drop feature TagRemoval.issue

3 create feature                → pull up feature TagEvent.tag
     TagEvent.tag → Tag           create feature TagRemoval.timestamp
  drop feature TagAddition.tag    pull up feature TagEvent.timestamp
  drop feature                    drop feature TagEvent.timestamp
     TagAddition.timestamp
  drop feature TagRemoval.tag

4 pull up feature               → drop class TagEvent
     TagEvent.issue               drop super TagEvent from TagAddition
                                  drop super TagEvent from TagRemoval
                                  extract super TagEvent
                                     {issue, tag, timestamp}
                                     from {TagAddition, TagRemoval}
                                  push down feature TagEvent.tag
                                  push down feature TagEvent.timestamp
```

Figure 5.16 Masked detection applied to running example

requires a normal form grouping on operator category and the renaming rule requires a normal form on feature similarity.

5.5.3 *Applying Masked Detection Rules*

We apply masked detection rules to the running example. Figure 5.16 shows the intermediate steps. Step 1 applies feature pull up detection to the feature `TagAddition.issue`. After normalizing the trace, we apply an inverse pattern to creation and drop of `TagRemoval.issue` and reduce the trace (step 2). `TagEvent.issue` is not reduced yet. It will be used later as a component of extracting class `Event`. Next, we repeat steps 1 and 2 by pulling up `tag` and `timestamp` (step 3). Subsequently, the pull up of `TagEvent.issue` triggers detection of super class extraction of `TagEvent` in step 4. The drop class, both super drops and both feature push downs are subsequently neutralized by a class creation, super additions and pull ups respectively. We then repeat detection of super class extraction for `Event`, using the rename pattern to neutralize create and drops of `timestamp` - `time` and `author` - `actor`. Finally, we get the result shown in Figure 5.3.

The regular rewrite rules, which we defined in the previous section all reduced the number of operators in the trace. Furthermore, we did not consider overlapping (interfering) complex operators. These two assumptions enabled fast detection. The rules for detecting masked operators, on the other hand, can increase the size of the trace. For example, the feature pull up pattern increases the trace by the number of occurrences of this feature in sibling classes

plus one (for dropping the pulled up feature). Furthermore, for each trace, several rules may be applicable at different positions in the trace. To find a solution, we therefore use a backtracking approach. Each backtracking step tries to apply each of the rules to a trace, yielding zero or more new traces, to which rule application is applied recursively.

5.6 RELATED WORK

Research on difference detection is found in differencing textual documents, matching structured artefacts and detection of complex evolution. Text differencing is ignorant of structure or semantics. We discuss related work on matching and complex detection.

5.6.1 *Matching*

A matching algorithm detects evolution between two artefacts, by linking elements of one artefact to elements of the other. Links are either established based on similarity, or using an origin tracking technique such as persistent identifiers. Links are concerned with one element in each artefact. Consequently, matching approaches detect atomic changes. They do not offer support for detecting complex changes. Nevertheless, we discuss them as potential input to our approach. Matching has received attention in the domains of UML, source code reorganization, database schemas and metamodels.

In the domain of UML, Ohst et al. [2003] first proposed a solution to compare two UML documents. They compare XML files and use persistent ids for matching. The matching algorithm can detect intra-element changes (such as renames) and structural changes, such as element creates, deletes and moves. Later work by Xing and Stroulia [2005] presents UMLDiff, a matching tool set using similarity metrics instead of persistent ids to establish links. Similar to the work of Ohst et al., UMLDiff detects element additions, deletions, renames and moves. Lin et al. [2007] propose a generalization of the work of Xing and Stroulia, which is not restricted to UML models, but uses domain specific models as input instead.

In the domain of source code reorganization, Demeyer et al. [2000] proposes to find refactorings using change metrics. Later work by Tu and Godfrey [2002] uses statistical data and metrics to match evolved software architectures, a process referred to as origin analysis. The approach has a strong focus on understanding and visualizing software evolution. The work on evolving architectures is extended by Godfrey and Zou [2005], by adding detection of merged and split source code entities.

In schema matching, a body of work exists, which generally offers a basis for the other works presented in this section. Rahm et al. and later Shvaiko et

al. present surveys on schema matching [Rahm and Bernstein, 2001, Shvaiko and Euzenat, 2005]. Sun and Rose [2003] present a study of schema matching techniques.

Lopes et al. [2006] consider schema matching applied in the context of model-driven engineering, but propose a new matching algorithm for models. Instead, Falleri et al. [2008] take the existing similarity flooding algorithm from the field of schema matching and apply it to metamodels. Work by Del Fabro and Valduriez [2007] and by Kolovos et al. [2009] propose new matching algorithms to the modeling domain. Finally, EMFCompare offers metamodel independent model comparison in the Eclipse Modeling Framework (EMF). EMFCompare is presented by Brun and Pierantonio [2008]. They distinguish calculation, representation and visualization as relevant aspects in comparing models. The approach uses heuristic-based matching and differencing. Both matching and differencing are pluggable and can thereby be adjusted to a specific domain.

In general, matching approaches are concerned with at most one element in each artefact. Consequently, regular matching approaches detect atomic changes. Some approaches are augmented to detect slightly more complex changes, detecting, inter-element moves, element merges or element splits. In practice, evolution is not bound to merely atomic changes. To obtain the actual evolution trace and not to overwhelm the developer with changes that did not actually take place [Brun and Pierantonio, 2008], complex changes need to be taken into account.

5.6.2 *Complex Detection*

Detection of complex operators has received significantly less attention in research than matching. Cicchetti et al. [2008] discuss an approach for model migration along complex metamodel evolution. They obtain the complex evolution from an arbitrary matching algorithm, but do not offer such an algorithm on their own. Instead, they emphasize the need for a matching algorithm able to detect complex evolution. Our approach fulfills this need. Later work of Cicchetti addresses the problem of dependencies between evolution steps [Cicchetti et al., 2009]. Since their work focuses only on dependency ordering but not on complex operator detection, they specify operator dependency only statically in terms of the metamodeling formalism. This is too restrictive for the detection of complex operators since it limits possible reorderings dramatically. By defining dependency only in the context of an actual metamodel, our approach enables reordering into various normal forms which allow for the detection of complex operators.

Garcés et al. [2009] present an approach to automatically derive a model migration from metamodel differences. The difference computation uses heuristics to detect also complex changes. Each heuristic refines the matching model, and is implemented by a model transformation in ATL. The trans-

formation rules for detecting complex changes are similar to the patterns presented in Section 5.4. There are different kinds of heuristics. Creation heuristics create an initial matching model from two metamodel versions. Similarity heuristics decorate the equivalences of a matching model with annotations about the similarity. Filtering heuristics remove unwanted equivalences from the matching. Differentiation heuristics identify differences between metamodel elements in the match. Rewriting heuristics structure equivalences and differences, and detect complex changes. The approach does not cover operator dependencies, was not able to detect complex changes in a Java case study, and does not address operator masking.

5.7 IMPLEMENTATION

We implemented our approach prototypically in the tool set *Acoda*[1], a data model evolution tool for WebDSL [Visser, 2008a], which is a DSL for web applications. Acoda offers an Eclipse plugin to seamlessly integrate into regular development. The plugin offers editor support for evolution traces (such as syntax highlighting, instant error marking and content completion); generation of SQL migration code; application of migrations to a database; and the evolution detection presented in this chapter. The implementation uses an existing data model matching algorithm. We relied on rewrite rules in Stratego [Visser, 2004] to specify each step of the reconstruction algorithm, i.e., mapping data model changes to primitive operators, dependency ordering, normal form rewriting, complex operator detection and masked operator detection. Acoda's Eclipse plugin presents different evolution traces to the user who can select and potentially modify the best match.

5.8 DISCUSSION

5.8.1 *Metamodeling Formalism*

In this chapter, we focus only on core metamodeling constructs that are most interesting for coupled evolution of metamodels and models. Concrete metamodeling formalisms like Ecore [Steinberg et al., 2009] or MOF [Object Management Group, 2006] provide additional metamodeling constructs like packages, interfaces, operations, derived features, volatile features, or annotations. Since our approach allows for extension, we can add support for these constructs. Therefore, we need to provide additional primitive operators, define their preconditions, extend existing preconditions with respect to new invari-

[1] http://swerl.tudelft.nl/bin/view/Acoda

ants, derive additional complex operators, and define detection patterns for them.

5.8.2 *Trace Selection*

Involving the user in the selection process prevents complete automation, but with a rich set of supported coupled operators, detection is likely to yield several suitable traces. Only the user can decide which migration is correct. We can assist this decision by presenting migrations of example models. Conversely, the user can assist the detection by giving examples for original and migrated models. The detection can then drop all traces which cannot reproduce the examples. Additionally, the user may choose to only consider information-preserving traces, thereby narrowing down the set of suitable traces.

5.8.3 *Completeness*

The set of primitive operators guarantees completeness at the metamodel level as it allows us to evolve any source metamodel to any target metamodel. Completeness at the model level is not feasible since it would imply that we can detect any model transformation between the instances of two arbitrary metamodels. Yet, we can add more complex coupled operators to our detection. This increases the search space for both the user and for the detection. As for the user, we have a tradeoff between completeness and usability. There will be many similar operators with minor differences in their migration. Understanding and distinguishing operators becomes harder. In a number of real-life case studies, we identified the most common operators (Chapter 3). We propose to support only the detection of these operators and to leave rare cases to the user. As for the detection, supporting more complex operators increases the search space and we have a tradeoff between completeness and performance.

5.8.4 *Performance*

Besides the number of supported complex operators, detection performance is influenced by evolution size and mask depth, not by metamodel size, which only affects the matching process. The GMF case study [Herrmannsdoerfer et al., 2010a] showed us that a larger distance between original and evolved metamodel gives less precise results from a matching algorithm making it more unlikely to still detect a good evolution trace. On the other hand, we found that the evolution between two commits to the repository could mostly be captured by 20 evolution steps.

Pruning the Search Space. Masked detection rules can increase the length of the trace. It is the potential of reducing this length again, which confirms that application of the detection rule was indeed a correct choice. As we know that in reality, traces are of limited length, unbound expansion has no purpose. We therefore limit the search space on trace length.

A trace length bound needs to be chosen carefully. If too high, it makes the search space too large, if too low, it may exclude the sought trace. In practice, the length bound mostly determines the number of nested masks we can detect. Considering deeply nested masks are unlikely to occur in practice, a lower bound (close to the original trace length) is more likely to be suitable than a higher one. Using the masked detection rules, there are many ways of rewriting one trace into another. It is therefore likely that we will come across equivalent traces during search. Excluding these from a recursive descent, prunes the search space significantly.

Triggering Carefully. The search space is defined by the original trace and by the rewrite rules we can apply. There is variability in the presented approach of deriving masked detection rules. A variability, which primarily determines the search algorithm efficiency.

Firstly, the choice of trigger determines how often a rewrite rule can be applied. We chose feature drop for pull up. As feature drops are common, the pull up detection decreases performance. On the other hand, the triggers chosen for fold and extract super class are uncommon, thus having little impact. Most complex operators contain uncommon components, therefore, most detection rules can have uncommon triggers. Additionally, we assumed that except for the trigger, the complete operator is masked. This is flexible, but unlikely in practice. Choosing multiple triggers improves performance significantly.

Case Study. A preliminary case study of Acoda on part of the evolution of Researchr (see Appendix B), a publication management system, showed the applicability of the presented detection. Traces in Researchr between subsequent repository commits are short, hence we applied the detection to steps of ten subsequent commits, which yields traces up to 52 steps in length.

A detection run generally takes several seconds and is significantly shortened when reducing the number of commits considered in a single detection run.

5.9 CONCLUSION

In coupled metamodel evolution, a metamodel evolution is mapped onto a suitable model migration to preserve the conformance relation between model and metamodel. Tools to perform this mapping exist (e.g., [Wachsmuth, 2007b, Hößler et al., 2005, Rose et al., 2010]) yet they require an explicit evo-

lution specification. Since evolution is generally applied implicitly by editing a metamodel, explicit evolution needs to be reconstructed. A typical first step in finding evolution is metamodel matching (or differencing), in which a set of differences is derived by comparing the two metamodel versions. Although these differences describe what changed in the metamodel, they neither offer knowledge on how the metamodel changed, nor on what the developer intended in his change, both of which are (partially) captured in an evolution sequence. To construct a suitable migration, we need to know the metamodel evolution.

This chapter presents a technique to reconstruct metamodel evolution, given two metamodel versions, using the result of a metamodel matching algorithm as input. The chapter provides solutions to three major problems faced in reconstruction. Firstly, it offers an approach to resolve evolution dependencies. While metamodel differences can be applied in any order, metamodel evolution steps (and their corresponding model migrations) show dependencies, which need to be resolved upon reconstruction. Secondly, while metamodel differences show atomic changes, evolution steps are generally complex and comprise multiple atomic differences. Thirdly, different evolution steps can hide, change or partially undo the effect of other evolution operators. Their presence in the to be reconstructed evolution is thereby obfuscated, or hidden completely in the metamodel changes.

This chapter provided an approach to reconstruct complex metamodel evolution steps, from a set of metamodel differences. It first showed how to map metamodel changes onto primitive evolution operators, then it solved the evolution operator dependency problem by ordering operators based on their pre- and postconditions, yielding a valid evolution trace. Next, it discussed how to reconstruct complex evolution steps by using the variability in the operator dependency ordering and various normal forms. Finally, it showed how to extend this approach to detect evolution patterns, which are, by interference, partially or completely hidden from the metamodel differences.

Additionally, the chapter formalized the concepts of metamodel evolution and defined pre- and postconditions for the various operators found in Chapter 3. The reconstruction approach is implemented in Acoda and applied to the Researchr case study (see Appendix B), in which it was used to reconstruct the complete Researchr evolution trace.

ACKNOWLEDGMENTS

This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*.

Heterogeneous Coupled Evolution of Software Languages

6

ABSTRACT

As most software artifacts, meta-models can evolve. Their evolution requires conforming models to co-evolve along with them. Coupled evolution supports this. Its applicability is not limited to the modeling domain. Other domains are for example evolving grammars or database schemas. Existing approaches to coupled evolution focus on a single, homogeneous domain. They solve the co-evolution problems locally and repeatedly. In this chapter we present a systematic, heterogeneous approach to coupled evolution. It provides an automatically derived domain specific transformation language; a means of executing transformations at the top level; a derivation of the coupled bottom level transformation; and it allows for generic abstractions from elementary transformations. The feasibility of the architecture is evaluated by applying it to data model evolution.

6.1 INTRODUCTION

Data models are an integral part of software development. They define the structure of data that is processed by an application and the schema of a database. Running applications produce and store data that conforms to the data model.

Due to changing requirements or maintenance, data models need to evolve. This process is known as *format evolution* [Lämmel and Lohmann, 2001]. As a consequence of evolution, stored data no longer conforms to the evolved data model and can thereby become useless to the evolved application. To continue using existing data, the data needs to be transformed to reflect the evolution, which is an instance of coupled evolution.

PROBLEM. Coupled evolution does not only apply to data transformation, but is a reoccurring problem in computer science [Lämmel, 2004]. Models need to be transformed to reflect evolution in their meta-models [Gruschko et al., 2007, Favre, 2003, Wachsmuth, 2007b]. Programs need to be transformed when the programming languages (or domain specific language) they have been written in evolves [Pizka and Jürgens, 2007b]. And a data model itself needs to be transformed to reflect evolution in the data modeling language

(e.g. UML). We unify these scenarios by considering evolving software languages and transformation of sentences in these languages.

Current approaches to support coupled evolution of software languages are homogeneous. They solve the problem in a specific domain. They repeatedly implement the coupled evolution structure and solve problems common to coupled evolution locally. Instead, we would like a systematic approach to realize heterogeneous coupled evolution for any scenario of software language evolution.

CONTRIBUTION. In this chapter, we present two generalizations over coupled software language evolution scenarios and introduce the concept of heterogeneous coupled evolution. To enable the generalization, we present an architecture to support heterogeneous coupled evolution of software languages. We have implemented a tool to support the architecture. It generates a domain specific transformation language (DSTL) for an arbitrary software language domain. It generates an interpreter of transformations defined in the DSTL. And it supports generic abstraction from the basic transformations that are defined in the DSTL. We illustrate the architecture and tool by elaborating their application to coupled data model evolution.

OUTLINE. The chapter is structured as follows: In Section 6.2 we briefly introduce data model evolution and its context. In Section 6.3 we elaborate on coupled data evolution by defining data model transformations and deriving data transformations to reflect these. In Section 6.4 we generalize over the different scenarios of coupled software language evolution. In Section 6.5, we discuss the architecture to support heterogeneous coupled evolutions. Section 6.6 discusses related work. Section 6.7 concludes.

6.2 DATA MODEL EVOLUTION

Data models describe the structure of data that is processed and stored by an application. As example application we consider a Wiki. It consists of web pages, users to edit these and webs, which are collections of pages that cover a similar topic. The corresponding data model is shown in Figure 6.1 (left).

Changing requirements and maintenance cause data models to evolve along with the application they are set in, a process known as format evolution [Lämmel and Lohmann, 2001]. Consider for example the shift from a user-based to a group-based access control security mechanism and the addition of page topics. The new data model to support these is shown in Figure 6.1 (right).

Since the Wiki is a running application during evolution, it has stored pages, users and webs. Such data conforms to some version of the data model. To prevent the loss of data when the data model changes, the data needs to be transformed to reflect these changes. Figure 6.2 (left) shows the

