TUCS

Marcus Alanen

A Metamodeling Framework for Software Engineering

Turku Centre for Computer Science

TUCS Dissertations

No 89, May 14, 2007

A Metamodeling Framework for Software Engineering

Marcus Alanen

To be presented, with the permission of the Faculty of Technology at Åbo Akademi University, for public criticism in Auditorium Alpha at the ICT building, Turku, Finland, on June 1st, 2007, at 12 noon.

Åbo Akademi University
Department of Information Technologies
Joukahaisenkatu 3–5
20520 Turku
Finland

2007

Supervisors

Academy Professor Ralph-Johan Back
Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3–5
20520 Turku
Finland

Docent Ivan Porres
Department of Information Technologies
Åbo Akademi University
Joukahaisenkatu 3–5
20520 Turku
Finland

Reviewers

Professor Jean Bézivin
Department of Computer Science
University of Nantes
2, rue de la Houssiniére, BP 92208
44322 Nantes Cedex 3
France

Professor Bernhard Rumpe
Software Systems Engineering Institute
Braunschweig University of Technology
Mühlenpfordtstraße 23, 3. OG
D-38106 Braunschweig
Germany

Opponent

Professor Jean Bézivin
Department of Computer Science
University of Nantes
2, rue de la Houssiniére, BP 92208
44322 Nantes Cedex 3
France

ISBN 978-952-12-1901-6
ISSN 1239-1883

Abstract

The software engineering discipline strives for techniques and tools for developing software faster, cheaper and more reliably. In its most prevalent form today, software is developed using text-based programming languages described using Backus-Naur Form (BNF) grammars. The current solutions to the software engineering goals consist of several different tools that perform different operations on the BNF syntax tree of a program text. The tools analyze, verify, test, compile and transform it. Unfortunately, many project artifacts are seldom linked to the programs, such as requirements documents, error reports and test data. We postulate that this is partially due to the limitations in representing programs as syntax trees, and that this hinders the development of better tools.

Another way of describing these artifacts is the use of graphs, where graph grammars constrain which kinds of graphs can be built. By using graphs, we are not as restricted to describe information when compared with a BNF syntax tree. Software modeling is a research area which purports to address the aforementioned goals by using different graphs, called models in the discipline, for all relevant artifacts in a software project. The standards body for defining software modeling is the Object Management Group (OMG), a consortium consisting of both industrial and academic participants.

If software modeling is to become as mainstream as program development in the textual domain, it must provide a development infrastructure based on a sound theory that solves common software engineering issues, as well as provide improved solutions to some relevant issues when compared with the textual domain. In this thesis, we aim to lift the foundations of software modeling to the same level as software engineering using textual programming languages and integrated development environments. We accomplish this by analyzing some current OMG standards for software modeling, finding omissions and errors and suggesting improvements. Therefore this thesis covers a broad range of topics: metametamodel constructs, serialization technology, version control, and abstract and concrete syntax of models.

A contribution of this thesis is a set-theoretical metamodeling framework supporting subset and union properties with static constraints over the metamodels and models, as well as pre- and postconditions and implementations of model operations respecting these constraints. Furthermore, we assess the suitability of the

XML Metadata Interchange (XMI) serialization technology with respect to various usage scenarios for model interchange. The core of software configuration management is version control, and we show algorithms for calculating the difference between two arbitrary models, applying a difference and inverting a difference. We also discuss conflict resolution when merging several differences together. Finally, we present a domain-specific weaving metamodel that supports reconciling the concrete syntax of models based on changes in the abstract syntax.

We have deliberately laid an emphasis on the engineering aspect of the solutions described in the thesis. The research has been validated by implementing and testing the contributions and suggestions as a working open source prototype tool called Coral.

Acknowledgements

I would like to thank my supervisors Docent Ivan Porres and Academy Professor Ralph-Johan Back for their steadfast confidence in, and support for, me. I have been privileged to work closely with Ivan, whose concentration and goal-oriented approach I very much admire. Ivan has also taught me a more methodical way to work, and Ralph has patiently taught me how software engineering can benefit from computer science.

I highly appreciate that Professor Bernhard Rumpe and Professor Jean Bézivin took time off from their busy schedules and did a fantastic job in reviewing this thesis. I would like to thank Jean for accepting the task of being my opponent at the public defense.

I am grateful that Professor Johan Lilius offered me a job at the department early during my undergraduate studies. At the time, I did not expect to undertake a PhD degree, but I am happy that Johan took the initiative for my academic career so early.

I would especially like to thank Torbjörn, who has diligently worked on implementing our ideas and whose cheerfulness was always welcome. In times of mathematical trouble, Patrick has been astoundingly patient and helpful in explaining set theory and related branches of mathematics to me. My thanks to Johannes for our collaboration on Gaudí projects and for many interesting discussions; I sincerely hope they will not be our last ones. During my whole studies, I have had many a fun pizza evening with Kim.

The practical contributions of this thesis have received numerous help from undergraduate students working on the SMW and Coral projects. I would also like to thank Bahareh for the modeling tool comparisons in Chapter 6. Also, we have had the luxury of people using the Coral tool. This feedback has both steered our research of modeling and development of Coral, as well as improved their quality. Thanks for this go to Dragoş, the SOCOS and MICAS teams, and to Ian Oliver at Nokia Research Center.

I would like to thank the department and TUCS for providing me with a salary and the financial aid to travel to several conferences. It has been truly fantastic to meet smart people in nice places around the world; especially my thanks to Laurie. I am grateful for the friendship of my colleagues and friends who have made working at the department much more fun: Luka and Riia, Maria, Venja, Petri, Linda, Robert L., Robert G., Mats, Andreas, Stefan, Fredrik, Ulf, Viorel, Cristina, Orieta and Herman. Thanks to Christel for keeping the department running, and to Nina for taking care of the publications.

I am honored to have received two scholarships from the Nokia Foundation.

In moments of work frustration, the Nethack DevTeam has provided the greatest of games—and timesinks. Other recreational activities have included volleyball with the wonderful people at the Åbo Akademi personnel's volleyball practices and the JaLPa team.

I would also like to express my deep and sincere appreciation to several friends: Sanna, Mathias, Anna, Torulf, CaGe, Henry, Camilla, the people at #58, and many others, for just being there and for having the courtesy of not asking too many questions about the thesis during the whole ordeal. I am thankful for the love and support of my family.

Turku, April 30th, 2007

Marcus Alanen

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Textual Programming

Despite years of experience, successful software engineering is still considered to be a very difficult problem [156]. The execution of any software development project includes the creation and maintenance of many types of documents: requirements, specifications, tasks and project plans, actual source code, test reports, et cetera. It is important to understand that an executable computer program is not our greatest concern. Many of the documents—artifacts—mentioned are not computer programs, yet all of them are highly relevant to planning, building and maintaining one. Furthermore, all of these artifacts relate to each other, but are usually described and manipulated using different languages, data formats and tools. Interoperability between tools would improve the pace and correctness of program development. Overcoming problems between tools or data formats can require a significant amount of effort.

Therefore we postulate that it should be possible to describe all of the relevant artifacts using several interoperable languages. This implies two things. One, that an artifact can be described using the constructs of some language. That is, there are constructs at our disposal which are sufficient to model the information of the artifact. Two, that these languages can be combined and used together, where such usage makes sense in the domain of the languages. In other words, that the constructs for creating languages are common building blocks across all languages.

Informally we can understand that these common building blocks also form a language, the language of creating languages. Thus, we have what is called a *metalanguage* to describe languages, and since the metalanguage itself is a language, it can be used to describe itself. This feature is called *metacircularity* [7, 10]. The benefit of a metalanguage is that all artifacts can be manipulated uniformly, since they all conform to some language, which all conform to the metalanguage.

The mathematical definition of a language in word theory is the set of words that is accepted. A compact way of expressing these words is the use of a (word) grammar. Noam Chomsky and Marcel-Paul Schützenberger categorized formal grammars in the Chomsky-Schützenberger hierarchy in 1956 [38], to which context-free grammars belong. A programming language is usually defined as such a context-free grammar, and described using Backus-Naur Form (BNF) or the semantically equivalent various Extended Backus-Naur Forms (EBNF). BNF was developed by John Backus and Peter Naur in the end of the 1950's for the ALGOL 60 programming language [4].

There are many benefits to using BNF, but its use often requires additional information to be constructed together with the parsing of the input into a syntax tree. In layman's terms, consider for example the sentences *"The hen panicked. The red fox attacked it."* We cannot infer what the fox attacked without reading and understanding the first sentence. This is information that a (hypothetical) BNF grammar for English does not convey. This complicates the understanding of the information, since the meaning of the pronouns are context-dependent. At worst, we have to read the whole text from the beginning, remembering subjects, objects and actions taken up to and including the last sentence of interest to understand to what the relative pronouns refer. Thus we also have a sense of direction in how we must read the text for the information to make sense.

We see similar issues in the textual programming languages and programming environments of today, although a lot of effort has been put into mitigating these problems. On one hand, we have the extreme case of the C++ [170] language, the designers of which had to keep compatibility with C [91] whilst adding object-oriented concepts, the end result being a very complicated syntax. This in turn led to a proposal for a different syntax for the same language [204], which has not been adopted. A more modern example is the Java [64] programming language, where for example JavaDoc documentation is written inside comments before methods, and then extracted by a separate tool, and JML [36], which allows the developer to write pre- and postconditions to methods inside comments. Instead of changing the BNF grammar of Java, we have circumvented it and overloaded the meaning of comment strings: are they comments, JavaDoc documentation or JML expressions? On the other hand, an example of a tool which successfully incorporates the compiler into the integrated development environment is Eclipse with its on-the-fly Java compiler [76].