```
entity User {
   name    :: String
}

entity Web {
   admin   :: Set of User
   topic   :: String
}

entity Page {
   content :: String
   date    :: Date
   author  :: User
   web     :: Web
}
```

```
entity User {
   name    :: String
}

entity Web {
   admin   :: Set of Group
   topic   :: String
}

entity Page {
   content :: String
   date    :: Date
   author  :: User
   web     :: Web
   topic   :: String
}

entity Group {
   name    :: String
   members :: Set of User
}
```

Figure 6.1 Data model original (left) and evolved (right) versions

process graphically. At the top level, we see two versions of the data model. At the bottom level we see the stored data and the transformation needed to reflect the data model change. The vertical lines indicate conformance. The dashed arrow indicates the changes applied to the data model and is usually performed manually by editing the data model. The transform arrow on the other hand requires tools for database transformations, as it is usually too much of an effort to reenter all data manually.

6.3 COUPLED DATA EVOLUTION

When a data model evolves, stored data may no longer conform. In practice the data is usually no longer usable. To continue to use the data, we need to reflect the data model changes in a transformation of the stored data. Supporting a single data model change, requires a significant effort. Supporting data model changes in an evolution process, requires repeated data transformations. If these transformations are defined manually, this becomes costly and holds back the development process.

Coupled data evolution automates the data transformation process. It is based upon the assumption that the data model transformation and the data transformation are related. The concept of coupled data evolution is shown in Figure 6.2 (right). A coupled evolution application consists of two components, which are represented by the top two arrows: (1) A definition of the data model transformation (the evolve arrow) and (2) a mapping from data model transformation to data transformation (the vertical arrow). The first

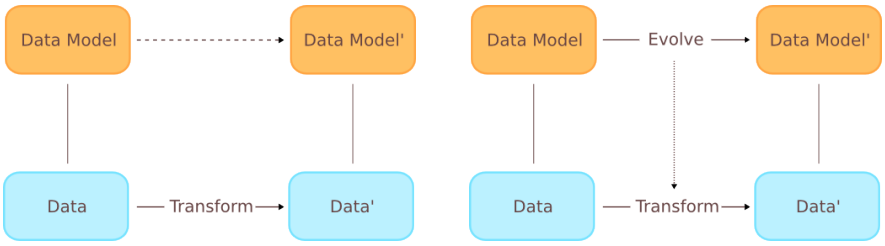


Figure 6.2 Data model changes (left) and coupled data evolution (right)

needs to be specified for each change of the data model, whereas the second is typically defined once for the modeling formalism.

The questions that remain are: How to define the data model transformation and how to derive a data transformation. In Chapter 2 we discussed various approaches to formalize both. In this section, we take a closer look at the two, using the Wiki data model as a running example. We introduce a language for defining data model transformations and show a mapping that targets a broad set of databases.

6.3.1 *Defining Data Model Transformations*

We distinguish two methods to formalize transformations for coupled evolution [Gruschko et al., 2007]: Specify the difference between the two versions of the data model or specify a trace of elementary transformations defining how the new version is obtained from the old version. Both have advantages and disadvantages. We choose the second because it allows us to define the mapping, as we will see in the next section.

Our data models are relatively basic. They consist of entities with a name and properties. Each property has a name and a type. Entities, properties, types and names are the constructs of our data model. A fairly limited set of constructs. Consequently, the number of elementary transformations we can perform on our model (on our constructs) is fairly small. We identify:

- adding or removing entities
- changing the name of an entity
- adding or removing properties
- changing the name of a property
- substituting the type of a property
- substituting the type of a set

For example, we define the addition of a new entity "Group" as follows:

```
add Group {
   name        :: String
   description :: String
   members     :: Set of User   }
```

Although the addition is in itself a valid transformation, in general the elementary actions above do not have sufficient meaning on their own. One cannot change the name of an entity without knowing which entity is being referred and a substitution of a type should not only specify the new type, but also the old type that is being replaced. The transformations above are local and need a location to make them executable to a specific model.

The representation of a data model is tree-structured. The root node is the model itself. Its direct children are entities, which have in turn properties as their children and so on. We define a unique location in a data model by specifying a path from the root node. For example: "Entity *Group* - Property *members* - Type" indicates the type of the `members` collection in the `Group` entity that we have just added. In a similar way, we define locations for our transformations using the APath [Janssen, 2005] notation, which is based on the XPath language [Clark et al., 1999]. APath expressions consist of a /-separated list of construct names. The above would be written as:

```
Entity [Id="Group"] / Property [Id="members"] / Type
```

The `[...]`-part indicates a predicate on the node that is being evaluated. If we would have written `Entity/Property/Type`, we would have got all types, of all properties, of all entities. The predicates restrict this by only allowing those with the right id's.

We define a transformation to be a combination of an APath and a local transformation. The two are separated by a ::-sign. As examples, we specify the removal of the 'description' property and the substitution of the type in a set:

```
Entity[Id="Group"] / Property[Id="description"]
   :: remove
Entity[Id="Web"] / Property[Id="admin"] / Type / Set / Type
   :: substitute with Group
```

We also need more complex transformations, such as copying properties over an association, or merging entities. Although these can be modeled as separate transformations or transformation patterns [Hößler et al., 2005], we recognize them to be similar to the already defined transformations, with the addition of being able to use other data model elements as input. So copying the topic from a Web to all of its pages is similar to adding a topic property to every page, with its web topic as a source:

```
Entity[Id="Page"] / Property
   :: add Web/Property[Id="topic"]
```

Elementary transformations are combined by sequential composition indicated by a semi-colon. Figure 6.1 (right) shows the result of applying the above transformations to the Wiki data model in Figure 6.1 (left).

6.3.2 *Deriving Data Migrations*

We have defined the data models and the data model transformation. The last step is therefore to specify a 'data model transformation' to 'data transformation' mapping, as indicated by the dashed arrow in Figure 6.2 (right). The implementation of the mapping depends on numerous factors, such as how the data is stored, what platform is available to execute the data transformation on and the quantity of the data. The implementation is therefore driven by the context. We have implemented the mapping using Stratego/XT [Visser, 2004] and a data model to Java classes mapping from the WebDSL project [Visser, 2008a]. It maps the data model transformations as shown above to a data migration program in Java. The migration program loads objects from a database, transforms them to conform to the new data model and stores the new objects. It follows a so-called Extract-Transform-Load (ETL) process.

The migration mainly uses two libraries, namely an object to relational mapping and an object transformation library. The first library provides functionality for loading and storing Java Objects in a relational database. The migration program is based upon the Java Persistence API (JPA) [Biswas and Ort, 2006], which provides an interface to accessing these types of libraries. Consequently, any JPA compliant library is suitable. An example of such a library is Hibernate [Hibernate, 2008]. The combination of JPA and for example Hibernate supports a large number of database systems. The second library provides functionality for transforming Java objects and managing these transformations. It supports transformations such as adding attributes, changing attribute types and changing attribute names, but due to Java restrictions, the set of transformations does not directly cover the elementary transformation set we have seen above. We have written the transformation library specifically for the mapping, but it could also be used in different settings.

In the remainder of this section, we introduce the mapping using the examples we have presented above. Although the mapping directly refers to the transformation library, we use a domain specific language (DSL) for the library to abstract away from the underlying Java and JPA details. Nevertheless, the DSL can directly be mapped onto executable Java code.

Basic concepts. The group addition, introduces most of the basic concepts. It is mapped to the following:

```
transform () to (Group)  {
   EmptyObject();
   AddAttributeByValue("name", "Group Name");
   AddAttributeByValue("description", "...");
   AddAttributeByValue("members", null)
}
```

The `transform` directive defines a transformation as follows:

```
transform INPUT-TYPES to OUTPUT-TYPE {
   TRANSFORMATION-DEFINITION
}
```

In the Group addition, `INPUT-TYPES` is empty and `OUTPUT-TYPE` is a Group. The transformation itself starts with an empty object (an object with no attributes) and subsequently adds the various attributes of Group. The `AddAttributeByValue` directive has the new attribute's name as first parameter and the attribute's value as second.

Similarly, the description removal is mapped to:

```
transform (Group) to (Group)  {
   DropAttribute("description")
}
```

ANNOTATIONS. In the data transformations above, we carelessly introduced values for each of the attributes (`"Group Name"`, `"..."` and `null`). These are required in a data transformation, but unknown in the data model or data model transformation. Such information can be considered to be a separate input of the mapping, yet at the same time, storing it separately from the data model transformation would be impractical.

As a solution, we allow data model transformations to be annotated. When transforming data models, these annotations can be ignored, but when looking at data transformations, annotations provide the information we were missing. Instead of writing:

```
Entity[Id="User"]
   :: add age :: int
```

to add an attribute `age`, we therefore write:

```
Entity[Id="User"]
   :: add age :: int   defaultValue(25)
```

Using a similar approach we specify the value of a newly added group.

DATA-LEVEL COMPUTATIONS. Copying the topic property from a web to a page is done by an attribute addition. The attribute addition by constant value we have seen above is not sufficient. We need an attribute addition by

computed value here. The computation itself is a parameter to the attribute addition:

```
transform (Page) to (Page)  {
   AddAttribute("topic", getWeb().getTopic() )
}
```

In Java, the computation is represented by an anonymous class.

The substitution from the previous section indicates a type substitution. At the data level this is reflected by a conversion to a value of the new (substituted) type. There are various type substitutions that have a standard value conversion. Examples are int to string, string to int, but also set of int to set of string. We have explicitly included the conversions for these substitutions in our mapping.

The substitution from the previous section is a set of User to a set of Group substitution. Such a substitution does not have a standard value conversion. To still be able to execute the data transformation, the user is required to explicitly specify the desired conversion by means of an annotation[1]. An example conversion would be to convert our set of user to a set of singleton groups in which each group holds exactly one user. This is mapped to:

```
transform (Web) to (Web)  {
   AttributeSetConversion(
      "admin",
      new Group( getUser().getName(), { getUser() } )
   )
}
```

Note that the name `AttributeSetConversion` indicates that the conversion (the second parameter) is applied to each of the elements in the admin set, not to the set as a whole, which would be the functionality of the `AttributeConversion` transformation.

TYPES. Each of the above transformations refers to types. They have a set of source types and a target type. Since the code above is directly mapped to Java, these should represent actual Java types. The final step of the mapping is therefore to construct a Java type base to support the transformations. To establish the type base, we use a Java type generator from the WebDSL project, that takes a data model as input and produces the corresponding Java classes as output. Without investigating the transformations, we generate all types for the data model before transformation as well as all types in the data model after transformation. The resulting types are stored in different Java packages to prevent name clashes.

[1]Not by means of a parameter, as this computation only influences the data, not the data model.

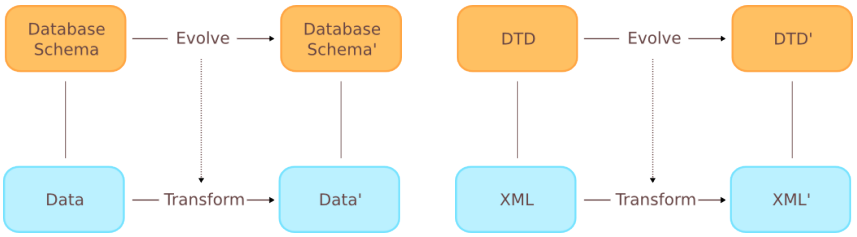


Figure 6.3 Schema evolution (left) & DTD evolution (right)

In addition to the source and target types, we also sometimes need types half-way through the transformation (e.g. when using the attribute addition by computation). These are generated when performing the mapping.

6.4 HETEROGENEOUS COUPLED TRANSFORMATION

When data models evolve, conforming data needs to be transformed to reflect these changes. Although a frequently reoccurring approach is to define data transformations manually, it requires a significant effort and can hold back a development or maintenance process. We have shown that it can be performed automatically. In this section we step away from the detailed look on data model transformations and take a broader look at the problem from a higher level of abstraction.

6.4.1 *Horizontal Generalization*

Recall Figure 6.2. It shows the outline of the coupled data evolution problem. We have looked at Object Oriented data models describing data in a data base. If we would have described our data by means of a database schema (e.g. SQL schema), we would have an evolving database schema and data that has to be transformed to reflect these changes [Cunha et al., 2006, Berdaguer et al., 2007, Gupta et al., 1993], as shown in Figure 6.3 (left). The problem of coupled data evolution therefore reoccurs when using a different formalism for describing our data.

Similarly, we could use XML to store our data and DTD's to describe it. Again our DTDs evolve to satisfy changing requirements and our XML data needs to be transformed to reflect these changes [Lämmel and Lohmann, 2001], as shown in Figure 6.3 (right).

We also see the same problem reoccurring in different domains. When programming languages evolve, the programs written in it have to be migrated to the new version of the languages. The programs have to conform to

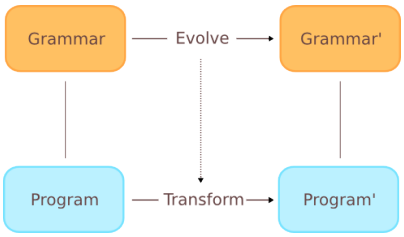


Figure 6.4 Grammar evolution

```
Entity*              → DataM    { cons("Model")   }
Id "{" Prop* "}"     → Entity   { cons("Entity")  }
Id "::" Type         → Prop     { cons("Prop")    }
"int"                → Type     { cons("Int")     }
"bool"               → Type     { cons("Bool")    }
Id                   → Type     {                 }
"set of" Type        → Type     { cons("Set")     }
Name                 → Id       { cons("Id")      }
```

Figure 6.5 Data model grammar

the grammar of the programming languages [Pizka and Jürgens, 2007b] (Figure 6.4). Similarly, when meta-models evolve, conforming models need to be transformed to reflect the evolution [Gruschko et al., 2007, Xiong et al., 2007, Favre, 2003, Wachsmuth, 2007b, Hößler et al., 2005, Hearnden et al., 2006].

Coupled evolution is a reoccurring phenomenon. Naming conventions for the coupled evolution problem vary in the different areas between co-evolution, two-level data transformations, coupled transformation and simply synchronization or adaptation. But they effectively address the same problem of coupled evolution. Lämmel discusses this for a subset of the above in [Lämmel, 2004], naming it the ubiquity of coupled transformation problems. Identifying the coupled evolution problem in different domains is a form of *horizontal* generalization.

6.4.2 *Vertical Generalization*

We introduced the data model language on the fly in the previous sections. We thereby implicitly defined its syntax. The syntax is formalized by the grammar found in Figure 6.5. It is written in SDF [Visser, 1997] format. For simplicity, it only shows the context-free production rules. The lexical syntax definitions and the start symbol (`DataM`) definition have been left out. On the left-hand-side of each production rule the construction of the specific sort is defined, on the right-hand-side the produced sort. Each of the rules are annotated, which is indicated by the `{...}` text at the end of each rule. We ignore these annotations for now.

As data conforms to a data model, data models conform to the data model

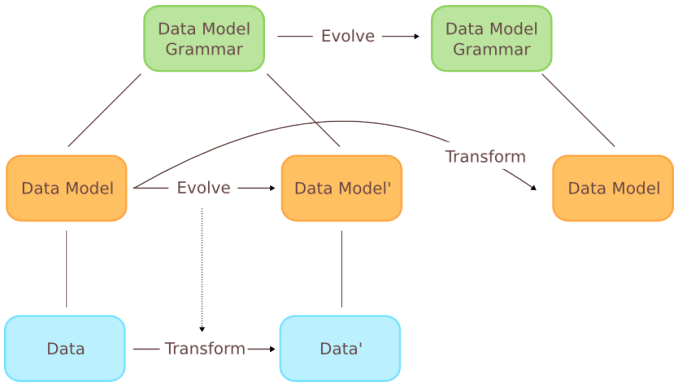


Figure 6.6 Vertical generalization

grammar. The grammar describes the structure of the data model and the data model describes the structure of the data. The data model grammar itself is rather limited. In future, it may for example be useful to add support for more attribute types, inheritance, or support for uniqueness of property values. So, in practice, the grammar is far from fixed and is itself subject to expansion and modification. No different from the data model scenario, if the grammar changes, data models that originally conformed to it are invalidated and need to be migrated along with the grammar. In other words, we have a second scenario of coupled evolution in the single context of data models. Figure 6.6 shows the extension with the additional evolution scenario.

We cover another conformance level by saying that also the SDF syntax may be subject to change, at which point, the grammar defined above has to be migrated along with the changing SDF definition. From which we see that the same coupled software transformation problem reoccurs over different conformance levels, which is a vertical generalization of the problem. In terms of model-driven architectures [Soley et al., 2000] the vertical generalization can be phrased as coupled evolution on the levels M_1-M_0 (data model - data), M_2-M_1 (data model grammar - data model), M_3-M_2 (SDF - data model grammar), or even higher if M_3 is not defined in itself.

To abstract away from a specific conformance level and from specific areas of application, we will from now on use a generalized representation of the problem as shown in Figure 6.7 (left). In this generalized view, we see the common aspects of coupled software language evolution:

- An evolving software language (M i)
- Software that is subject to transformation to reflect the evolving language (M i-1)
- A means to define software languages (M i+1)

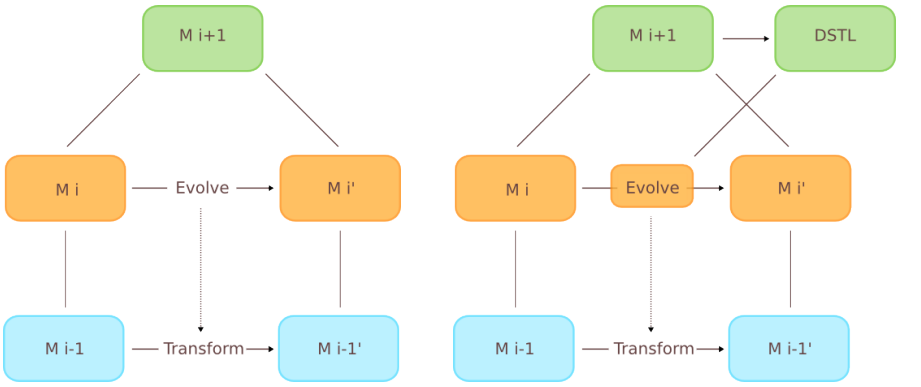


Figure 6.7 Generic representation of software language evolution (left) & generic architecture for supporting software language evolution (right)

For the case of data model evolution, we have automated the transformation process. To do the same in the generic case, we need a way to formalize the evolution for an arbitrary software language. Furthermore, we need a mapping from the language evolution to a concrete transformation.

6.5 GENERIC ARCHITECTURE

In this section we propose and outline a generic architecture for coupled software evolution. Its goal is to reduce the manual effort involved in traditional coupled evolution. Furthermore, it structures the evolution process, increases the transformation abstraction levels and allows for common problems to be solved once instead of repeatedly.

Traditional approaches to coupled evolution are usually based on architectures similar to the one in Figure 6.2. The generic solution is based on the generalized and extended architecture displayed in Figure 6.7 (right). The main component in the architecture is the definition of the transformation language used to formalize the evolution (named DSTL). In earlier work, the transformation language is usually fixed and considered to be an assumption of the approach. We assume it to be variable and consider it an artifact in coupled evolution.

Although the transformation language stands out most in the figure, the key concept of the architecture lies in the added arrows. The dashed arrow denotes a transformation defined by the user. The solid arrows denote automatic transformation, these do not require human interaction.

Input to the architecture is a coupled evolution scenario as explained in the previous section and a mapping from a top-level transformation to a bottom-level transformation. The first should come for free (either implicit or explicit),

since it is merely defining what is to be evolved. Without it, the coupled evolution problem does not exist. The second is also part of most domain-specific approaches and may to that respect be reused in these specific domains.

The mapping, or dashed line in the figure, is defining a semantic link between the two levels, which is by definition sufficient to allow for coupled evolution. Since it is indirectly based upon M i+1, it is generic over any software (or model) being evolved within the same domain. We therefore have fixed mappings for the domain of data model evolution, or the domain of SDF evolution. In practice, the evolution scenario is therefore the main input, varying most frequently.

Based upon the two inputs, the architecture provides a structured approach to software language evolution, consisting of:

- Automatic derivation of a transformation language for each domain
- Automatic derivation of an interpreter for transformations in the transformation language
- Automatic software migration along a specified transformation

In practice, the transformation language is needed to define the mapping. Yet, since the transformation language derivation is automatic, the ordering will not be a problem in practice. The following subsections focus on the aspects of the architecture individually.

6.5.1 *Deriving Domain Specific Transformation Languages*

The first and most central component of the architecture is a transformation language specific to the M i+1 definition. We will refer to it as the Domain Specific Transformation Language (DSTL). The transformation language cannot be generic, as we cannot construct a complete mapping from a generic language. Generic languages contain by definition concepts that are not part of the domain[2].

Many of the traditional transformation languages for coupled evolution define a large set of elementary transformations and an extensive mapping. In contrast to this, we focus on a transformation language that is as small as possible, but still covers all transformations. This makes defining the mapping as easy as possible. Usability of the language is achieved through abstractions. We have implemented the DSTL derivation in Stratego/XT and assume M i+1 to have been defined in SDF. The DSTL syntax is again defined in SDF.

Input to the derivation is M i+1, the software language grammar (in SDF). The derivation produces elementary transformations from the production

[2]Having a partial mapping is similar to using an implicit domain specific language. The language is defined by the domain of the mapping

rules in the grammar. It starts at the productions of the start symbol and traverses the grammar recursively. We distinguish different types of production rules, for which different types of elementary transformations are generated.

LISTS. The top-most production rule in the data model grammar (Figure 6.5) defines a data model to be a list of entities:

```
Entity*                    → DataM
```

In the transformation language, the list is reflected by two list operations, namely addition and removal of entities. The syntax for these transformations is defined by the context-free productions:

```
"add" Entity               → Transformation
"remove"                   → Transformation
```

In the same way, the addition and removal of properties are generated when considering the Entity production recursively. Furthermore, we generate transformations for optional symbols in a similar way (these are set and unset transformations).

LEXICAL SYNTAX. When a symbol is defined to be lexical, it has no more productions and can thus not be decomposed further. The recursion therefore stops and a transformation is generated to substitute its value. An example of a lexical symbol is `Name`, for which the following transformation is generated:

```
"substitute with" Name     → Transformation
```

MULTIPLE PRODUCTIONS. The symbols that are considered above are either lexical, or produced by a single production. The `Type` symbol inside a property can be produced in multiple ways (namely, `"int"`, `"bool"`, `"set of Type"`, or `Id`). Consequently, we must allow it to be substituted by one of these:

```
"substitute with" Type     → Transformation
```

We import the original data model grammar into the DSTL definition to reuse the `Type` symbol that was defined in the original grammar.

TYPE CHECKING. A software language defines groups of software elements. For data models we have entities, properties, ids, but also ids inside sets, or ids inside a property. The local transformations defined above are only applicable to some of the element groups, which make up the domain of a local transformation. For example, the addition of entities can only be applied to data models and the substitution of types only to properties or sets.

We use APath expressions to indicate where a local transformation is to be applied. Each APath expression results in certain groups of software elements. To make sure a local transformation is applied within its domain, we need to verify that the APath expression to which it is connected can only result in elements that are in the domain of the local transformation, which is a form of type checking.

We have implemented type checking for any DSTL. It primarily consists of three components: A generation of domains for each of the local transformations. A (generic) type derivation for APath expressions and (generic) functionality that checks whether the result of the APath will indeed fall inside the domain of the local transformation. The type checking is complicated by the use of recursive productions: the `set of int` should fall in the same group as the set inside `set of set of int`.

LARGER GRAMMARS. The presented data model grammar is small. We have used a much larger data model grammar, which was developed as part of the WebDSL project. Although the principles above can be applied to all the rules in a larger grammar, in practice, one does not want to be able to transform every group of software elements. In the small grammar, we could for example leave out the type substitution within sets if we would not be able to map it to a data transformation.

By means of annotations on the production rules of a grammar, the user can indicate which rules (and thereby what symbols) should be transformed and which should not be transformed. There are two possible annotations: A 'transform' annotation, which tells the tool to generate transformations for a production rule and a 'constant' annotation, which tells the tool to take the production rule into account by recursively generating transformations for each of the symbols on the left-hand-side (in SDF), but not generating transformations for the production rule itself. No annotation on a rule means that it is ignored during DSTL derivation.

6.5.2 *Automated Transformation*

The DSTL syntax we have defined allows us to write transformations. The next step is to execute these transformations. For this purpose, we have defined an interpreter generator. Similar to the syntax generator, it takes the software language definition as input, but instead produces an interpreter for the associated DSTL as output. The interpreter mainly consists of:

- A mapping of the elementary DSTL transformations onto generic transformations
- A generic transformations library (build on top of Stratego/XT)
- Implementation of generic DSTL constructs such as composition and abstraction

- An APath evaluation library

The first is specific for the DSTL and therefore generated. It mainly consists of production rules that denote the specific to generic mapping. These look like:

```
transform(|mmodel, path):
   AddProperty(newValue) → <addAtLocator(|path, newValue)> mmodel
```

The last three items are generic over all DSTLs, so defined once. Their definition is in most ways straightforward and is therefore not discussed here.

6.6 RELATED WORK

Coupled evolution plays a significant role in computer science and has been treated in various areas. Earlier research has primarily focused on constructing coupled evolution support for specific domains. We discuss related work in the most important domains of coupled evolution: model evolution, domain specific language evolution and schema evolution.

Coupled evolution for the meta modeling domain is introduced by Gruschko et al. [2007]. As is the case for most publications on coupled evolution for models, Gruschko models evolution using small elementary transformation steps. A classification of these steps is proposed: non-breaking changes, breaking and resolvable changes and breaking and unresolvable changes. A classification, which is frequently reused in later work and is also applicable to our work, yet not directly relevant to the proposed architecture. In his paper, Gruschko also identifies different steps in coupled evolution, although these steps are generic, they mainly consider what we have called 'the mapping' and are in that sense only applicable to a subset of what has been discussed here. The only step, which does not fall inside the scope of this mapping is a change detection, to determine the evolution steps that have occurred between two given models. Chapter 5 discusses evolution detection in detail.

Wachsmuth [2007b] introduces a set of transformations specific to MOF [Object Management Group, 2006] compliant meta models. The set is very similar to the set of elementary transformations for data models as we have introduced in Section 6.3.1 and which is derived automatically in our approach. Different to the transformations we have derived is their distinction between two type changes, namely generalization and restriction, yet they do not provide a specification on how these should be mapped to concrete types. Furthermore, they have transformations to support changes to inheritance and inlining of classes. Both concepts were not included in our input data model grammar and are therefore not reflected in the output. Wachsmuth proposes a mapping to model migrations implemented in QVT [Object Management Group, 2007].

Similar to Wachsmuth, Herrmannsdoerfer considers coupled evolution on metamodels based on small evolution steps [Herrmannsdörfer, 2007]. He focuses on the Eclipse Modeling Framework (EMF) [Eclipse Foundation, 2008], in which ECore, the meta-meta model implements a subset of MOF. In his approach, named COPE, Herrmannsdoerfer distinguishes two types of evolution steps: open and closed coupled evolution. The first is what we have named the elementary transformations and the second are transformations based upon these. In contrast to other works, this view does provide a way of abstracting from meta-model specific transformations. However, the derivation of the elementary transformations as well as the definition of these transformations are left to the user. This does not only require additional effort, it also prevents structured abstractions as are possible in our approach. Herrmannsdoerfer provides a prototypical editor based on Eclipse.

In the area of domain specific languages, Pizka et al. discuss the evolution of DSLs. They claim three obstacles in DSL development: (1) Stepwise bottom-up generalization is required, which is a special case of the evolution we have been looking at. (2) DSLs should be layered, which is specific to DSLs and not directly related to coupled evolution. (3) Automated co-evolution is required for DSLs, which is what we generically solve in our work. As we have seen in [Wachsmuth, 2007b], Pizka's work is focused on a single domain, namely DSLs, it is limited to the discussion of a transformation definition and mapping specific to this domain.

With respect to the data variants of coupled evolution (schema evolution) and the related two-level data transformations, numerous approaches have been found to solve these problems [Lämmel and Lohmann, 2001, Cunha et al., 2006, Berdaguer et al., 2007, Gupta et al., 1993, Alves et al., 2008b]. These mainly focus on the schema to data mapping, frequently taking different types of complicating concepts into account, such as data restrictions and performance optimization. These are typically aspects that may also be solved generically, such that they can be used in any domain. In current work, we have not focused on this, but it may be interesting as future work.

6.7 CONCLUSION

In this chapter, we presented two directions of generalizing coupled software language evolution scenarios and introduced the concept of heterogeneous coupled evolution. We presented an architecture to automate coupled evolution on an arbitrary software domain (e.g. programming languages, modeling or data modeling). The architecture requires as input: a coupled software evolution scenario and a mapping from software language transformations to software transformations. The outputs are: Automatic derivation of a domain specific transformation language (DSTL) to formalize the software language evolution; automatic derivation of an interpreter for transformations conform-

ing to the DSTL; and automatic software migration along the evolving software language.

Using Stratego/XT, we have implemented a coupled evolution tool to support the architecture. It is based on software languages defined in SDF. We have successfully applied the tool to the domain of data modeling in the web modeling language WebDSL [Visser, 2008a], where we have used it to create a tool for automatic database migration along an evolving data model, which targets a broad set of databases.

ACKNOWLEDGMENTS

This research was supported by NWO/JACQUARD project 638.001.610, *MoDSE: Model-Driven Software Evolution*.

Conclusion

7

In this dissertation we studied concepts, techniques and tools to support coupled evolution in the context of a conformance relation. We aimed to ease the coupled evolution process, while reducing its impact on development and broadening its view across the traditional boundaries of coupled evolution domains such as dataware, modelware and grammarware.

We studied existing literature to find directions for further research. We analyzed a set of evolution case studies, in order to identify common evolution patterns, to assess their automation potential and to analyze their characteristics. We implemented automatic coupled evolution in the context of a double conformance mapping, aiming to simplify the coupled evolution process for developers. We formalized coupled metamodel evolution, and derived a technique for automatically detecting evolution patterns from a sequence of evolved metamodel versions. Finally, we generalized the coupled evolution solution across domains, by automating the common efforts needed to establish a new coupled evolution implementation.

The remainder of this chapter enumerates the core contributions of this dissertation, it answers the research questions posed in the introduction and it provides recommendations for further research in the coupled evolution domain.

7.1 SUMMARY OF CONTRIBUTIONS

- A systematic literature survey [Kitchenham and Charters, 2007] on coupled evolution approaches in the context of conformance, across different technological spaces (Chapter 2).

- A feature model for classifying coupled evolution approaches independent of technological spaces. Its application to existing approaches and the interpretation of the results (Chapter 2).

- An extensive catalog of coupled operators, which are either motivated from the literature or from case studies that we performed. An organization of this catalog to ease selection of the right coupled operator and to assess the impact of the coupled operator on the modeling language and its models (Chapter 3).

- A domain-specific language for specifying data model evolution in the context of WebDSL, including checks for evolution validity and correctness (Chapter 4).
- Implementation of coupled operators for the evolution of WebDSL data models and migration of WebDSL databases through SQL migration scripts (Chapter 4).
- A formalization of the core concepts involved in coupled evolution of metamodels and models, namely metamodels, difference models, and evolution traces (Chapter 5).
- Automatic reconstruction of complex metamodel evolution traces from difference models, dealing with operator dependencies and interference (Chapter 5).
- Two generalizations over coupled software language evolution scenarios, introducing the concept of heterogeneous coupled evolution (Chapter 6).
- An architecture to support heterogeneous coupled evolution of software languages, offering an automatic domain specific transformation language generation (DSTL) and an automatic DSTL interpreter generation (Chapter 6).

7.2 RESEARCH QUESTIONS REVISITED

Research Question 1

> *How do we characterize and compare coupled evolution approaches across technological spaces?*

Different approaches to coupled evolution, show different characteristics, yet many of these are comparable. Characteristics can be formalized into features, offering a discrete distinction and a basis for comparison. In Chapter 2, we have derived a set of features to characterize coupled evolution approaches and organized the set in a feature model. Subsequently, we have applied the feature model to existing coupled evolution approaches.

By interpreting the results of applying the feature model, we derived reusable solutions and directions for further research. Amongst others we concluded: Views are a key concept in the domain of dataware and commonly used to unobtrusively support coupled evolution. Other domains make little use of views. Porting views to other domains could offer novel approaches; Migration is a transformation, yet only few approaches use a transformation

language for specifying migrations. Existence of suitable transformation languages would ease the implementation of coupled evolution; Versioning of schemas, classes, or objects has common use in dataware, received some, but little research in modelware, yet proves beneficial in both domains. A generalization or reimplementation of versioning in other domains, would support and simplify coupled evolution; Finally, in-place transformations may well be used in modelware or grammarware, to support larger artefact size, or artefacts that require continuous availability.

Research Question 2

How can coupled evolution concepts and solutions be generalized across technological spaces?

Coupled evolution is a reoccurring phenomenon in various domains. Nevertheless, its underlying concepts remain similar. In Chapter 6 we showed that coupled evolution can be generalized horizontally – across domains – and vertically – across meta-levels. Furthermore, in Chapter 2, we showed through the feature model, that characteristics are comparable along the horizontal generalization.

By exploiting the horizontal and vertical generalizations, we derived a framework that emphasizes the commonalities between domains and extends them to automate the process of deriving a new coupled evolution approach. It offers an automatic domain specific transformation language (DSTL) generation and an automatic DSTL interpreter generation. The generation results partially implement arbitrary coupled evolution approaches. However, the generalization is not complete: The mapping from evolution to migration is domain-specific and hard to generalize. Chapter 4 highlights several of the domain-specific problems faced in the object-oriented dataware domain.

Research Question 3

What metamodel evolution patterns can be distinguished, which allow automation in the context of migration?

Determining common evolution in each of the domains of coupled evolution in the context of conformance would require extensive investigation of each of the domains. As our implementations primarily targets the domains of dataware and modelware, we performed an analysis of evolution in these two domains. We looked at evolution in various case studies, several of which are industrial. Additionally, we analyzed which evolution is supported in the existing evolution tools and publications.

Using the analysis results, we have derived a catalog of 61 operators, that can accurately describe most of the evolution of the inspected cases. Evolution which is not supported by the catalog but which did occur in the case studies, only occurred in a single case study and therefore offers little ground for

automation. We organized the catalog along the effect on the modeling language or the migrated models, to ease evolution automation and to ease its usage by developers. The catalog, its coupled operators and their properties are discussed in Chapter 3.

Research Question 4

> *How can software language evolution be formalized, such that it both functionally and understandably represents the developer's evolution intent?*

For understandability, an evolution specification of a domain-specific model should follow the domain-specific modeling concepts. This reduces the gap between model and evolution specification. Formalizations come in different forms for different types of models. We offer a formalization for evolution of the object-oriented data models found in WebDSL models. Nevertheless, the concepts of WebDSL data models are similar to object-oriented data models and metamodels, hence the formalization can be generalized to the domains of modelware and dataware.

Chapter 4 discusses the formalization of evolution of WebDSL data models. It follows the catalog of common evolution operations outlined in Chapter 3, to ensure a broad coverage. The formalization targets the same abstraction level as WebDSL, by reusing the domain-specific concepts found in WebDSL. Thus reducing the gap between the evolution formalization and the original model as to increase the understandability of the evolution. At the same time, the formalization abstracts away from the lower-level details of the underlying implementation frameworks (Hibernate and Relational databases), to ensure a focus on the evolution, rather than the implementation.

As the formalization follows the catalog of operators, from Chapter 3, we know that it is not sufficiently complete to cover arbitrary evolution. A means to deviate from the operators in the catalog is needed, either by customization or by addition. Yet this remains a topic for further research. As for the Acoda tooling, implementing unsupported migrations manually in SQL is the recommended, yet insufficient solution.

Research Question 5

> *How do we support coupled evolution unobtrusively and prevent the undesired loss of information during migration?*

Coupled evolution has an impact on software development. On the one hand, coupled evolution requires manual effort, as the developer's evolution intent cannot be completely obtained automatically. On the other hand, artefact migration implies a risk of information loss. We aim to reduce the impact of coupled evolution as much as possible, to streamline the evolution process.

We discussed two directions to reduce coupled evolution impact. In Chapter 5, we discussed how to automatically reconstruct evolution from two ver-

sions of a metamodel (or data model). The automated reconstruction supports complex evolution operators, operator dependencies as well as operator interference – where the effect of one operator is partially or completely hidden from the final result. Reconstruction still requires developer feedback to ensure the correct evolution has been detected, but reduces evolution effort drastically compared to defining evolution manually. In Chapters 3 and 4, we discussed how to reduce the risk of information loss in coupled evolution. On the one hand, evolution operator characteristics assess their impact on the modeling language and the set of models. Combination of such properties can predict information loss before applying migration. On the other hand, evolution validity is automatically validated, to ensure a functionally correct evolution after it has been edited or completely specified by a developer.

7.3 EVALUATION

We used two types of research throughout the dissertation. We used analytical research to increase our understanding of the status quo (Chapters 2 and 3). We used research of a constructive nature, to improve the status quo (Chapters 4, 5 and 6). Analytical research requires case studies that are a representation of the status quo. Constructive research, requires case studies to evaluate that a change indeed improves the status quo.

The case studies used for analytical research are summarized in Figure 3.2. Their histories were analyzed by mining their repositories and examining their commit logs. They primarily provided input to the set of evolution patterns derived in Chapter 3. Both industrial cases as well as open source cases were used.

The constructive research we discussed throughout the dissertation primarily targets to improve the process of coupled evolution. Evaluation of such constructive research is complicated as the change does not merely affect one measurable outcome, does not merely affect a whole set of outcomes, but completely changes the case itself. The presence of support for evolution does not just shorten the evolution, or shorten development time by automating migration, it completely changes the evolution that would have taken place.

To still evaluate the constructive research, we used two types of case studies. Firstly, we used case studies, which we were (partially) executed without support of new evolution tooling. These case studies are Bugzilla and Researchr. Secondly, we used case studies, which (partially) benefit from the newly developed technology. These case studies are Researchr and YellowGrass. The latter category had to be developed from scratch, these cases are therefore typically smaller, but examined in much greater detail. Nevertheless, both Researchr and YellowGrass are much-used systems, making them

realistic cases. The three main cases YellowGrass, Researchr and Bugzilla are discussed in Appendices A, B and C respectively.

In Chapter 3, case studies were used for quantitative analysis and collection of evolution patterns. In Chapter 4, YellowGrass was used to qualitatively validate the functioning of the migration component of Acoda. We verified that on YellowGrass, Acoda migrations yield the correct results and do not cause undesired loss of data. Although we optimize the generated migrations for performance (see Section 4.8), we did not evaluate these improvements, as an evaluation would rather evaluate the performance of the database management system than the specified migration itself. The time it takes to map evolution onto migration is negligible. Furthermore, migration of very large databases, or distributed migration remains a topic of further research. In Chapter 5, we used the Researchr history to do a preliminary evaluation of performance. In particular, we were interested in how big evolution traces typically get, as they determine the reconstruction time. Further evaluation on different case studies could provide input to further improvements of the algorithm. The reconstruction algorithm can principally reconstruct any evolution trace. A (heuristics-based) selection of the sought-after evolution trace falls outside the scope of the work (although the current heuristic on trace length seems to work exceptionally well). An evaluation of such selection therefore also falls outside the scope of this work.

7.4 FUTURE RESEARCH RECOMMENDATIONS