There is also a problem of representation. Although a BNF grammar describes a syntax tree, the source code is written as a stream of characters. In other words, there is a difference between the concrete syntax and the abstract syntax of a language, and a grammar cannot only support describing a syntax tree. It must also be augmented to support parsing from a stream of data. There also exists popular languages which do not have a pure BNF grammar. For example, anecdotally the grammar of Perl [201] cannot be reduced to BNF, and a parser of a Python [180] program needs special support because the language has context-dependent whitespace. Yet another problem is that of abstraction. Because of the relatively simple data structures used, almost the only artifact used to describe programs is source code. Our anecdotal experience is that relatively few programming languages are concerned about supporting refinement [15] or Design by Contract (DbC) [116], and any such support is added later, or not standardized. Eiffel [117] is a notorious exception. Its designer Bertrand Meyer introduced the concept of DbC in the 1980's and created a new language to support it.

1.2 Graphs

However, if we consider the sentences about the hen and the fox to be analyzed prior to any other interpretation, so that the meaning of pronouns were substituted by links to their actual meaning and successive words were also linked, we suddenly obtain a different mathematical structure: a *graph* consisting of words and links, or *nodes* and *edges*. Graphs also have the equivalent of word grammars, called graph grammars, which define which graphs are allowed.

These might not seem to be big changes, but the consequences are rather profound. First, this encodes the meaning of words in a sentence more explicitly, since we can follow a link from a pronoun to its meaning. In fact we might not use pronouns at all, but instead link directly to the meaning from the verb. Second, we lose our sense of direction, in that there is no obvious starting point. We can check what happened to the hen by examining the links that point to it, but we could just as well start from the fox. Third, we can add more detail by adding new links to nodes, thus refining our data to be more precise. In this way, we can use graphs on multiple levels of abstraction and examine our artifacts at the level of detail we are interested in.

For this thesis, we claim as our underlying premise that artifacts relevant to a software development project can be represented as graphs, and that graphs are in fact better primitives for describing information than syntax trees. The benefit is that graph theory has a very strong and well-understood mathematical foundation. However, the claim is an important assumption and the contents of this thesis—and current research on software modeling as well—rely heavily on it.

In general, a graph consists of two kinds of elements, *edges* and *nodes*, which can be *typed*, *attributed* and *hierarchical*. The type of an element determines its classification. All elements of the same type can be seen as having some structural or semantic commonality. Attributes are key-value pairs of primitive type such as strings which describe the element further. Allowing an element to include other elements into itself supports hierarchical graphs. This concept is usually seen on two different levels, as an ownership relation between elements and between types; the latter concept is used for collecting types into *packages*.

We consider that each edge connects n elements together. If $n > 2$ the edge is said to be a *hyperedge*. An edge can also be *directed* which splits the n connections into two nonempty sets, one considered the *source* collection of elements, and the other the *target* collection. Very often edges may only connect to nodes, not to other edges. Edges in more specialized or complex graphs have additional properties which describe the *ownership* between the source node(s) and the target node(s). Edges are often grouped into specific categories depending on which kinds of nodes they interconnect and what the semantics of the edge is. These groups can then be considered *ordered* or *unordered* and the *multiplicity* (number) of the edges in a group is of importance. If *multiple edges* between the same source and target node are allowed, the group can be considered a *bag* or *multiset* instead of a *set*.

The graphs that represent an artifact should conform to a grammar, i.e., a formal definition of all graphs or models allowed in a given language. We say that a graph is an *instance* of a grammar, which provides a *type system* that the graphs must follow. The grammars can also be represented as graphs.

1.3 Modeling

We will not delve here on which ways the differences in the leap from syntax trees to graphs are advantages and in which ways they are disadvantages. Nevertheless, *software modeling* is, to us, the use of interlinked graphs and graph grammars for purposes of creating, analyzing, transforming and maintaining software engineering artifacts over multiple levels of abstraction.

In the long run, standardization of computer technology has given many benefits. For example, BNF made it easier to describe the syntax of languages and compiler-compilers like yacc [101] made it easier to create parsers. We also have standards such as Simple Mail Transfer Protocol (SMTP) for electronic mail and Hypertext Markup Language (HTML) for World Wide Web (WWW) pages, the use of which has become ubiquitous. It can be assumed that if modeling were to become widespread in software engineering, it must be standardized to some extent.

The industry consortium setting the modeling standards is the Object Management Group (OMG) [123]. Their modeling effort is coined under the umbrella term Model Driven Architecture (MDA) [143]. Their suggestion is that instead of using BNF to build grammars, we should use the Meta Object Facility (MOF) [139]. Instead of BNF grammars, we should build metamodels, and their flagship metamodel is the Unified Modeling Language (UML) [135]. Thus the artifacts that are produced are not text and syntax trees but models. The pairs we have mentioned are at different metalevels and are listed in Table 1.1. An additional metalevel is the lowest metalevel common to both text-based and modeling technologies: the runtime metalevel which actually contains a program that is being executed.

MDA is an initiative of a more general technology called Model Driven Engineering (MDE) [89]. MDE advocates the use of models to represent all the relevant information in a software development project. Software development is then carried out as a sequence of model development and transformation steps. MDE is

Metalevel	**Text-Based**	**OMG**
Metalanguage	BNF	MOF
Languages	Java, C++, Python, etc.	UML
Artifact	Syntax tree	Models

Table 1.1: Traditional Text-Based Metalevels and Their OMG Modeling Equivalents

the result of the recent development of modeling languages, awareness of the need for software development methodologies and the constant need to tackle larger and more complex development projects.

We believe that MDE opens a window for new development methods and tools that are not available or are too expensive to implement in other approaches such as text-based driven development. However, MDE also presents new challenges that should be addressed before the approach can be used in practice. We are faced with many issues if we were to use modeling technologies. Many of these are already known from textual programming languages. These issues must be solved or reverified to work properly in the modeling domain so that the benefits we have acquired in the textual domain are transferred to the modeling domain.

Information about the various issues have been acquired from the following sources. The standards themselves, made by OMG, lay the ground for a modeling framework around which much of the research activity also exists. Publications about OMG standards and suggested alternatives provide novel ideas and discover problems, either in OMG standards or in the field of software engineering. Finally, our own experience in software engineering and programming aids us informally in which problems to tackle and what kinds of solutions are preferable.

In practice, the core of a modeling environment is a model manager. It consists of two parts: a metamodeling language and a tool infrastructure. The metamodeling language is a definition that lets us create metamodels and models, and a tool interface lets us create, query, modify and serialize metamodels and models. The research on model managers, implementing them and assessing their usability is relevant because not only are we interested in the theoretical benefits that graphs might provide, we are also interested in improvements and solutions bringing a real-world benefit to the practitioners of software engineering.

Thus, the research approach followed in this thesis can be summarized as follows. In parallel, we have strived to read relevant OMG standards and scientific publications, discover issues and develop solutions in a theoretical context as well as in a working tool. These results have been published in scientific conferences and journals.

We have deliberately laid an emphasis on the engineering aspect of these solutions, and quite little is formally proved in the thesis. Instead we strive to validate the research by a solution that actually runs on a computer and can be tried out in practice to assess how well it works and whether or not it is usable. We consider this an important aspect of the work presented in the thesis.

However, many issues are left unaddressed by modeling technology. Creating a software system is still a fundamentally hard problem. Modeling aims to remove some of the complexity of creating software, but the underlying inherent complexity of designing a system meeting a set of customer requirements is still left. Frederick P. Brooks stated this problem in 1986 [25]:

> I believe the hard part of building software to be the specification, design and testing of this conceptual construct [of a software entity], not the labor of representing it and testing the fidelity of the representation.

He continues with:

> If this is true, building software will always be hard. There is inherently no silver bullet.

However, it is the belief of the author of this thesis that modeling, most likely in combination with textual programming languages, can lower the threshold for solving the *essential complexity* in software by making it easier to build and maintain tools that reduce the amount of *accidental complexity*. Charles Connell summarized this in 2001 as follows [45]:

> If the study of software engineering helps us improve, by even a small amount, our ability to create software, the entire field justifies its existence.

A succinct summary of this thesis is that we wish to give software modeling the same benefits as traditional textual programming languages. Therefore we analyze some modeling standards from OMG from this perspective, discussing their weak and strong points. We suggest improvements where possible. We have strived to validate the work by implementing the suggested solutions and improvements.

1.4 Contributions of this Thesis

The contributions of this thesis range over several OMG standards and problems mentioned. We find several cases where the standards are underspecified. This must be addressed for software modeling to become as mainstream as programming in the textual domain.

In each of the chapters, we begin by introducing the topic and end by discussing related work and summarizing the topic. Additionally, each chapter begins by listing to which papers it refers. More concretely, we provide the following.

We are interested in the expressivity of the metalanguage, i.e., what building blocks are available to language designers. We give a brief overview of the current state-of-the-art software modeling frameworks in Chapter 2 by analyzing the Graph eXchange Language [208], the Meta Object Facility 1.4 [128], Eclipse ECORE [28] and the Kernel Meta Meta Model [84].