This dissertation addresses different concepts, techniques and implementations on coupled evolution. As shown in Chapter 2, coupled evolution is a domain of ongoing research. New techniques are developed and new domains are explored. The evolution problem is common, yet hardly supported in practice. Much additional research is needed before coupled evolution can commonly find its way into regular software development. This section addresses some directions of research that follow from the earlier chapters.

7.4.1 *Metamodeling Formalism*

A coupled evolution approach is bound by the metamodeling formalism it supports. The discussed concepts and implementations primarily focus on the core modeling concepts found in most metamodeling languages. We considered classes, inheritance, cardinalities and inverses. However, concrete metamodeling formalisms offer a broader range of constructs, requiring the need to extend the presented ideas and implementations.

Most concrete metamodeling formalisms, such as Ecore [Steinberg et al., 2009] or MOF [Object Management Group, 2006], offer additional constructs,

such as interfaces, packages and annotations. Although support for these requires additional implementation, they have limited impact on persisted artefacts and thereby limited impact on migration or coupled evolution. They do not affect the presented concepts for coupled evolution.

More influential additions to the metamodeling formalism, such as invariants, have more significant impact on coupled evolution. All metamodel definitions offer restrictions on the data set. More definitions implies a more restrictive set. Each of the restrictions have in common that they have a fairly predictable impact on the extension. For example, cardinalities restrict the data by quantity. In migration, we consider the full impact of these quantity restrictions and adapt the data migration accordingly. Due to their wide scope, the impact of invariants on a metamodel's extension is harder to predict. Therefore, it is harder to adjust migration to meet invariants. Additional research is needed to support invariants in coupled evolution.

Previous chapters mostly discussed metamodeling in the dataware and modelware domains. Although these domains are different, their metamodeling formalisms and conformance relation are similar. Also the domain of XMLware shows a similar metamodeling formalism and conformance relation. There are however clear differences with the domain of grammarware. This, and other differing domains may need a different type of detection, a different type of migration and a different catalog of coupled operators. The presented work may not be directly applicable. As presented in Chapter 2, some research into coupled evolution of grammars and programs exists [Staudt et al., 1987, Garlan et al., 1994, Jürgens and Pizka, 2006, Pizka and Jürgens, 2007b,a, Lämmel, 2001, Lämmel and Zaytsev, 2009a,b]. Concepts may be similar, but additional research is needed to see to what extent existing technology can be applied to other less related modeling domains.

7.4.2 *Coupling Customization*

We support coupled evolution through a set of coupled operators. By a series of case studies, we showed that the set is near complete in practice. Yet at the same time, we also showed that there are evolution steps in real-life development, in which the set does not suffice. Some evolution is not supported.

To enable unsupported evolution, developers can implement a suitable migration manually. However, similar evolution may occur repeatedly, evolution may be similar to an existing coupled operator or evolution may comprise a complete or partial combination of existing coupled operators. In each of these cases, automation through coupled evolution is preferable, when new coupled operators can be added, existing coupled operators can be adjusted, or existing coupled operators can be combined. To fully support an evolution process, coupled operators need to support customization.

Some approaches exist, which allow developers to define custom operators and reuse them during evolution (e.g. [Herrmannsdoerfer et al., 2009]). Yet

these neither support reusing existing coupled operators in operator definitions, nor do they support detection of custom coupled operators. Support for coupled operator customization throughout the entire coupled evolution workflow would allow automation of the complete evolution process.

7.4.3 *Implementing Migrations*

A large amount of the time spent on implementing a new coupled evolution approach involves implementing the set of coupled operators. Coupled operator implementation mostly involves implementing migrations or migration patterns. The choice of migration platform, or migration language, greatly influences the ease of implementing coupled evolution.

Some domains, offer transformation languages that can directly be used for the purpose of migration and can therefore directly be used to implement the migration component of a coupled operator. For example, in the domain of grammarware, program transformation techniques can be used to implement program migrations. Similarly, in the modelware domain, model transformation techniques can be used to migrate models. However, model transformation techniques are generally not suitable for implementing migrations. They are frequently homogeneous – lacking support for metamodel changes – or may require excessive descriptions, when migration steps are small. In the dataware domain, data transformation techniques are also mostly homogeneous. The heterogeneous approaches, like those offered by database systems, are primitive and lack support for information preservation.

The lack of transformation frameworks suitable for migration hampers the development of coupled evolution approaches. Research into this particular type of transformation is needed to ease migration development. Additionally, alternative approaches to migration may offer new advantages, like seen in the dataware domain. For example, migration by constructing views may offer a coupled evolution solution to running systems.

7.4.4 *Coupled Evolution in the Wild*

Most research on coupled evolution has a strong focus on concepts and prototypical implementations (Chapter 2). At the same time, there appears to be limited use of coupled evolution techniques in practice. Despite the common need for migration upon evolution, manual implementation is generally chosen in favor of automation. Different causes of limited use can be considered. Existing tools may be too prototypical, or there may be a mismatch between what is needed and what is offered through research.

In general, much research ignores practical restrictions, to ease coupled evolution. For example, minimum artefact uptime may be required, thus disabling offline migrations. Artefacts may be physically distributed, requiring the need for distributed migration. Larger artefacts, or larger numbers

of artefacts require better migration performance. But also, organizational restrictions, such as the management of risks can complicate coupled evolution. Additional research needs to reveal the evolution limitations of real-life software development and provide suitable solutions. Additional engineering effort is needed to turn existing approaches into tools suitable for use in industry.

Appendix: Case Study YellowGrass

A

This appendix and the following discuss case studies that provided input to the research underlying this dissertation. This appendix describes YellowGrass, a tag-based and web-based issue tracker. The evolution of YellowGrass was used for statistical analysis in chapter 3 and (slightly condensed) used as running example in chapter 5.

A.1 CONTEXT

Due to the complexity of software, any reasonably-sized piece of software is bound to have bugs. Bugs reduce the quality of software and therefore, most of them need to be addressed during a software's life time. To improve software quality, software projects need to keep record of found bugs, which is known as *bug tracking*. To aid software project management, bug tracking software offers functionality to collect, record, manage and process known bugs.

Found bugs imply the need for software changes. Yet, in software project management, one often needs to keep track of a broader collection of incentives for software change, such as requested new functionality, suggested improvements, or potential scope changes. We refer to these as *software issues*. *Issue tracking software* aids in managing the flow of issues, thus offering a valuable tool to software project management.

A.2 ISSUE TRACKING IN YELLOWGRASS

Various bug trackers and issue trackers exist, such as Mantis[1], Bugzilla[2], or JIRA[3]. However, these are either intended for large projects or for small projects. Yet, typically, software projects grow from a small code base at project start, to a larger code base after some months or years of development. The number of issues and their management tend to complicate over time. YellowGrass supports a growing software development project by making issue management flexible and customizable through the use of tagging.

Tags are simple (unstructured) strings, which are used to label issues. YellowGrass supports any choice and usage of tags. Typically projects use simple

[1] http://www.mantisbt.org
[2] http://www.bugzilla.org
[3] http://www.atlassian.com/software/jira

tags to divide issues into categories, divide issues across software components, or to assign urgency levels to issues. YellowGrass offers generic tag support for any tag, such as filtering issues on tags, applying issue ordering, or enhancing issue search results. More complex tags are structured strings, which receive special functionality support. For example, tagging an issue by `@john` assigns this issue to john and makes it show up on his YellowGrass home page. Tagging an issue by `!john` will make john follow the issue, as a result of which he will be informed of changes to the issue or comments on the issue by other users.

Additionally, tags can be tagged themselves by *meta-tagging*. Meta-tagging assigns special meaning to tags, which allows YellowGrass to reveal extra functionality as a project grows in size. For example, issues can be tagged by a software version number (as tags are unstructured, any versioning scheme is supported). A set of issues with a particular version tag can be combined into a snapshot and a series of issue sets tagged by subsequent versions comprise a software's evolution description. Yet, with merely the generic tag management, these snapshots and evolution descriptions would be cumbersome to maintain. Therefore, YellowGrass allows the users to tag tags as "release". This enables additional functionality, such as a road map, scheduling releases and postponing issues. Meta-tagging offers the possibility to extend the issue tracking functionality as a project grows. Smaller projects are not bothered by functionality they do not use, but as they grow, the functionality can be enabled when needed.

YellowGrass issues are collected in projects, which each has its own tag set. Projects have members, who each have a login and to whom issues can be assigned. YellowGrass is a web application written in WebDSL. Figure A.1 shows a screenshot of the Acoda road map on YellowGrass. YellowGrass is freely available and open source[4].

A.3 YELLOWGRASS.ORG

A public instance of YellowGrass is hosted on http://yellowgrass.org. YellowGrass.org has been live for 2 years. At current date, it supports 40 active projects (some public, some private) recording several thousands of issues. The YellowGrass.org database has been used extensively to test Acoda. The database is not open, due to the sensitive user information and various private projects, yet most information is accessible as public information through the website.

[4]https://svn.strategoxt.org/repos/issolar

Figure A.1 Screenshot of a YellowGrass road map. The left column shows a list of versions, issues for each versions and controls to postpone issues. The right column shows project controls, description and statistics.

A.4 EVOLUTION

Development of YellowGrass started approximately two years ago. It followed an agile development process, offering correctly functioning versions at nearly all stages of development. Its evolution (to date) follows a total of 232 revisions spread across 22 releases[5]. The evolution is recorded in subversion and as the project is open source, the evolution is public.

Evolution of YellowGrass is primarily focused on extension of functionality, refactorings to support the agile development methodology and some design changes to fix poor design decisions (such as the use of global tags versus project-based tags). Due to the use of for example meta-tagging, the data model of YellowGrass is complex with respect to the size of the application. Consequently, the evolution trace follows rather complex steps, which provide excellent input to the previous chapters. A detailed documentation of the changes applied to YellowGrass can be found in the road map on YellowGrass.org and in the subversion logs.

YellowGrass.org has gone live after about one month of development, by then offering a very small subset of the functionality it offers now. Nevertheless, it has been actively used since it went online. As various projects relied on the issue data recorded, the YellowGrass.org database could not loose information. Therefore, Acoda was used extensively since the start of the project. By offering data model evolution support that is in traditional development not available, Acoda has influenced the evolution process, making it more complex, longer and more focused on agile development. There are no steps to be found that prevent or work around evolution, as these were simply not needed.

[5] http://yellowgrass.org/roadmap/YellowGrass

Appendix: Case Study Researchr

B

Researchr[1] is a web-based publication management system, on which one can find, collect, share and review scientific publications and their meta data. The evolution of researchr was used for statistical analysis in Chapter 3 and as feasibility study in Chapter 5. A simplified version of researchr's data model and database is used as running example in Chapter 4.

B.1 CONTEXT

One of the outcomes of research are publications. Over the years, research – in all its forms – has produced large quantities of publications. These large quantities make finding, traversing, or keeping track of publications hard. For the purpose of literature surveys, conference reviewing processes, or simply the process of research, one needs to deal with the quantities, to use existing work and to prevent duplication in future work.

It is not the publications themselves, which are hard to manage in these processes, but their meta data (a publication title, author names, author affiliations, abstract, citations, digital object identifier etc.). Using the publication meta data, the publication itself is generally easily found. Some search engines exist to simplify the process of finding publications. For example, Google Scholar[2] searches publications based on their content. It offers a list of publication meta data as result. Nevertheless, once found, there is little or no support for keeping track of publications, staying up to date with new publications or managing publications, such as reviewing or classifying.

B.2 RESEARCHR.ORG

Researchr[3] is a web application, offering support for the management of publication meta data. It allows users to register new publication meta data and browse a large set of publications already present in the researchr database (imported from DBLP[4]). Users can group publications into bibliographies, review publications and classify them. Users can unite in groups, to share bibliographies and reviews. Publications can be tagged and an extensive meta data

[1] http://researchr.org
[2] http://scholar.google.com
[3] http://researchr.org
[4] http://www.informatik.uni-trier.de/~ley/db

search functionality (not content-based) is available. Additionally, researchr offers a registration of past conferences and a conference calendar marking future events and their deadlines.

Figure B.1 shows a screenshot of the researchr start page.

B.3 EVOLUTION

The evolution of researchr started in March 2008 and is still under active development. Over the past four years, there have been 665 source code revisions and about 60 releases[5]. The website has gone live shortly after the start of the project. Its registered data has been kept since.

Two evolutions are input to the work in the previous chapters. Firstly, the evolution of researchr itself was used in chapters 3 and 5. Secondly, separate from the regular researchr evolution, Acoda was used to evolve the DBLP data model into the current researchr data model and to migrate the DBLP data to researchr data. The migration is later repeated to import new DBLP publications into the researchr database. The example in Chapter 4 is based upon the evolution trace from the original DBLP data model, first to an early researchr data model and next to the current researchr data model. The evolution trace spans a total of ten data model revisions. As it is relatively short in length, it is not considered a significant case study and therefore only served as input to the running example of Chapter 4.

Acoda was not used in the regular evolution of researchr. It may therefore not have had the freedom of evolution available in the evolution of YellowGrass. However, Acoda was used in the evolution from DBLP meta data to researchr meta data, which therefore could evolve more freely and presents more complex evolution steps (hence its suitability as example).

[5] http://yellowgrass.org/roadmap/researchr

Figure B.1 Screenshot of the Researchr start page. The left column shows a set of recently added publications, the right column shows some statistics and shortcuts to active of new items.

Appendix: Case Study Bugzilla

C

This appendix presents Bugzilla[1]. Bugzilla is a widely-used defect (or bug) tracking system. The evolution of Bugzilla was used for statistical analysis in chapter 3. The context of Bugzilla is similar to the context of YellowGrass, which is discussed in Section A.1. This appendix discusses Bugzilla's functionality and its evolution.

C.1 BUG TRACKING IN BUGZILLA

Although Bugzilla and YellowGrass address the same domain, Bugzilla has a larger scope and is feature-wise richer. Bugzilla has little support for tagging, but offers (on top of the functionality offered by YellowGrass): tracking of code changes; handling of patches, such as patch submission and patch review; automated duplicate bug detection; time tracking for larger projects; project status analysis and reporting for project management; and quality assurance management.

At the time of writing, the Bugzilla implementation consist of 128,320 lines of code. 83% of this code is written in Perl. The Bugzilla documentation and page templates are written in XML, they comprise a total of 10% of the code. The remainder of code is mostly stylesheets, written in CSS and client-side code written in JavaScript.

Bugzilla is a web application, supporting different database engines (namely MySQL, Oracle, PostgreSQL and SQLite). It is open source and distributed under the Mozilla Public License[2]. The source code and its revisions are stored using Bazaar (which replaces the original CVS system) and available online[3]. Figure C.1 shows a screenshot of Bugzilla.

C.2 EVOLUTION

Development on (the current variant of) Bugzilla has started in 1998. Bugzilla has been under active development since. There have been approximately 8000 revisions over the past 13 years. The discussion of Bugzilla in this dissertation (Chapter 3) focuses on Bugzilla's data model, of which there have been 280 revisions over the years.

[1] http://bugzilla.org
[2] http://mozilla.org/MPL
[3] http://bzr.mozilla.org

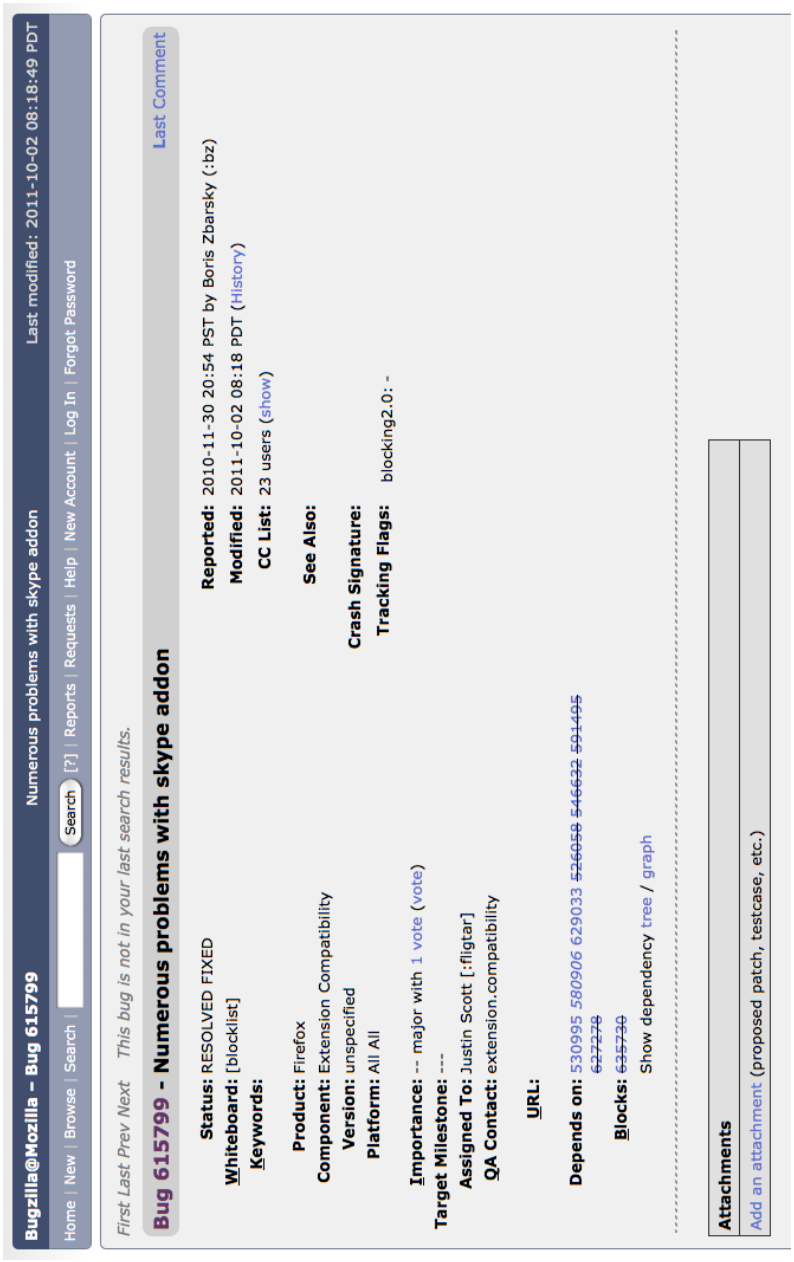


Figure C.1 Screenshot of a Bugzilla bug report. It shows the bug details, its relations to other bugs and attachments. Further down on the page (not visible in the screenshot), comments on this bug are listed.

```
bug_see_also => {
   FIELDS => [
       id     => {TYPE => 'MEDIUMSERIAL', NOTNULL => 1,
                  PRIMARYKEY => 1},
       bug_id => {TYPE => 'INT3', NOTNULL => 1,
                  REFERENCES => {TABLE  => 'bugs',
                                 COLUMN => 'bug_id',
                                 DELETE => 'CASCADE'}},
       value  => {TYPE => 'varchar(255)', NOTNULL => 1},
       class  => {TYPE => 'varchar(255)', NOTNULL => 1,
                  DEFAULT => "''"},
   ],
   INDEXES => [
       bug_see_also_bug_id_idx => {FIELDS => [qw(bug_id value)],
                                   TYPE   => 'UNIQUE'},
   ],
},
```

Figure C.2 Excerpt of Bugzilla's data model.

The Bugzilla data model[4] is encoded in 3000 lines of Perl code. Declarations of tables, columns, foreign keys and indexes are encoded in Perl hashes. Figure C.2 shows an excerpt of the schema. For statistical analyses, these hashes were converted to a more concise format similar to the data models used in WebDSL.

Data model evolution and the development of required associated database migrations received little tool support in Bugzilla's evolution. Upon evolution, Bugzilla's developers are expected to construct a migration script (also in Perl) and add it to a large database modification script[5], which is run during installation or upgrade of a Bugzilla instance. Figure Figure C.3 shows a small excerpt of the migration code.

[4]The data model can be found in the repository at Bugzilla/DB/Schema.pm

[5]The accumulated database migration script can be found in the repository at Bugzilla/Install/DB.pm