Based on this experience, we describe our own modeling framework called the Simple Metamodel Description Language (SMD) using a set-theoretic formalization in Chapter 3. We accomplish this task with successive versions of a modeling framework and note the constraints over the static structure of metamodels and models of each version. We especially support subset and union properties, a novel way to specify relationships.

In Chapter 4, we extend this theoretical framework by adding operations. We describe pre- and postconditions and implementations for the basic operations on models: creating and deleting elements, and inserting an element into or removing an element from a slot. We note that these operations preserve subsetting and bidirectionality, but multiplicity constraints do not hold.

Although the theoretical framework from Chapters 3 and 4 is interesting, we are also concerned about practicality. We have strived to validate our work by implementing our ideas and solutions in the context of a working tool called Coral. It is open source and licensed under GNU GPL version 2 [59]. We describe the core part of it in Chapter 5 by introducing an implementation of the theoretical framework from Chapter 3.

One of the most important operations on a model is persistence, which is easiest to accomplish by being able to serialize the model to files in the filesystem. We assess the suitability of the XMI file format standard [129, 132, 137] with respect to several model interchange scenarios in Chapter 6.

As stated previously, if we want modeling to be used in a professional software engineering context, we must add similar capabilities to modeling frameworks as currently exist in textual programming languages and integrated development environments. An important part of that is software configuration management (SCM). There are several issues with SCM, of which version control is one. In Chapter 7, we show algorithms for calculating the difference between two models, how to apply such a difference to a model producing the other, calculating the inverse of a difference, and detecting conflicts between two developers simultaneously modifying two models. We also show a proof of concept implementation of how a relational SQL database can be used to store models for a centralized version control server.

Finally, we show a solution to the problem of maintaining both an abstract and a concrete syntax in Chapter 8. We describe a domain-specific weaving metamodel between abstract models and Diagram Interchange [136] models. It can be used to create new diagrams from existing abstract models, and to synchronize changes which have occurred in the abstract model to the obsolete diagrams, bringing them up-to-date. The work is interesting because OMG has recently issued a Model View to Request for Proposals [140] that addresses this exact same issue. We have used our idea for managing the diagrams in our graphical user interface.

Chapter 9 summarizes the thesis. Appendix A describes the mathematical notation used in this thesis, whereas Appendix B is a summary of Chapter 3 and describes the metamodeling language used in Coral.

The main thesis is that the current modeling standards by the Object Management Group are lacking if we wish to provide the same level of quality from the software development infrastructure as we have become accustomed to in the textual programming language domain.

1.5 List of Published Papers

Below is a list of papers published relevant to this thesis. We note that in the European communities of Mathematics and Computer Science, no distinction is usually made between the first author and the other authors. Authors are thus listed alphabetically.

I. Marcus Alanen, Torbjörn Lundkvist, and Ivan Porres. Comparison of Modeling Frameworks for Software Engineering. *Nordic Journal of Computing*, 12(4):321–342, 2005.

II. Marcus Alanen, Torbjörn Lundkvist, and Ivan Porres. A Mapping Language from Models to DI Diagrams. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, volume 4199 of *Lecture Notes in Computer Science*, pages 454–468. Springer Berlin / Heidelberg, October 2006.

III. Marcus Alanen, Torbjörn Lundkvist, and Ivan Porres. Reconciling Diagrams After Executing Model Transformations. In Hisham M. Haddad et al., editors, *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1267–1272, April 2006. ACM ISBN 1-59593-108-2.

IV. Marcus Alanen and Ivan Porres. Difference and Union of Models. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Spinger-Verlag, October 2003.

V. Marcus Alanen and Ivan Porres. Coral: A Metamodel Kernel for Transformation Engines. In D. H. Akehurst, editor, *Proceedings of the Second European Workshop on Model Driven Architecture (MDA)*, number 17, pages 165–170. University of Kent, September 2004.

VI. Marcus Alanen and Ivan Porres. Model Interchange Using OMG Standards. In Bob Werner, editor, *Proceedings of the 31st Euromicro Conference on Software Engineering and Advanced Applications*, pages 450–458. IEEE Computer Society, August 2005. ISBN 0-7695-2431-1.

VII. Marcus Alanen and Ivan Porres. Version Control of Software Models. In Hongji Yang, editor, *Advances in UML and XML-Based Software Evolution*, chapter III. Idea Group Publishing, April 2005.

VIII. Marcus Alanen and Ivan Porres. Basic Operations Over Models Containing Subset and Union Properties. In Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, editors, *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, volume 4199 of *Lecture Notes in Computer Science*, pages 469–483. Springer Berlin / Heidelberg, October 2006.

IX. Marcus Alanen and Ivan Porres. A Metamodeling Language Supporting Subset and Union Properties. *Springer International Journal on Software and Systems Modeling.* Accepted for publication.

1.6 Validation

We conclude this introduction by giving a brief account of the Coral tool itself, and how Coral has been used in various other projects by other people in our research group and elsewhere.

1.6.1 Coral

The Coral model manager is a library for managing metamodels and models. It is written in C++ and consists of around 23 000 lines, in addition to some Python scripts that help in building the library. It also exposes a SWIG [173] interface for the Python language. The Python interface is used to create a graphical modeling toolkit and several scripts. The development has been test-driven, in that new unittests have been created before or after the code has been written to ensure that some particular functionality exists and works as intended. In our opinion, when errors have been found, we have made a solid effort in reproducing them by creating new test cases and then correcting the program code.

The Coral model manager contains many features that are not described in detail but that we consider relevant in a modeling tool: operations, transaction mechanism with undo/redo, copying, deleting, pasting, transforming, and comparing, to name a few. Several of these features can either be implemented in the model manager or transparently by other third-party scripts.

Coral is not a replacement for any existing modeling tool but a demonstrator of different research ideas. This has ensured that we are not tied to the idiosyncrasies of any particular metamodeling implementation, although creating it has required a significant effort.

1.6.2 Applications

We will describe some applications of Coral in this section. We wish to make it clear that these applications are not an accomplishment of the author of this thesis. They are described here to validate that the Coral model manager can in fact be used as a library for manipulating and maintaining models.

UML Support

The Coral model manager has been used to define several modeling languages. So far, UML is the primary and largest modeling language supported by Coral and will probably always be, since other domain-specific languages tend to be smaller and simpler than UML. There are metamodels for UML versions 1.1, 1.3, 1.4, 1.5 and 2.0. Version 1.4 is the most complete one, as it includes diagrammatic support and we have extensive experience in using it. Version 2.0 is achieved using the UML 2.0 metamodel from the EMF project. However, the redefinitions are lacking since Coral does not support them. Future plans include supporting the concrete syntax of UML 2.0.

It is interesting to note that although Coral does not use the MOF metametamodel, it is compatible with the UML metamodel with respect to the serialization of models in XMI. Coral can import and edit models created with commercial and open source UML tools, such as Poseidon, Rational Rose, ArgoUML and Umbrello. Unfortunately, at the moment only some old versions of Poseidon are compatible with the DI [136] diagram interchange language, and so their diagrams are the only ones compatible with Coral. DI will be discussed in Chapter 8. Currently, Coral supports class, use case, state, collaboration and (Poseidon-specific) deployment diagrams. The conclusion is that as long as the metametamodel is sufficiently expressive compared with another metametamodel, constructs from it can be emulated at the model layer.

There is also some basic code generation scripts that can create the class structure as Java or Python files and an initialization routine that creates objects and calls their methods according to one collaboration diagram.

Toni Jussila et al. have created a round-trip transformation tool from UML state machines to Promela code [85].

MICAS

MICAS [103, 102] is an architecture for describing component-based embedded systems. Each component consists of a microcontroller which controls traffic between smaller computational units. In Figure 1.1, we can see a microcontroller which can send commands to a sound recorder and video encoder. These are connected by one high-speed data bus together, and to other components via a special socket construct.

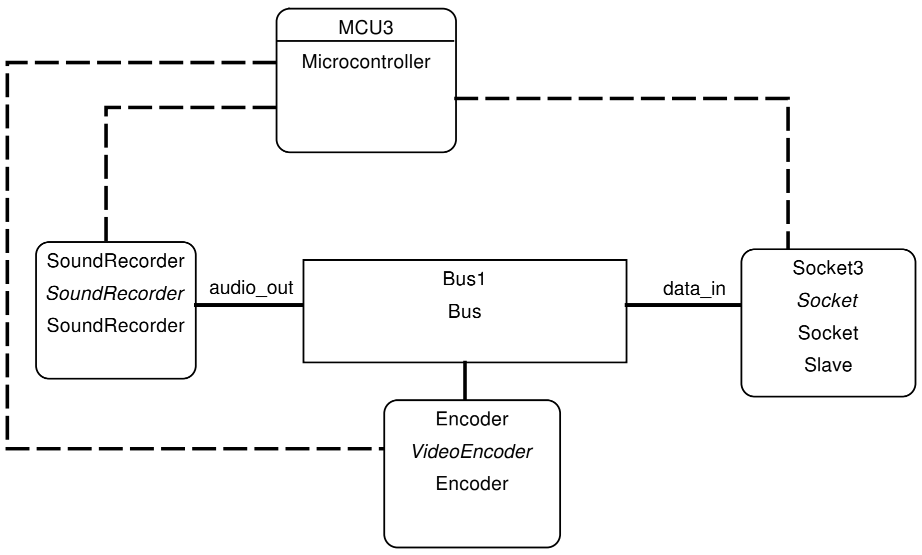


Figure 1.1: Example of a MICAS Model

This kind of a conceptual design is transformed using a double-pushout graph transformation [160] to a detailed design (not shown), from which SystemC [145] code can be generated.

SOCOS

SOCOS [12, 13] is a tool for describing software using invariant-based diagrams. An example SOCOS model can be seen in Figure 1.2. A set of program states is represented as a colored region, and guarded statements can be executed to change state. Due to the use of invariants and guarded commands, it is possible to generate proof obligations that can be proved by tools such as Simplify [49] or PVS [144].

Miscellaneous

There are also several smaller tools built into the Coral modeler.

There is a metamodel for describing additional constraints on models. These can then be evaluated on some specific model and a list of violations is shown. An interesting and useful feature is constraint checking in the background. After a change has been done to a model, a heuristic is used to make an educated guess of which constraints to reevaluate on which model elements. These are then run while the user is allowed to perform additional modifications to the model.

A metamodel and an algorithm for subgraph isomorphism matching with negative acknowledgment conditions has been implemented by Tomas Lillqvist [104]. Effectively it can be used as a query language that asks whether a certain pattern

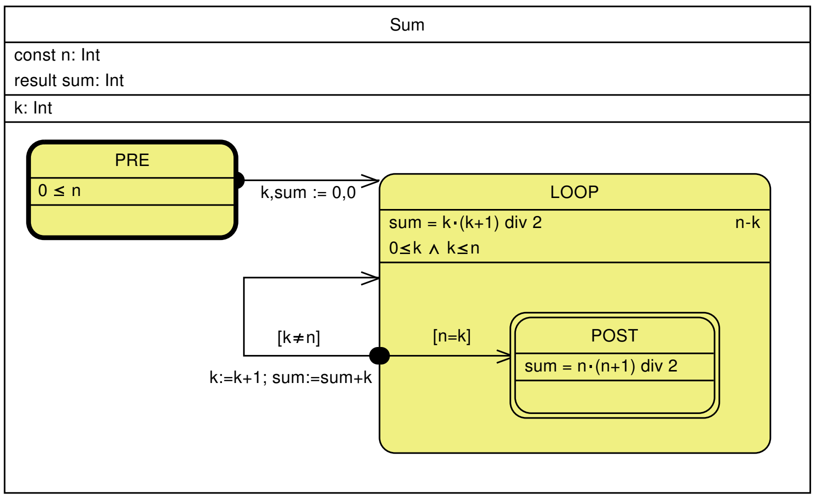


Figure 1.2: Example of a SOCOS Model

exists in a model. Transitive closures were later added by Johan Lindqvist, Torbjörn Lundkvist and Ivan Porres [107].

Using the model query language described above, it is possible to implement a transformation engine using the double-pushout approach, an example of which can be seen in Figure 1.3. The figure specifies that any pattern matching the left-hand side from a source model will be transformed into the right-hand side.

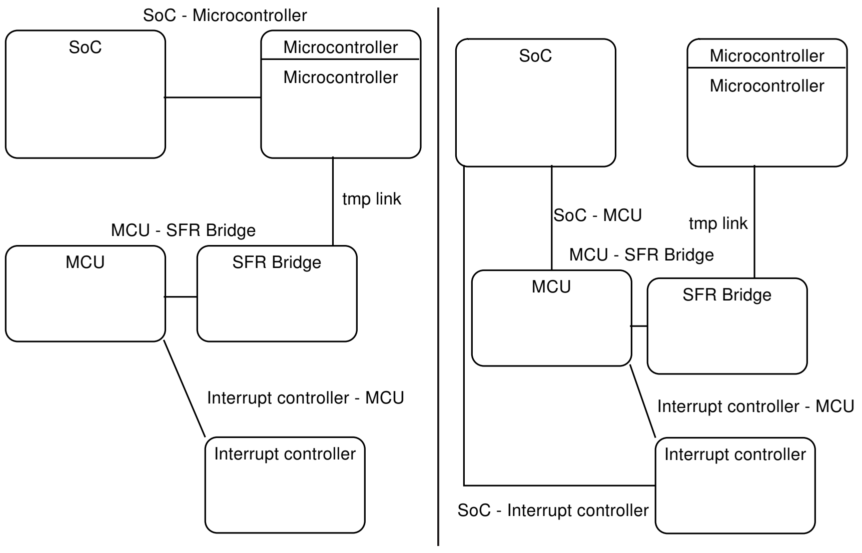


Figure 1.3: Example of a Transformation with MICAS Models

Chapter 2

Software Modeling Frameworks

2.1 Introduction

In this chapter we compare four modeling frameworks which have been created to model and interchange data about software and software development artifacts. Our main focus is in discussing the rationale and expressivity of modeling frameworks, although various practical considerations are also mentioned. This chapter can be considered as a study of the state of the art in the field of modeling frameworks.

We consider a modeling framework to be a software component that implements a metamodeling language, with which several metamodels and models based on those metamodels can be created. In other words, users of the framework are not tied to produce abstract models conforming to some specific metamodel such as UML, but can create and use multiple metamodels. This is contrary to many tools which only support one specific metamodel, usually some specific version of UML. We do not consider these tools to be modeling frameworks.

The frameworks which we will compare are the Graph eXchange Language (GXL) [208, 206], the Meta Object Facility 1.4 (MOF) [128], ECORE from the Eclipse Modeling Framework (EMF) [28] and finally the Kernel Meta Meta Model (KM3) [84] from INRIA. All these frameworks enable us to create different kinds of metamodels or graph grammars. We will primarily analyze the graph structures that can be created with them and how they are suitable for software engineering.

GXL is a standard exchange format for graphs by Richard C. Holt, Andy Schürr, Susan Elliott Sim, Andreas Winter et al. [207] with the backing of several research communities. GXL is used to describe arbitrary graphs, but additionally it can be used to define GXL schemas which constrain the graphs so that only specific kinds of graphs can be built.

MOF from the Object Management Group (OMG) [123] is a framework for describing metamodels. The metamodels can be used to create models. Metamodels can also be seen as models, with MOF as their metamodel. Serialization is done using the XML [188] Metadata Interchange (XMI) [129, 132] format, which is an XML application. In this chapter, we concentrate on the older and significantly simpler MOF version 1.4 instead of the relatively new and complex version 2.0 [139], although we are aware that version 2.0 includes some interesting enhancements; they will be discussed in Chapter 3.

ECORE can be seen as a practical implementation of MOF: EMF is the modeling framework used by the Eclipse platform, and uses ECORE as its underlying metametamodel. As the platform originally started as a Java development platform, it is natural that ECORE has concentrated on making program development and especially Java program development easier with modeling technology. ECORE is also serialized using XMI. It must be noted that ECORE is not a standard endorsed by any group or organization, contrary to GXL and MOF.

KM3 by Jean Bézivin et al. at INRIA can be seen as a lightweight ECORE. It is in fact implemented in the Eclipse framework, so there are several similarities. It is also serialized using XMI, but there is an additional human-readable and -editable textual syntax for defining new languages. KM3 is not a standard.

We have obtained the results presented here using two different approaches. First, we have studied the documents that describe the GXL and MOF standards and any documents available on ECORE and KM3. Second, we have implemented different modeling tools (SMW [11, 152], and Coral, which will be discussed in Chapter 5) that include, in one way or another, support for XMI, MOF, GXL and ECORE. Unfortunately time constraints have meant that we do not have practical experience with KM3.

The four frameworks studied can be considered as four different approaches to define graph grammars. The main differences between each approach investigated is the structure and constraints of the graphs representing our artifacts. Another difference will be the mechanisms to serialize these graphs into an XML document. The terminology used in each approach also varies considerably.

This chapter is based on Publication I. We proceed as follows. The next section explains more closely in what aspects of modeling frameworks we are interested. Sections 2.3, 2.4, 2.5 and 2.6 describe GXL, MOF, EMF and KM3, respectively. We summarize our findings in Section 2.7. We finally discuss related work in Section 2.8 and conclude in Section 2.9.

2.2 Comparing Modeling Frameworks

In this section we explain what aspects of modeling frameworks are interesting for our comparison. From a theoretical point of view we might consider which structures and behaviors it is possible to create with a modeling framework. From a practical point of view we might consider which facilities are available and standardized for using the modeling framework. Since the modeling of information in the context of software engineering is the primary scope of this thesis, we hesitate to include any discussion and comparison of behavior. Although behavior, such as defining complex operations or execution semantics on models, is crucial for the success of modeling, the comparing of which structures can be built from the modeling frameworks is even more important since that lays the foundation on which we can model behavior. We need to find the basic building blocks necessary to describe various information artifacts and make sure that they can be integrated together as seamlessly as possible. Therefore our primary basis of comparison is structure, whereas practical aspects are our secondary basis; behavior is not relevant in the scope of our study.

2.2.1 Structure

Modeling frameworks propose *models* consisting of interconnected *elements* as our primary means to model information. Elements are typed, and for practical purposes carry a *globally unique identifier* but contain no more information than that. Thus, almost all expressive power of models is in the *relations* between elements. An n-ary relation consists of *n association ends*, where each association end links to an element, and the association ends together link the elements together. The association ends are, depending on their type, given certain characteristics such as ownership and ordering.