```
# 2002-12-20 Bug 180870 - remove manual shadowdb replication code
$dbh->bz_drop_table("shadowlog");
_rename_votes_count_and_force_group_refresh();

# 2004/02/15 - Summaries shouldn't be null - see bug 220232
if (!exists $dbh->bz_column_info('bugs', 'short_desc')->{NOTNULL}) {
   $dbh->bz_alter_column('bugs', 'short_desc',
                         {TYPE => 'MEDIUMTEXT', NOTNULL => 1}, '');
}
$dbh->bz_add_column('products', 'classification_id',
                    {TYPE => 'INT2', NOTNULL => 1, DEFAULT => '1'});
_fix_group_with_empty_name();
$dbh->bz_add_index('bugs_activity', 'bugs_activity_who_idx',
   [qw(who)]);
# Add defaults for some fields that should have them but didn't.
$dbh->bz_alter_column('bugs', 'status_whiteboard',
   {TYPE => 'MEDIUMTEXT', NOTNULL => 1, DEFAULT => "''"});
if ($dbh->bz_column_info('bugs', 'votes')) {
    $dbh->bz_alter_column('bugs', 'votes',
       {TYPE => 'INT3', NOTNULL => 1, DEFAULT => '0'});
}
$dbh->bz_alter_column('bugs', 'lastdiffed', {TYPE => 'DATETIME'});
```

Figure C.3 Excerpt of Bugzilla's database migration script

Bibliography

(2008). *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. W3C. `http://www.w3.org/TR/REC-xml/`. (Cited on page 40.)

Al-Jadir, L. and Léonard, M. (1998). Multiobjects to ease schema evolution in an OODBMS. In *Conceptual Modeling - ER 98, 17th International Conference on Conceptual Modeling*, volume 1507 of *LNCS*, pages 316–333. Springer. (Cited on page 32.)

Altmanninger, K., Seidl, M., and Wimmer, M. (2009). A survey on model versioning approaches. *International Journal of Web Information Systems*, 5(3):271–304. (Cited on page 50.)

Alves, T., Silva, P., and Visser, J. (2008a). Constraint-aware schema transformation. In *Ninth International Workshop on Rule-Based Programming*. (Cited on page 106.)

Alves, T., Silva, P., and Visser, J. (2008b). Constraint-aware schema transformation. In *Ninth International Workshop on Rule-Based Programming (Rule 2008)*. (Cited on page 153.)

Ambler, S. W. and Sadalage, P. J. (2006). *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Professional. (Cited on page 31.)

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. Morgan Kaufmann Publishers Inc. (Cited on pages 35 and 49.)

Balasubramanian, D., Levendovszky, T., Narayanan, A., and Karsai, G. (2009). Continuous migration support for domain-specific languages. In *The 9th OOPSLA Workshop on Domain-Specific Modeling*. (Cited on page 44.)

Banerjee, J., Chou, H.-T., Garza, J. F., Kim, W., Woelk, D., Ballou, N., and Kim, H.-J. (1987a). Data model issues for object-oriented applications. *ACM Trans. Inf. Syst.*, 5(1):3–26. (Cited on page 32.)

Banerjee, J., Kim, W., Kim, H.-J., and Korth, H. F. (1987b). Semantics and implementation of schema evolution in object-oriented databases. *SIGMOD Rec.*, 16(3):311–322. (Cited on pages 32, 61, 65, 67, 69, 71, 73, 75, 78, and 79.)

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 pages 45, 60, 63, 65, 67, 69, 71, 73, 75, and 78.)

Benatallah, B. (1999). A unified framework for supporting dynamic schema evolution in object databases. In *ER 99: 18th International Conference on Conceptual Modeling*, volume 1728 of *LNCS*, pages 16–30. Springer. (Cited on pages 8, 16, 32, 36, and 49.)

Berdaguer, P., Cunha, A., Pacheco, H., and Visser, J. (2007). Coupled schema transformation and data conversion for XML and SQL. In *Practical Aspects of Declarative Languages (PADL 2007)*, volume 4354 of *LNCS*, pages 290–304. Springer. (Cited on pages 105, 145, and 153.)

Biswas, R. and Ort, E. (2006). The java persistence api - a simpler programming model for entity persistence. http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html. (Cited on page 142.)

Bordbar, B., Draheim, D., Horn, M., Schulz, I., and Weber, G. (2005). Integrated model-based software development, data access, and data migration. In *Model Driven Engineering Languages and Systems*, volume 3713 of *LNCS*, pages 382–396. Springer Berlin / Heidelberg. (Cited on page 34.)

Bouneffa, M. and Boudjlida, N. (1995). Managing schema changes in object-relationship databases. In *OOER 95: 14th International Conference onObject-Oriented and Entity-Relationship Modelling*, volume 1021 of *LNCS*, pages 113–122. Springer. (Cited on pages 35 and 49.)

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. Springer-Verlag. (Cited on pages 34, 61, 65, 67, 69, 71, 73, 75, and 78.)

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 page 49.)

Brèche, P. and Wörner, M. (1995). How to remove a class in an object database system. In *ADB 95: Proceedings of the 2nd International Conference on Applications of Databases*, pages 476–495. (Cited on page 34.)

Brereton, P., Kitchenham, B. A., Budgen, D., Turner, M., and Khalil, M. (2007). Lessons from applying the systematic literature review process within the software engineering domain. *J. Syst. Softw.*, 80:571–583. (Cited on page 22.)

Brun, C. and Pierantonio, A. (2008). Model differences in the Eclipse modelling framework. *UPGRADE, The European Journal for the Informatics Professional*, IX, issue No. 2:29–34. (Cited on pages 114 and 131.)

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 pages 60 and 63.)

Casais, E. (1995). *Managing class evolution in object-oriented systems*, chapter 8, pages 201–244. Prentice Hall International (UK) Ltd. (Cited on pages 4, 8, 16, 32, 36, 57, and 109.)

Cicchetti, A., Ruscio, D. D., Eramo, R., and Pierantonio, A. (2008). Automating co-evolution in model-driven engineering. In *Enterprise Distributed Object Computing Conference (EDOC 2008)*. IEEE. (Cited on pages 6, 46, 57, 61, 65, 67, 69, 71, 73, 75, 78, 81, 106, 114, and 131.)

Cicchetti, A., Ruscio, D. D., and Pierantonio, A. (2009). Managing dependent changes in coupled evolution. In *ICMT2009 - International Conference on Model Transformation*, LNCS, pages 35–51. Springer. (Cited on pages 46, 110, and 131.)

Clamen, S. M. (1994). Schema evolution and integration. *Distributed and Parallel Databases*, 2(1):101–126. (Cited on pages 35 and 49.)

Clark, J., DeRose, S., et al. (1999). XML Path Language (XPath). W3C Recommendation 16. (Cited on page 141.)

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. ACM. (Cited on page 34.)

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 *LNCS*, pages 893–917. Springer Berlin / Heidelberg. (Cited on pages 34, 61, 65, 67, 69, 71, 73, 75, and 78.)

Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387. (Cited on page 29.)

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 pages 36 and 49.)

Cunha, A., Oliveira, J., and Visser, J. (2006). Type-safe two-level data transformation. In *Formal Methods Europe (FME 2006)*, volume 4085 of *LNCS*, pages 284–299. Springer. (Cited on pages 106, 145, and 153.)

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, pages 323–332. (Cited on page 31.)

Curino, C., Moon, H. J., and Zaniolo, C. (2008b). Graceful database schema evolution: the PRISM workbench. *Proceedings of the VLDB Endowment*, 1(1):761–772. (Cited on page 31.)

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 page 31.)

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. ACM. (Cited on pages 114 and 131.)

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. ACM. (Cited on page 130.)

Dig, D. and Johnson, R. (2006). How do APIs evolve? a story of refactoring. *J. Softw. Maint. Evol.*, 18(2):83–107. (Cited on pages 61, 65, 67, 69, 71, 73, 75, and 78.)

Draheim, D., Horn, M., and Schulz, I. (2004). The schema evolution and data migration framework of the environmental mass database IMIS. In *SSDBM 04: 16th International Conference on Scientific and Statistical Database Management*, pages 341–344. IEEE Computer Society. (Cited on page 34.)

Eclipse Foundation (2008). Eclipse Modeling Framework Project (EMF). http://eclipse.org/emf. (Cited on page 153.)

Falleri, J.-R., Huchard, M., Lafourcade, M., and Nebut, C. (2008). Metamodel matching for automatic model transformation generation. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 326–340. Springer-Verlag. (Cited on pages 114 and 131.)

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 pages 137 and 146.)

Favre, J.-M. (2005). Languages evolve too! changing the software time scale. In *IWPSE 05: Eighth International Workshop on Principles of Software Evolution*, pages 33–42. IEEE. (Cited on pages 16, 57, and 109.)

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. Morgan Kaufmann Publishers Inc. (Cited on page 34.)

Fowler, M. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc. (Cited on pages 61, 65, 67, 69, 70, 71, 73, 75, and 78.)

G. de Geest and S. D. Vermolen and A. van Deursen and E. Visser (2008). Generating version convertors for domain-specific languages. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 197–201. IEEE Computer Society. (Cited on pages 14 and 46.)

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 *LNCS*, pages 34–49. Springer Berlin / Heidelberg. (Cited on pages 6, 7, 46, 57, 81, 114, and 131.)

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 pages 39 and 161.)

Godfrey, M. and Zou, L. (2005). Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, pages 166–181. (Cited on page 130.)

Gruschko, B., Kolovos, D. S., and Paige, R. F. (2007). Towards synchronizing models with evolving metamodels. In *CSMR 07: Workshop on Model-Driven Software Evolution*. (Cited on pages 7, 45, 137, 140, 146, and 152.)

Guerrini, G. and Mesiti, M. (2008). X-Evolution: A comprehensive approach for XML schema evolution. In *DEXA 08: 19th International Conference on Database and Expert Systems Application*, pages 251–255. IEEE. (Cited on pages 7 and 42.)

Guerrini, G., Mesiti, M., and Sorrenti, M. A. (2007). XML schema evolution: Incremental validation and efficient document adaptation. In *Database and XMLTechnologies*, volume 4704 of *LNCS*, pages 92–106. Springer Berlin / Heidelberg. (Cited on page 42.)

Gupta, A., Mumick, I. S., and Subrahmanian, V. S. (1993). Maintaining views incrementally. In *International conference on management of data (SIGMOD 1993)*, pages 157–166, New York, NY, USA. ACM. (Cited on pages 105, 145, and 153.)

Hainaut, J.-L., Tonneau, C., Joris, M., and Chandelon, M. (1994). Schema transformation techniques for database reverse engineering. In *Proceedings of the 12th Intl. Conf. on the Entity-Relationship Approach (ER 1993)*, pages 364–375, London, UK. Springer-Verlag. (Cited on page 105.)

Halevy, A. Y. (2001). Answering queries using views: A survey. *The VLDB Journal*, 10:270–294. (Cited on page 30.)

Hearnden, D., Lawley, M., and Raymond, K. (2006). Incremental model transformation for the evolution of model-driven systems. In *Models in Software Engineering*, volume 4199 of *LNCS*, pages 321–335. Springer. (Cited on page 146.)

Herrmannsdoerfer, M., Benz, S., and Juergens, E. (2008). Automatability of coupled evolution of metamodels and models in practice. In *MODELS 08: Model Driven Engineering Languages and Systems*, volume 5301 of *LNCS*, pages 645–659. Springer Berlin / Heidelberg. (Cited on pages 43, 61, and 62.)

Herrmannsdoerfer, M., Benz, S., and Juergens, E. (2009). COPE - automating coupled evolution of metamodels and models. In *ECOOP 2009 - Object-Oriented Programming*. Springer. (Cited on pages 10, 46, 57, 58, 61, 62, 65, 67, 69, 71, 73, 75, 78, 80, 106, and 161.)

Herrmannsdoerfer, M., Ratiu, D., and Wachsmuth, G. (2010a). Language evolution in practice: The history of GMF. In *Software Language Engineering*, volume 5969 of *LNCS*, pages 3–22. Springer Berlin / Heidelberg. (Cited on pages 46, 61, 62, 65, 67, 69, 71, 73, 75, 78, 80, and 133.)

Herrmannsdoerfer, M., Vermolen, S. D., and Wachsmuth, G. (2010b). An extensive catalog of operators for the coupled evolution of metamodels and models. In *Software Language Engineering, Third International Conference (SLE 2010)*, LNCS. Springer. (Cited on pages 13 and 47.)

Herrmannsdoerfer, M., Vermolen, S. D., and Wachsmuth, G. (2011). Coupled software language evolution – a survey across technical spaces –. *ACM Computing Surveys*. Submitted for publication. (Cited on page 13.)

Herrmannsdörfer, M. (2007). Metamodels and models. Master's thesis, München University of technology, München, Germany. (Cited on page 153.)

Hibernate (2008). Relational persistence for Java and .NET. http://www.hibernate.org. (Cited on pages 10 and 142.)

Hilderman, R. and Peckham, T. (2007). Statistical methodologies for mining potentially interesting contrast sets. In *Quality Measures in Data Mining*, volume 43 of *Studies in Computational Intelligence*, pages 153–177. Springer Berlin / Heidelberg. (Cited on page 50.)

Hößler, J., Soden, M., and Eichler, H. (2005). Coevolution of models, metamodels and transformations. In Bab, S., Gulden, J., Noll, T., and Wieczorek, T., editors, *Models and Human Reasoning*, pages 129–154, Berlin. Wissenschaft und Technik Verlag. (Cited on pages 46, 106, 134, 141, and 146.)

Janssen, N. (2005). Transformation tool composition. Master's thesis, Institute of Information and Computing Sciences Utrecht University, Utrecht, The Netherlands. (Cited on page 141.)

Jürgens, E. and Pizka, M. (2006). The language evolver lever – tool demonstration –. *Electronic Notes in Theoretical Computer Science*, 164(2):55–60. LDTA 06: Proceedings of the Sixth Workshop on Language Descriptions, Tools, and Applications. (Cited on pages 7, 39, and 161.)

Kats, L. C. L., Kalleberg, K. T., and Visser, E. (2009). Domain-specific languages for composable editor plugins. In *Proceedings of the Ninth Workshop on Language Descriptions, Tools, and Applications (LDTA 2009)*, Electronic Notes in Theoretical Computer Science. Elsevier Science Publishers. (Cited on page 103.)

Kim, W. (1990). *Introduction to object-oriented databases*. MIT Press. (Cited on page 29.)

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. Morgan Kaufmann Publishers Inc. (Cited on pages 35 and 49.)

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 pages 17, 20, 22, and 155.)

Kleppe, A. (2008). *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional. (Cited on page 15.)

Kniesel, G. and Koch, H. (2004). Static composition of refactorings. *Science of Computer Programming*, 52(1-3):9–51. (Cited on page 123.)

Kolovos, D., Di Ruscio, D., Pierantonio, A., and Paige, R. (2009). Different models for model matching: An analysis of approaches to support model differencing. In *Comparison and Versioning of Software Models, 2009. CVSM '09. ICSE Workshop on*, pages 1–6. (Cited on pages 114 and 131.)

Kurtev, I., Bézivin, J., and Aksit, M. (2002). Technological spaces: An initial appraisal. In *CoopIS, DOA Federated Conferences*. (Cited on pages 6 and 16.)

Lämmel, R. (2001). Grammar adaptation. In *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *LNCS*, pages 550–570. Springer Berlin / Heidelberg. (Cited on pages 39, 46, 63, 73, and 161.)

Lämmel, R. (2004). Coupled software transformations - extended abstract. In *First International Workshop on Software Evolution Transformations*. (Cited on pages 4, 16, 20, 105, 137, and 146.)

Lämmel, R. and Lohmann, W. (2001). Format evolution. In *RETIS 01: Proc. 7th International Conference on Reverse Engineering for Information Systems*, volume 155 of *books@ocg.at*, pages 113–134. OCG. (Cited on pages 42, 106, 137, 138, 145, and 153.)

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. Springer-Verlag. (Cited on pages 39 and 161.)

Lämmel, R. and Zaytsev, V. (2009b). Recovering grammar relationships for the Java language specification. In *SCAM '09: Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 178–186. IEEE Computer Society. (Cited on pages 39, 81, and 161.)

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. IEEE Computer Society. (Cited on page 35.)

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. IEEE Computer Society. (Cited on pages 35 and 49.)

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: Workshop on Intelligent Signal Processing*, volume 17. (Cited on page 43.)

Lehman, M. M. and Belady, L. A. (1985). *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA. (Cited on page 4.)

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)*, pages 172–181. IEEE Computer Society. (Cited on page 34.)

Lerner, B. S. (2000). A model for compound type changes encountered in schema evolution. *ACM Trans. Database Syst.*, 25(1):83–127. (Cited on pages 31 and 34.)

Lerner, B. S. and Habermann, A. N. (1990). Beyond schema evolution to database reorganization. *SIGPLAN Not.*, 25(10):67–76. (Cited on page 34.)

Li, X. (1999). A survey of schema evolution in object-oriented databases. In *TOOLS 31: Technology of Object-Oriented Languages and Systems*, pages 362–371. IEEE. (Cited on page 37.)

Lin, Y., Gray, J., and Jouault, F. (2007). DSMDiff: a differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4):349–361. (Cited on page 130.)

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 pages 36 and 49.)

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 *ER 94: Business Modelling and Re-Engineering, 13th International Conference on the Entity-Relationship Approach*, volume 881 of *LNCS*, pages 132–151. Springer. (Cited on pages 36 and 49.)

Lopes, D., Hammoudi, S., and Abdelouahab, Z. (2006). Schema matching in the context of model driven engineering: From theory to practice. In *Advances in Systems, Computing Sciences and Software Engineering*, pages 219–227. Springer. (Cited on pages 114 and 131.)

Mens, T. and Van Gorp, P. (2006). A taxonomy of model transformation. *Electron. Notes Theor. Comput. Sci.*, 152:125–142. (Cited on page 44.)

Meyer, B. (1996). Schema evolution: Concepts, terminology, and solutions. *Computer*, 29(10):119–121. (Cited on page 37.)

Monk, S. and Sommerville, I. (1993). Schema evolution in OODBs using class versioning. *SIGMOD Rec.*, 22(3):16–22. (Cited on pages 35 and 37.)

Monk, S. R. and Sommerville, I. (1992). A model for versioning of classes in object-oriented databases. In *BNCOD 10: Proceedings of the 10th British National Conference on Databases*, volume 618 of *LNCS*, pages 42–58. Springer Berlin / Heidelberg. (Cited on pages 35 and 49.)

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 *LNCS*, pages 706–711. Springer Berlin / Heidelberg. (Cited on pages 44 and 81.)

Nguyen, G. T. and Rieu, D. (1989). Schema evolution in object-oriented database systems. *Data Knowl. Eng.*, 4(1):43–67. (Cited on page 32.)