The fundamental question is which characteristics are necessary or useful for creating new languages. Also, one quickly realizes that not all combinations of characteristics in a modeling framework are valid. For example, ownership between elements is usually required to be acyclic, but often association ends have a boolean flag denoting ownership; this scheme makes it possible—but not valid—to create a relation where both association ends denote ownership. To overcome this problem, wellformedness rules (WFR) are often used to specify clearly which languages and models are valid and which are not. However, the fewer WFRs there are while still retaining sound languages and models, the easier it is to check metamodels for validity. We call two characteristics *orthogonal* if using either one does not affect the use of the other one, i.e., they are independent. A modeling framework can be considered better the more orthogonal its characteristics are, while still providing the necessary expressive power.

Although we could thus measure the orthogonality of the characteristics of a framework, we cannot easily measure the utility or necessity of them. This can only be acquired by analyzing in what contexts the characteristics can be used.

2.2.2 Practical Aspects

We will also informally cover some practical aspects of the modeling frameworks in question. It is not immediately clear what kinds of features should be present in the context of software engineering, and the features we have chosen is based on the experience we have drawn from using and implementing modeling frameworks.

Our current data retainment and exchange infrastructure is heavily based on files and streams of data. A de facto file is one stream of octets (eight bits), although there are filesystems which operate on files with multiple streams and with a different bytesize than eight bits. If models are to be stored inside files, and we may potentially have several files constituting all the documents, there must be some way to *identify elements* inside files for interdocument linking of elements. Furthermore it must be possible to *identify the types* that various elements correspond to.

Visualization of models is an important issue. The Unified Modeling Language was a hallmark in software engineering for the simple fact that it tried to unify the different visualizations of common diagrams such as class diagrams or state machines. Now that we want to create several interlinked models using multiple languages, visualization is perhaps even more important, and a standardized viable solution for describing which diagrams are legal and how to construct them is necessary.

Model transformations are one of the core ideas in MDA, and can be accomplished in several ways. Naturally, we can have an Application Programming Interface (API) as some kind of library that can be interfaced by various programming languages. But if modeling really is a better approach to software engineering than textual programming languages, we might expect transformations to be specified as models as well. Research has shown that there are several kinds of model transformation languages describing the transformation specifications as models. These can be interpreted by using the standard transformation language of the modeling framework, if one exists. We assess the maturity of each framework-specific standard transformation language.

Realizing that not all artifacts are yet described as models, we might, from a pragmatic point of view, wish to extend the modeling framework somehow to take other nonmodel artifacts into account. *Extensibility* can mean several things, such as tagging of extra information to elements in the modeling framework and in the serialization format. We explore what facilities for extensibility the different modeling frameworks provide.

Finally, an idea of how the modeling framework or associated tools are licensed and supported is of interest if we wish to consider their use in modeling.

2.3 Graph Exchange Language

GXL 1.0 [75] was created by merging properties from several graph formats such as the GRAph eXchange format (GraX) [51] and the graph format of the PROGRES graph rewriting system [164]. The goal of GXL is to be a universal description format for graphs. It is partitioned into two separate documents, the graph model which can describe any graph, and the metaschema which is used to describe the type system used in a graph.

2.3.1 Structure

The GXL graph model arrangement can be seen in Figure 2.1. A GXL Node supports directly the properties defined previously for nodes in a graph. GXL supports ownership hierarchies by inclusion of other subgraphs via GraphElement.contains, which contain other nodes and edges. GXL supports binary edges as a special Edge element, and hyperedges with a Relation element. All elements can be attributed via Attribute elements, and all elements support an optional type via the hasType connection to the Type element. The type is defined in a GXL *schema*.

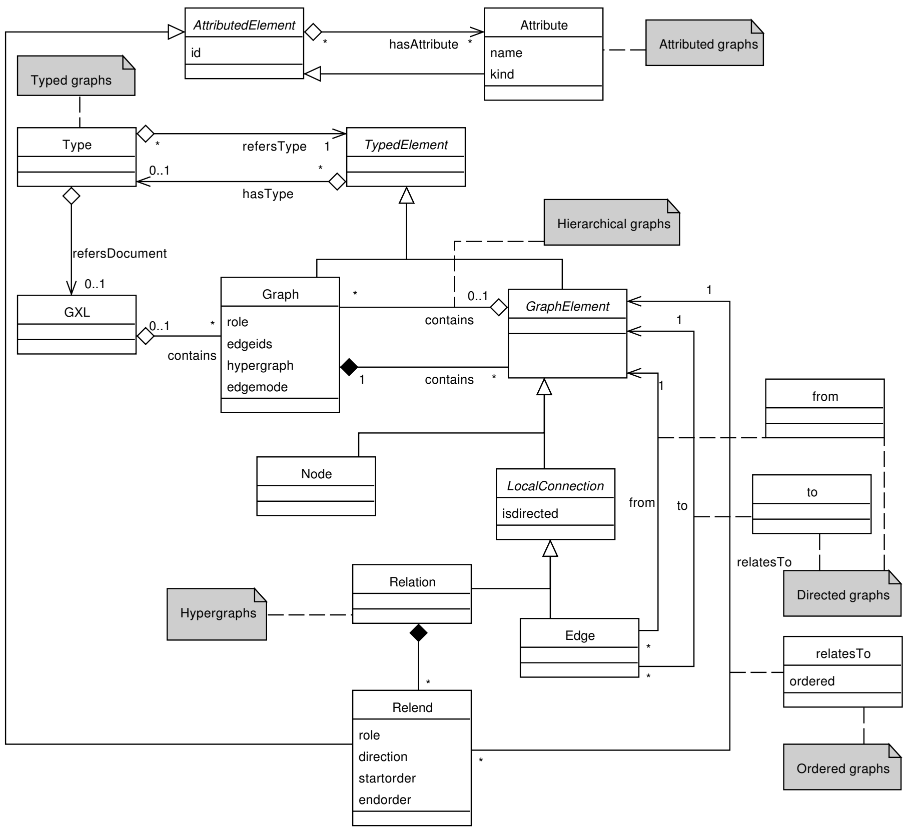


Figure 2.1: The GXL Graph Model

The GXL graph model establishes few restrictions on what graphs can be created. An example of this can be seen in its support for edges linking edges or nodes, not only nodes. It is therefore a very general solution. This can also be seen in its history. A small drawback of such a general solution is that in order to establish more constrained graphs it must be possible to define these constraints in some language. These languages are called schemas in GXL. If we also want tools to support generic manipulation of information all of these languages must adhere to some common metalanguage, which is called the GXL *metaschema*. The beauty of GXL is that it describes the schemas and the metaschema as GXL documents. This means that one serialization format is sufficient. A GXL information processing tool only needs the GXL DTD to load and save GXL graphs, schemas and the metaschema. This arrangement can be seen in Figure 2.2. Elements in the schemas can be used as types. However, this also means that there is an extra layer of indirection and understanding that tools must perceive, not just the XML document itself. Even though the tool can load arbitrary GXL documents, it must understand the relationship between metaschemas, schemas and vanilla GXL graphs. Failure to accomplish this renders graph modifications infeasible.

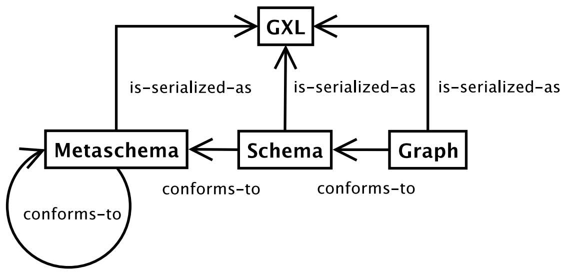


Figure 2.2: Overview of GXL and its Artifacts

The graph part of the metaschema is depicted in Figure 2.3. Instances of it are the GXL schemas, which serve as the structural constraints on actual GXL graphs. Conceptually, the metaschema should be compared with MOF and ECORE as they all define the restrictions on which languages can be built.

The primary artifacts in GXL are the various subclasses of the GraphElementClass and their interconnections. An inheritance hierarchy of the GraphElementClass metaelement with *multiple inheritance* can be created with the GraphElementClass.isA relation. Here, we can note an interesting detail: according to the GXL FAQ, "in a valid GXL schema, an EgdeClass cannot inherit from a NodeClass (and vice versa, the same applies to RelationClass)", i.e., the isA relation is really covariantly specialized in RelationClass, NodeClass and EdgeClass, even though the figure does not convey this information. We do not mean to indicate that this is a relevant flaw in GXL, but rather that it is very difficult to formalize metametamodels/metaschemas without relying on implicit assumptions from readers or descriptions in natural language. Even though we could add covariant specialization of relations to the metaschema, there would still be other cases where arbitrary constraints are necessary. This emphasizes the need for a formal language for arbitrary constraints and is worth remembering regardless of the framework used. Covariance is explained more thoroughly in Chapter 3.

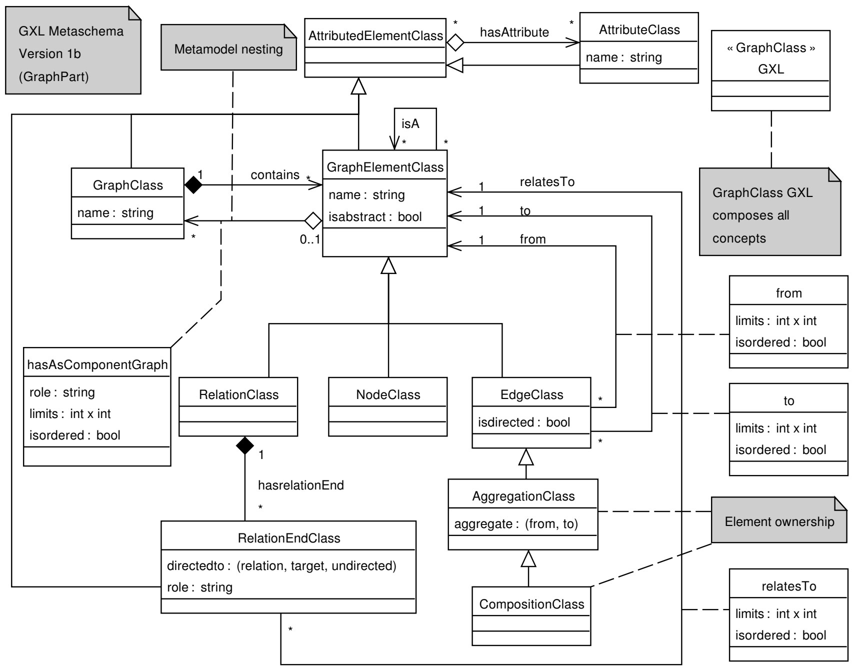


Figure 2.3: Part of the GXL Metaschema

A GraphElementClass can also be declared *abstract* with the isabstract property. Subgraphs can be created with the hasAsComponentGraph property. These are identified by a name, and have a lower and upper *multiplicity constraint* which tell how many subgraphs of the given name must exist. The order of the subgraphs can also be specified as important with the isordered property.

Edges can be of three types: compositions, aggregations (shared compositions) and "plain associations". Edges also have lower and upper multiplicity constraints and can be directed or undirected; both the source and target collection can be ordered or unordered, meaning that order is considered important and must be preserved by any input/output routines and must be taken into account by query or transformation algorithms. The edges represent an ownership hierarchy at the graph level, but a hierarchy of graphs can also be created instantiating GraphClass in the schema level and Graph on the graph level.

However, the metaschema cannot describe more complicated constraints. This has the benefit that representing and validating graphs remains fairly simple, although practical considerations might dictate a need for arbitrary constraints. For example, in the definition of UML, additional constraints have been used.

It is unclear why GXL has a separate concept for hierarchical graphs. The relation between a graph and the subgraphs it contains could be a Relation or Edge element with a special type. We noticed that there are some small discrepancies between the pictures presented on the website and the DTD itself, e.g., Attribute in Figure 2.1 is called attr in the DTD. This is, again, a minor flaw, but it emphasizes the need for clear documentation of the language.

2.3.2 Practical Aspects

Element Identification

For practical purposes of serialization, elements in a graph may possess an *identity*. In GXL this identity is described with the AttributedElement.id property and is a string unique to the XML document. This is correctly marked as an XML identifier in the GXL DTD, although in the long run the xml:id recommendation [198] standardized by the World Wide Web Consortium might be adopted instead. The filename (or URI) of the document and the identifier serve as a globally unique identifier. However, this limits the mobility of GXL elements to their document, as there might be a need to change the identifier of an element to avoid identifier collision if the element is moved to another document. A more opaque globally unique identifier is necessary.

Schema Identification

In order for tools to understand a GXL graph more thoroughly, it is important to be able to identify what schema is being used, i.e., what types are available to GXL elements. The schemas are usually defined in a separate GXL file and shared among all the GXL graphs of that type. Linking to a schema is done using the native facilities of XML, i.e., XLinks [189] and XPaths [197]. They make it possible to uniquely identify a document, for example on the WWW. This allows a GXL graph to explicitly reference a specific schema, and additionally it allows tools to download the schema from the location specified by the URI. This means that generic tools can be extended on-the-fly with new schemas, as long as they use the semantics of the same metaschema.

Visual Representation

GXL itself does not define a mechanism for presenting a graph visually on-screen, although this can be remedied in two ways. The simple solution is to define attributes that describe the position, size, form et cetera of a GXL GraphElement. The more complicated solution is to define a whole new schema for describing the visual representation, thereby decoupling the abstract syntax (the graph) from the concrete syntax (the presentation). This idea is similar to what is already being done by the OMG in the form of the Diagram Interchange (DI) [136] standard and has the benefit that the representation can be split into several possibly different diagrams, each showing a subset of the abstract graph. The drawback is the necessity of synchronization of changes in either the abstract or concrete syntax to the other.

Transformation

GXL graphs can be transformed with the Graph Transformation eXchange Language (GTXL) [174], although at this moment a revision of GTXL seems to be under way by Leen Lambers [98]. Unfortunately, we do not have experience with GTXL and cannot comment on its viability. On the other hand, graph transformations have been extensively researched and we believe it should be possible to adapt any transformation technology using graphs from one underlying schema to another with few problems.

Extensibility

GXL allows arbitrary embedding of extra XML information into any GXL element via user-defined extensions to the normative DTD at predefined entry points, e.g., a Node can have children XML elements as defined by the *node-extension* XML element. This has the disadvantage that tools must be ready to process the non-GXL information somehow, either by simply ignoring (and retaining) it or removing it.

Additionally GXL allows adding URIs as values to Attributes (not shown in Figure 2.1), which should be considered a viable way to link to some external resource, e.g., a Microsoft Word document.

Current Support and Licensing

Current support of GXL seems to be very good. There are several researchers and companies listed as supporters or contributors on the GXL website [207]. Several tools include export or import capabilities of GXL, such as the round-trip UML software engineering tool Fujaba [121] or the graph transformation toolset GROOVE [155].

Overall, there is activity in the GXL community, and a new version 1.1 is being planned. GXL is licensed without any fees or restrictions.

2.4 Meta Object Facility

MOF is one of the current flagships of the OMG industry consortium. It is used to define UML, which is one of the most well-known ways to describe software artifacts at the moment. MOF takes a slightly different approach to modeling than GXL. In MOF, the developer must first define a language (a metamodel) that can be used in creating the actual model (i.e., the actual information). One of the possible metamodels that can be defined in MOF is MOF itself, thereby closing the metacircularity.

2.4.1 Structure

The relevant part of the MOF metametamodel can be seen in Figure 2.4. We have restricted ourselves to the parts that mainly describe the structure of metamodels. As a simple starting point for comparing MOF models to a graph, we may say that the nodes in a graph are mainly Class metaelements, and that edges are represented by Association metaelements.

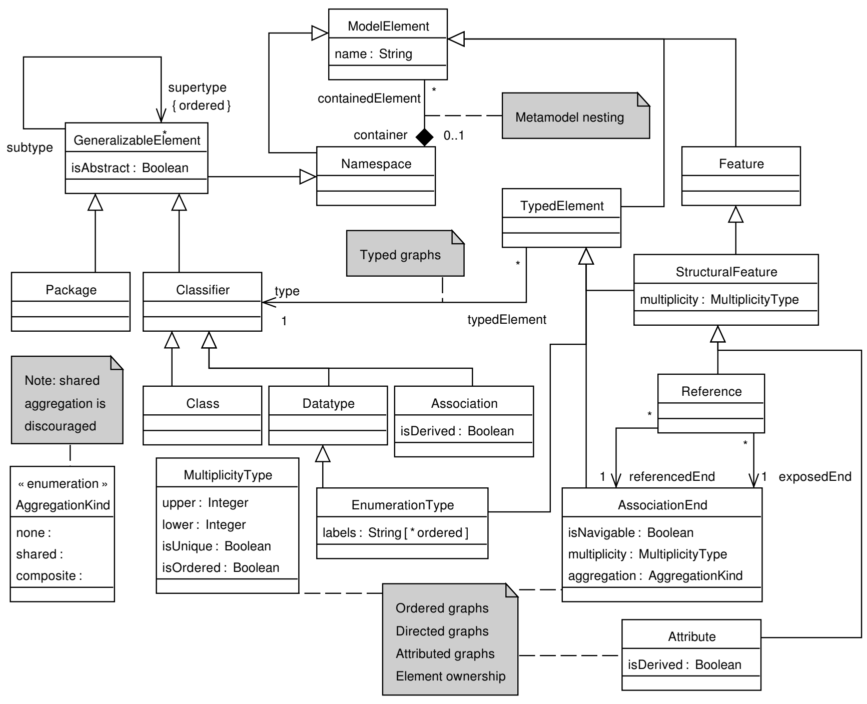


Figure 2.4: Part of the MOF Metametamodel

It can be understood that a metaelement can establish ownership by two means. One, a metaelement can have Attribute metaelements via the containedElement property. These parts have an obligatory type via TypedElement.type and a MultiplicityType that states the minimum and maximum number of, as well as possible ordering and uniqueness of, elements. Two, a metamodel can have Association metaelements which each contain exactly two AssociationEnds. These, almost similar to the Attribute, establish a link between two metaelements, but each AssociationEnd can be explicitly set navigable (which supports directed graphs) and can support three different kinds of ownership, along with the usual support of MultiplicityType. However, an Association can be and usually is bidirectional, meaning that if a source element is connected to some target element via their slots, that target element is also connected to the source element.