Object Management Group (2006). Meta Object Facility (MOF) core specification version 2.0. `http://www.omg.org/spec/MOF/2.0/`. (Cited on pages 43, 58, 59, 80, 81, 132, 152, and 160.)

Object Management Group (2007). MOF QVT Final Adopted Specification. (Cited on page 152.)

Ohst, D., Welle, M., and Kelter, U. (2003). Differences between versions of uml diagrams. In *Proceedings of the 9th European software engineering conference*, ESEC/FSE-11, pages 227–236. ACM. (Cited on page 130.)

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. ACM. (Cited on page 39.)

Paige, R. F., Brooke, P. J., and Ostroff, J. S. (2007). Metamodel-based model conformance and multiview consistency checking. *ACM Transactions on Software Engineering Methodologies*, 16. (Cited on page 2.)

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. ACM. (Cited on page 32.)

Pizka, M. and Jürgens, 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. IEEE Computer Society. (Cited on pages 39 and 161.)

Pizka, M. and Jürgens, E. (2007b). Tool supported multi level language evolution. In *In Proceedings of SVM'07: Software and Services Variability Management Workshop Concepts, Models and Tools*. (Cited on pages 39, 105, 137, 146, and 161.)

Pons, A. and Keller, R. (1997). Schema evolution in object databases by catalogs. In *IDEAS 97: Database Engineering and Applications Symposium*, pages 368–376. (Cited on pages 37, 61, 65, 67, 69, 71, 73, 75, and 78.)

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. ACM. (Cited on pages 36 and 49.)

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. IEEE Computer Society. (Cited on pages 36 and 49.)

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 pages 36 and 49.)

Rahm, E. and Bernstein, P. (2001). A survey of approaches to automatic schema matching. *the VLDB Journal*, 10(4):334–350. (Cited on page 131.)

Rashid, A. and Sawyer, P. (2000). Object database evolution using separation of concerns. *SIGMOD Rec.*, 29(4):26–33. (Cited on pages 10, 35, and 49.)

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 pages 8, 10, 16, 32, and 35.)

Roddick, J. F. (1992). Schema evolution in database systems: an annotated bibliography. *SIGMOD Rec.*, 21(4):35–40. (Cited on pages 7, 8, 16, and 29.)

Roddick, J. F. (1995). A survey of schema versioning issues for database systems. *Information and Software Technology*, 37(7):383 – 393. (Cited on pages 30, 32, and 37.)

Ronström, M. (2000). On-line schema update for a telecom database. In *Data Engineering, 2000. Proceedings. 16th International Conference on*, pages 329–338. IEEE. (Cited on page 30.)

Rose, L. M., Kolovos, D. S., Paige, R. F., and Polack, F. A. (2010). Model migration with Epsilon Flock. In *ICMT 10: Third International Conference on Theory and Practice of Model Transformations*, pages 184–198. Springer. (Cited on pages 44 and 134.)

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*, pages 6–15. (Cited on pages 8, 16, 39, 41, 44, 49, and 58.)

Shneiderman, B. and Thomas, G. (1982). An architecture for automatic relational database system conversion. *ACM Trans. Database Syst.*, 7(2):235–257. (Cited on page 30.)

Shvaiko, P. and Euzenat, J. (2005). A survey of schema-based matching approaches. In *Journal on Data Semantics IV*, volume 3730 of *LNCS*, pages 146–171. Springer Berlin / Heidelberg. (Cited on page 131.)

Sjøberg, D. (1992). Measuring schema evolution. Technical report, Technical report No. FIDE/92/36. FIDE, Dept. Computing Science, University of Glasgow, Glasgow, G12. (Cited on page 29.)

Sjøberg, D. (1993). Quantifying schema evolution. *Information and Software Technology*, 35(1):35–44. (Cited on page 29.)

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. ACM. (Cited on pages 35 and 49.)

Sockut, G. H. and Goldberg, R. P. (1979). Database reorganization—principles and practice. *ACM Comput. Surv.*, 11(4):371–395. (Cited on page 31.)

Soley, R. et al. (2000). Model driven architecture. OMG white paper 308. (Cited on page 147.)

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 page 44.)

Sprinkle, J. M. (2003). *Metamodel driven model migration*. PhD thesis, Vanderbilt University. (Cited on pages 4, 7, 31, 44, and 109.)

Staples, M. and Niazi, M. (2007). Experiences using systematic review guidelines. *J. Syst. Softw.*, 80:1425–1437. (Cited on page 22.)

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. ACM. (Cited on pages 7, 39, and 161.)

Steinberg, D., Budinsky, F., Paternostro, M., and Merks, E. (2009). *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley. (Cited on pages 43, 59, 132, and 160.)

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 *LNCS*, pages 431–444. Springer Berlin / Heidelberg. (Cited on page 43.)

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. IEEE Computer Society. (Cited on pages 7 and 42.)

Sun, X. L. and Rose, E. (2003). Automated schema matching techniques: An exploratory study. *Research Letters in the Information and Mathematical Science*, 4:113–136. (Cited on pages 109 and 131.)

Tan, M. and Goh, A. (2005). Keeping pace with evolving XML-Based specifications. In *EDBT 2004 Workshops: Current Trends in Database Technology*, volume 3268 of *LNCS*, pages 280–288. Springer Berlin / Heidelberg. (Cited on page 41.)

Tresch, M. and Scholl, M. H. (1993). Schema transformation without database reorganization. *SIGMOD Rec.*, 22(1):21–27. (Cited on pages 36 and 49.)

Tu, Q. and Godfrey, M. (2002). An integrated approach for studying architectural evolution. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 127–136. (Cited on page 130.)

van Sterkenburg, P. (2003). *A practical guide to lexicography*. John Benjamins Publishing Co. (Cited on page 19.)

Ventrone, V. and Heiler, S. (1991). Semantic heterogeneity as a result of domain evolution. *SIGMOD Rec.*, 20(4):16–20. (Cited on page 32.)

Vermolen, S. (2008). Software language evolution. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 323–326, Washington, DC, USA. IEEE Computer Society. (Cited on page 14.)

Vermolen, S. D. and Visser, E. (2008). Heterogeneous coupled evolution of software languages. In *Model Driven Engineering Languages and Systems (Models 2008)*, volume 5301 of *LNCS*, pages 630–644. Springer. (Cited on pages 14 and 37.)

Vermolen, S. D., Wachsmuth, G., and Visser, E. (2011). Generating database migrations for evolving web applications. In Denney, E. and Schultz, U. P., editors, *Generative Programming and Component Engineering, 7th International Conference, GPCE 2011, Portland, OR, USA, October 22-23, 2011, Proceedings*. ACM. (Cited on page 13.)

Vermolen, S. D., Wachsmuth, G., and Visser, E. (2012). Reconstructing complex metamodel evolution. In *Software Language Engineering, Fourth International Conference, SLE 2011, Braga, Portugal, Revised Selected Papers*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg. To Appear. (Cited on page 14.)

Visser, E. (1997). *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam. (Cited on page 146.)

Visser, E. (2004). Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer, C. et al., editors, *Domain-Specific Program Generation*, volume 3016 of *LNCS*, pages 216–238. Spinger. (Cited on pages 132 and 142.)

Visser, E. (2008a). WebDSL: A case study in domain-specific language engineering. In Lammel, R., Saraiva, J., and Visser, J., editors, *Generative and Transformational Techniques in Software Engineering (GTTSE 2007)*, LNCS. Springer. (Cited on pages 3, 83, 85, 132, 142, and 154.)

Visser, J. (2008b). Coupled transformation of schemas, documents, queries, and constraints. *Electron. Notes Theor. Comput. Sci.*, 200(3):3–23. (Cited on pages 4, 16, 20, and 106.)

Wachsmuth, G. (2007a). An adaptation browser for MOF. In *WRT'01: First Workshop on Refactoring Tools*, pages 65–66. (Cited on page 80.)

Wachsmuth, G. (2007b). Metamodel adaptation and model co-adaptation. In *ECOOP 2007 - Object-Oriented Programming*, volume 4609 of *LNCS*, pages 600–624. Springer Berlin / Heidelberg. (Cited on pages 6, 45, 46, 57, 58, 60, 63, 65, 67, 69, 71, 73, 75, 78, 106, 134, 137, 146, 152, and 153.)

Walmsley, P. (2001). *Definitive XML Schema*. Prentice Hall PTR. (Cited on page 40.)

Xing, Z. and Stroulia, E. (2005). Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 54–65. ACM. (Cited on pages 114 and 130.)

Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., and Mei, H. (2007). Towards automatic model synchronization from model transformations. In *Automated Software Engineering (ASE 2007)*, pages 164–173, New York, NY, USA. ACM. (Cited on page 146.)

Zicari, R. (1991). A framework for schema updates in an object-oriented database system. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 2–13. IEEE. (Cited on page 34.)

Samenvatting

SOFTWARETAAL EVOLUTIE

– Sander D. Vermolen –

Een groot deel van de informatica richt zich op de ontwikkeling van applicaties. Dit kunnen reguliere applicaties zijn, die een gebruiker opstart, gebruikt en weer afsluit, maar ook bijvoorbeeld web applicaties, de beheersing van je printer, of de besturing van een vliegtuig. In hun kern zijn applicaties niets meer dan een lange reeks 0-en en 1-en, welke door een computer worden omgezet naar concrete acties. Ontwikkeling van applicaties omvat primair het ontwikkelen van zulke reeksen 0-en en 1-en, maar het handmatig schrijven ervan is onpraktisch, te ingewikkeld en duurt te lang. In de beginjaren van de computers was dit handmatig schrijven nog nodig, maar over de jaren heen zijn er programmeertalen ontwikkeld. Een programmeertaal levert een toegankelijke manier om applicaties, ofwel programma's te schrijven. Programma's lijken vaak op een sterk gestructureerde en beperkte vorm van de Engelse taal. Een programma in een programmeertaal kan automatisch vertaald worden naar een reeks 0-en en 1-en die dan weer door een computer begrepen kunnen worden.

Gelijk aan het breed spectrum van natuurlijke talen, is er een, wellicht nog breder, spectrum aan programmeertalen. Programmeertalen vertonen over het algemeen grote verschillen. Als hetzelfde programma geschreven wordt in twee verschillende programmeertalen, dan zal het er hoogst waarschijnlijk anders uitzien. Net als dat een boek in de Nederlandse taal eenvoudig te onderscheiden is van hetzelfde boek in de Franse taal. Ze gebruiken andere woorden en een andere grammatica, of in bredere zin, een andere 'structuur'. Waar we bij een natuurlijke tekst zouden zeggen dat deze "in het Nederlands is geschreven", zeggen we dat een programma conformeert aan zijn programmeertaal.

Een veel gebruikte term voor computerprogramma's is software. Het gebruik van deze term is niet geheel juist. Alhoewel een computer programma een variant software is, beslaat de term software een breder domein. Software is de tegenhanger van hardware en slaat op informatie (gewoonlijk computergerelateerd) die eenvoudig te wijzigen is. Dit is een breed domein en omvat bijvoorbeeld ook foto's, films, gegevens die je verstuurt via internet, een tekstdocument, enzovoorts. Al deze software heeft gemeen dat ze een bepaalde structuur hebben: Voor foto's en films zijn er verschillende bestandsformaten, voor informatie die je verstuurt via internet gelden zogenaamde protocollen die exact vastleggen hoe de gegevens eruit moeten zien, een tekst document is gewoonlijk geschreven in Engelse tekens, enzovoorts. Net als dat een programmeertaal de structuur van een programma vastlegt, legt een softwaretaal

in bredere zin, de structuur van een stuk software vast. We zeggen dat software conformeert aan een softwaretaal.

Eén van de belangrijke eigenschappen van software is dat het eenvoudig te veranderen is. Dit heeft als direct gevolg dat het daadwerkelijk vaak verandert. Programma's veranderen bijvoorbeeld als er fouten in gerepareerd moeten worden, of als er functionaliteit moet worden toegevoegd. Verandering van software vindt over het algemeen over een langere periode en in veel stappen plaats. We spreken dan van software evolutie.

Net als software, veranderen software talen ook. Betere inzichten en technische ontwikkeling maken het vaak nodig om een taal aan te passen. Bij natuurlijke talen zien we hetzelfde proces: Het Nederlandsch van de middeleeuwen is anders dan het hedendaags Nederlands. De evolutie binnen de informatica vindt echter vele malen sneller plaats dan de evolutie bij natuurlijke talen. We spreken meestal over maanden of weken in de informatica, waar we over eeuwen spreken bij natuurlijke taal. Evolutie van een softwaretaal, heeft als gevolg dat bestaande software niet meer conformeert aan de taal. De software wordt daardoor moeilijk leesbaar. Immers, het lezen van een middeleeuws boek is voor ons lastiger dan het lezen van een hedendaagse tekst. Voor software, waar taal regels veel strikter worden gehanteerd, is software met een geëvolueerde taal, niet meer bruikbaar.

Om de gevolgen van softwaretaal evolutie tegen te gaan, moet software vertaald worden van de oude taal-variant naar de nieuwe taal-variant. We noemen dit migratie. Migratie kan handmatig uitgevoerd worden als de verandering van de taal bekend is. Maar vaak is de software is te groot voor een handmatige migratie. Daarom worden er programma's geschreven die de migratie uitvoeren. Het schrijven van zulke programma's is veel werk. Evolutie wordt daarom vaak zoveel mogelijk voorkomen. Het tegengaan, of vermijden van evolutie van een taal heeft echter tot gevolg dat de kwaliteit van de taal langzaamaan afneemt. Het wordt steeds lastiger om software te schrijven en te verwerken. Een betere ondersteuning voor de evolutie van software talen voorkomt dit.

Een van de aanpakken om softwaretaal evolutie beter te ondersteunen is gekoppelde evolutie. Hierbij worden vaak-voorkomende softwaretaal evolutiepatronen gekoppeld aan geschikte migraties (vertalingen) van software. Met gekoppelde evolutie kunnen evolutiestappen tegelijk met de migratie worden toegepast. Dit vereenvoudigt het proces en biedt de mogelijkheid om zulke koppelingen te hergebruiken als in de toekomst een soortgelijke evolutiestap optreed.

Gekoppelde evolutie van software en software talen komt voor in verschillende domeinen van de informatica: In de programmaontwikkeling vereist een evolutie van programmeertalen een migratie van programma's; in de dataverwerking vereist evolutie van datamodellen een migratie van databases; en in de modellering vereist evolutie van metamodellen een migratie van modellen.

Dit proefschrift bespreekt verschillende onderwerpen in de context van gekoppelde evolutie van software talen en software in verschillende domeinen van de informatica. Het richt zich op vragen als: Hoe evolueert een softwaretaal? Hoe kunnen we softwaretaal evoluties koppelen aan software migraties? Hoe kunnen we een gepasseerde softwaretaal evolutie reconstrueren? En hoe kunnen we het softwaretaal evolutie probleem generaliseren over de verschillende domeinen?

Naast een aantal wetenschappelijke publicaties en nieuwe inzichten in gekoppelde evolutie van software talen en software, heeft het onderzoek geresulteerd in een tool set voor gekoppelde evolutie van web applicaties en hun databases, genaamd Acoda[6]. Daarnaast heeft het onderzoek verschillende case studies opgeleverd, welke concrete en realistische informatie bieden over softwaretaal evolutie in de praktijk.

[6]http://swerl.tudelft.nl/bin/view/Acoda

Curriculum Vitae

PERSONAL DATA

Full name Sander Daniël Vermolen
Date of Birth 19^{th} September 1984
Place of Birth Arnhem, The Netherlands

EDUCATION

Ph.D. in Computer Science (2007 – 2012)
Delft University of Technology

M.Sc. in Computer Science (2005 – 2007)
Radboud University Nijmegen & Engineering College of Aarhus
Graduated with Honors

B.Sc. in Computer Science (2002 – 2005)
Radboud University Nijmegen
Graduated with Honors

EMPLOYMENT

2003 – 2007 Taught various courses at Radboud University Nijmegen
Faculty of Science, Radboud University Nijmegen, Heyendaalseweg 135, 6525 AJ Nijmegen, The Netherlands.

2007 – 2011 Assistent In Opleiding (AIO), Research Trainee
Software Technology Department, Delft University of Technology, Mekelweg 4, 2628 CD Delft, The Netherlands.

From 2012 Design Engineer – Development & Engineering
Software Metrology, ASML, De Run 6501, 5504 DR Veldhoven, The Netherlands.

Titles in the IPA Dissertation Series

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of*

Empirical Studies about the UML. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wild-*

cards. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science,

Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of

Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transfor-*

mations. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

M. Izadi. *Model Checking of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-22

L.C.L. Kats. *Building Blocks for Language Workbenches*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2011-23

S. Kemper. *Modelling and Analysis of Real-Time Coordination Patterns*. Faculty of Mathematics and Natural Sciences, UL. 2011-24

J. Wang. *Spiking Neural P Systems*. Faculty of Mathematics and Natural Sciences, UL. 2011-25

A. Khosravi. *Optimal Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2012-01

A. Middelkoop. *Inference of Program Properties with Attribute Grammars, Revisited*. Faculty of Science, UU. 2012-02

Z. Hemel. *Methods and Techniques for the Design and Implementation of Domain-Specific Languages*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-03

T. Dimkov. *Alignment of Organizational Security Policies: Theory and Practice*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-04

S. Sedghi. *Towards Provably Secure Efficiently Searchable Encryption*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2012-05

F. Heidarian Dehkordi. *Studies on Verification of Wireless Sensor Networks and Abstraction Learning for System Inference*. Faculty of Science, Mathematics and Computer Science, RU. 2012-06

K. Verbeek. *Algorithms for Cartographic Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2012-07

D.E. Nadales Agut. *A Compositional Interchange Format for Hybrid Systems: Design and Implementation*. Faculty of Mechanical Engineering, TU/e. 2012-08

H. Rahmani. *Analysis of Protein-Protein Interaction Networks by Means of Annotated Graph Mining Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2012-09

S.D. Vermolen. *Software Language Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2012-10