The three different kinds of ownership are the same as in GXL: composite, aggregate and plain. However, using aggregation (shared composition) is discouraged and it has been removed in MOF 2.0. The reason might be that if one ignores the plain associations, the resulting ownership structure in the form of composite connections forms a tree, which has been found to be a very useful structure and which directly maps to XMI. Aggregation, resulting in an ownership structure of directed acyclic graphs, is not as common, although it is certainly useful.

To summarize, an ownership hierarchy of metaelements is established via the Namespace.containedElement property, and as in GXL, it can be used to split metamodels into packages. An ownership hierarchy of elements is established by Associations with one AssociationEnd marked as composite.

Reference metaelements are owned by Classes and are used to track which AssociationEnds are connected to them. This seems a bit redundant, as the Classes could reference some of the AssociationEnds directly, and in fact MOF 2.0 has done exactly this.

Since MOF employs a two-step process whereby the user first creates a metamodel, which then allows them to create models, the resulting usage and serialization of those models in XMI is different from GXL. This is depicted in Figure 2.5 and shows that tools require metamodel-specific XMI importers/exporters. One serialization format is XMI(MOF) for MOF metamodels, and every such metamodel defines its own serialization format. In other words, to be able to load a UML 1.4 model from an XMI document, the tools must know how to acquire the UML 1.4 metamodel definition first, otherwise they are unable to load the model correctly. This is a big contrast with GXL-compliant tools, which may be able to load the graph with an unknown schema by ignoring the type system.

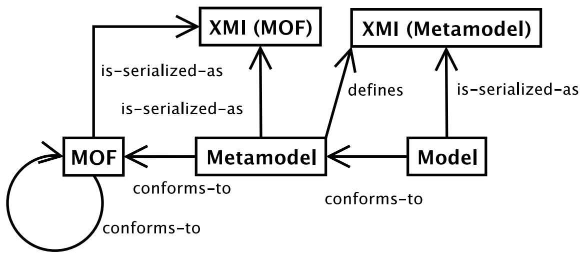


Figure 2.5: Overview of MOF and its Artifacts

Constraint support in MOF can be assessed as excellent due to the Object Constraint Language (OCL) [127, 131], an addition to MOF. OCL enables a metamodel developer to add arbitrary constraints to the users' models, thus enforcing very sophisticated constraints between elements. A tool can then check these constraints and report nonwellformedness. OCL is used extensively in defining constraints for metamodels.

Curiously, MOF is the only framework where the collection of superclasses is ordered. This aids in determining a monotonic linearization of superclasses [50], which is useful at least in method resolution in object-oriented programming languages.

2.4.2 Practical Aspects

Element Identification

Element identification in MOF is handled by XMI with its xmi.id and xmi.uuid XML attributes. They have been properly defined in XMI and the UUID specification [31] and we will discuss this extensively in Chapter 6. To summarize, elements can be identified in an XML document with both xmi.id and xmi.uuid, enabling rigid interfile element identification. On the other hand, current support for XMI import and export in tools is sometimes lacking [109]. We will revisit XMI more thoroughly in Chapter 6.

Language Identification

Similarly to GXL, it is important to detect which metamodel is being used in a model. XMI allows using several metamodels in the same document with XML namespace [185] declaration strings describing which languages are being used where. This usage is nicely aligned with advances in XML by the World Wide Web Consortium. A major issue is that "there is no requirement or expectation by the XML Namespace specification that the logical URI be resolved or dereferenced during processing of XML documents" (page 1-16 of [132]). This implies that a tool cannot in general be able to even load a model without knowing the metamodel in advance, because it cannot rely on acquiring the metamodel from the URI. For example, the UML 1.4 namespace is *http://schema.omg.org/spec/UML/1.4*, but there is no document at that address.

Visual Representation

MOF does not define a visual representation for models. The basic premise is that there is a strong separation of abstract models containing the semantic data and the diagram which merely display the artifacts on-screen. Thus, the Diagram Interchange (DI) standard [136] has been developed. DI has been successfully used in the Poseidon tool [61] and our Coral tool. This will be discussed in Chapter 8 and the conclusion is that DI is a viable if somewhat complicated standard that can be used to represent diagram models.

The standard thus far missing is a way to describe the mapping between abstract and concrete models. We will revisit this issue in Chapter 8.

Transformation

It is conceivable that several different transformation technologies are used for model transformations, although the Query-View-Transform (QVT) [133] is the primary standard pushed by the OMG to enable the transformation of MOF-based models. As the standard itself is relatively new, we feel it is too early to discuss its benefits or drawbacks.

Other transformation technologies have been described by several authors, for example UMLX [205], YATL [146], MT [177], MOLA [86] and VIATRA [184].

Extensibility

MOF metamodels and models cannot as such be extended, but both metaelements and elements can be tagged with arbitrary information using the XMI.Extension XML node. A whole XMI file can be tagged with the XMI.Extensions XML node.

Current Support and Licensing

Current support for MOF 1.4 is low. The metametamodel itself has some nonintuitive quirks and is quite big and complex, which presumably has lead the Eclipse team to create EMF and ECORE. Additionally, MOF 2.0 has become even more complex than its predecessor. The benefits are not clear since MOF lacks experience reports detailing which parts of the standard works and which do not.

Ironically, the low support for MOF will perhaps not matter, as the XMI serialization is not dependent on MOF per se, but on the metamodels created in MOF. That is, it is possible to create a tool that is not based on MOF but is still compatible with the UML 1.4 XMI serialization format.

MOF is released under a royalty-free license.

2.5 ECORE

EMF is the modeling component of the Eclipse project. The heart of EMF is the ECORE metametamodel, which is in many ways similar to MOF, with the same ideas about metamodels and models. It uses XMI as its serialization format as well, so several comments regarding MOF are also valid for ECORE.

2.5.1 Structure

A part of the ECORE metametamodel can be seen in Figure 2.6. Comparing with the general graph concept, we can say that nodes are represented by the EClass and EAttribute classes, and edges by the EReference class. Containment is indicated by the EReference.containment boolean value, so ECORE does not support shared composition, only composition and associations. EAttribute elements are implicitly contained by an EClass element.

ECORE uses a lot of directed associations, probably because of its roots as the underlying information framework in the context of a Java programming environment. There is also a lot of derived and thus redundant information available, e.g., eAllSupertypes is a collection of the flattened superclass hierarchy. There is also a lot of similarities with MOF, such as EPackages being used for metamodel nesting.

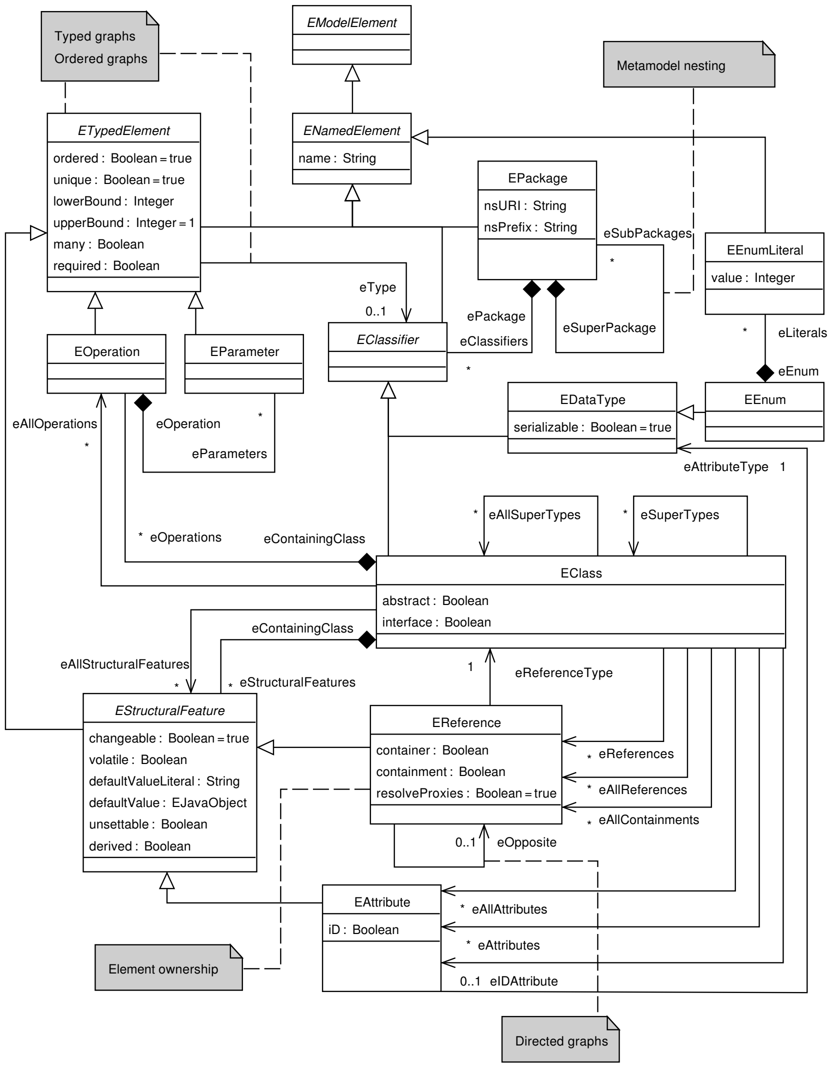


Figure 2.6: Part of the ECORE Metametamodel

ECORE also has two different ways to present relationships between types. Relationships between EClass elements must be described using EReference elements, whereas relationships between EDatatype elements must be described using EAttribute elements. The benefits are not clear, as EReference elements (without an eOpposite) to EDatatypes could be used instead. To us, an EAttribute should be equivalent to a unidirectional composite EReference to a primitive type. Neither is there a concept for describing the relation, akin to the MOF Association class; instead, two EReference elements simply link to each other.

Similar to the two previous frameworks, multiplicities, ordering, directedness and bagness are supported. A concise summary of ECORE is that it is, loosely speaking, a practical implementation of MOF. The metametamodel is more clear and developed by demand rather than the admittedly more complicated MOF standard. Due to the similarities, technologies developed for MOF should be very easy to transform into ECORE technologies. This is also evident from the choice of using XMI as the serialization format of ECORE. The artifacts needed and produced by ECORE are identical with MOF, as can be seen by Figure 2.7. This has the benefit that UML models from ECORE are indistinguishable from UML models from MOF, but the drawback that the UML language definition is serialized differently.

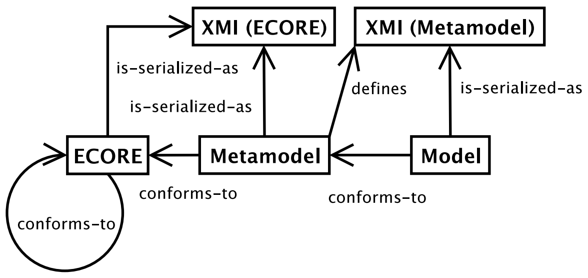


Figure 2.7: Overview of ECORE and its Artifacts

2.5.2 Practical Aspects

Element Identification

Element identification is handled by XMI in exactly the same way as in MOF. At the moment, the Eclipse implementation does not use UUIDs at all, and instead uses several concepts from the XLink and XPath standards for inter- and intrafile referencing of elements. Our understanding of future development is that UUIDs will be supported by EMF.

Additionally, ECORE specifies that if an EClass owns an EAttribute which has its iD attribute set to true, an EClass instance of that type can be uniquely identified by the value of that attribute. In our experience, these kinds of identifiers are not required for modeling purposes, since serialization approaches such as XMI reference elements directly via their UUID, xmi.id or xml:id. However, they can provide a bridge from other textual references to model elements and could be considered useful as a migration path from text to models.

Language Identification

Due to the XMI serialization, language identification is handled exactly the same way as by MOF.

Visual Representation

There is no formal visual representation of ECORE metamodels and models. As we understand it, the EMF development team strive toward supporting the DI standard as endorsed by OMG.

Transformation

Due to the rising popularity of the Eclipse platform, there are several transformation technologies for ECORE models. Among them are the Atlas Transformation Language (ATL) [21] and the Model Transformation Framework (MTF) [161]. The latter is a QVT prototype.

Extensibility

In addition to the extensibility supported by XMI, every ECORE element can have additional annotations. An EAnnotation element (not shown in Figure 2.6) can contain arbitrary ECORE elements.

Current Support and Licensing

EMF itself is released under a royalty-free license. Since the primary implementation of ECORE is EMF, and ECORE is maintained and updated by the EMF developers we assume ECORE is also released under the same license.

We have implemented the ECORE metametamodel in our Coral tool as a separate metamodel. This has enabled us to successfully load ECORE metamodels and convert them to our internal metamodel format.

Although there are few ECORE implementations, the Eclipse platform has such a big momentum and corporate backup by IBM that it is likely to be one of if not the most successful metametamodel in the foreseeable future. It has also lead to better XMI compatibility between tools [109], as many of them are based on the EMF platform.

A possible issue with ECORE is that it is still evolving along with the EMF effort. This has its benefits and drawbacks, as new ideas can be incorporated into ECORE, but it is at the same time a moving target and there is a risk of incompatible implementations.

2.6 Kernel Meta Meta Model

The Kernel Meta Meta Model (KM3) is developed by Jean Bézivin et al. [84] at INRIA, a French national institution researching computer science, among other fields. The primary implementation of KM3 is implemented on top of EMF on the Eclipse platform. KM3 also has a Prolog definition.

We note that there is a repository, called the Atlantic Zoo, of over 240 KM3 metamodels available [175].

2.6.1 Structure

Figure 2.8 shows the whole KM3 metamodeling language. There is an immediate resemblance between KM3 and ECORE. Comparing with ECORE, we see that KM3 does not support operations. However, it has support for subsets and derived unions. EStructuralFeature from ECORE is more expressive than the equivalent construct, StructuralFeature, from KM3. Primarily, KM3 does not support default values, default value literals, and the "derived" property. The other differences here are due to the fact that KM3 does not support operations. Also, there is no definition for a URI namespace in Metamodel, which is surprising.

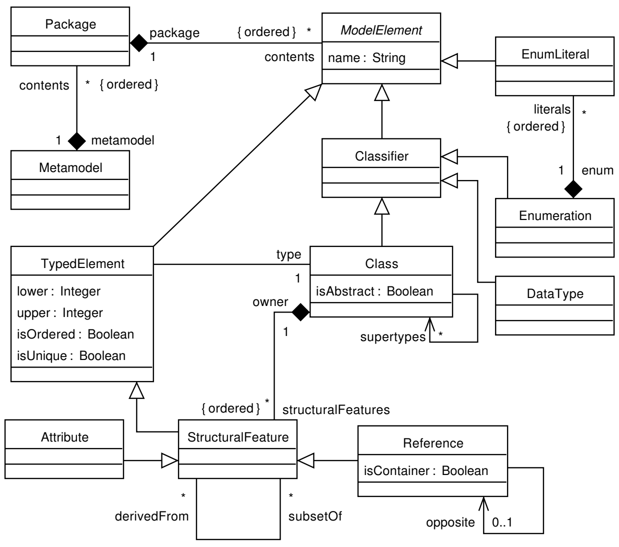


Figure 2.8: The KM3 Metamodel

Also KM3 supports the same aggregations as ECORE, and has two different constructs for relationships, Attribute and Reference. The redundant derived slots of ECORE do not exist in KM3. It is interesting to note that most one-to-many-to-one relationships are marked as ordered, even though keeping the order of, for example, ModelElements in a Package is not important. A benefit is that it is possible to retain the order for aesthetic purposes, as users can expect elements to retain their relative order across a load-save cycles. The ordering criteria might be related to the additional textual syntax definition.

The overview of KM3 and its artifacts as shown in Figure 2.9 is identical to its corresponding figures in ECORE and MOF, because of the similarity between them and KM3.

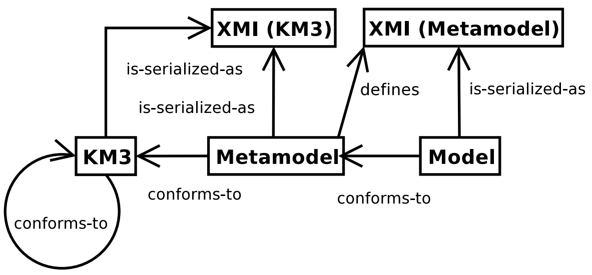


Figure 2.9: Overview of KM3 and its Artifacts

2.6.2 Practical Aspects

Element Identification

Technically KM3 supports the same mechanisms to identify elements as MOF and ECORE, since it uses XMI. However, the syntax of the human-readable KM3 language examples in the Atlantic Zoo do not use UUIDs. Without knowing further, we assume that names should be used for identification and that UUID is not as much used as it could.

Language Identification

Due to the XMI serialization, language identification is handled exactly the same way as by MOF and ECORE. We note that KM3 does not have support for saving the URI namespace of a language. This bases language identification on only the name of the language.

Visual Representation

There is no formal visual representation of KM3 metamodels and models. It can be noted that there is a textual representation of KM3 metamodels.

Transformation

The primary KM3 implementation is implemented on top of EMF and can thus use whatever transformation languages are available via it. Especially, the Atlas Transformation Language (ATL) [21] has been created by the people behind KM3.

Extensibility

There is no extensibility support in KM3, except for the extension support provided by XMI.

Current Support and Licensing

The research group at INRIA is participating very actively in the modeling community. However, there is to our knowledge only one implementation of KM3. The implementation is freely available under Eclipse, and KM3 itself is documented in academic publications [84].

2.7 Common Features and Differences

In this section, we summarize the finding of the four previous sections on GXL, MOF, ECORE and KM3. We discuss the common features and issues of the frameworks.

Comparing Sections 2.3, 2.4, 2.5 and 2.6, we can discern several common issues and differences between GXL, MOF, ECORE and KM3. It must be emphasized that MOF has the backing of an industry consortium which has enabled MOF and related technologies to evolve at a fast pace. Examples of these technologies are OCL, DI and QVT, not to mention the flagship metamodel UML, although there is perhaps an ever-increasing fear of “design-by-committee”, where a standard reflects few actual needs of its users. In contrast to this, GXL is more of a community-driven effort. ECORE, being part of the IBM-supported Eclipse environment, is taking the middle road between these two extremes. KM3 is the newest contender, and has already spurred the Kermeta [90] framework.

There are several similar features in the frameworks. We stress that it is not evident if a lack of a feature is unfavorable in itself, or if the presence of a feature is beneficial, although in most cases our features are beneficial in that they simplify the usage or increase the expressiveness of the framework. The similar or identical features are: abstract types, metamodel / schema packages, multiple inheritance, transformation language and typed elements.

On the metamodel/schema level, all frameworks have their positive and negative points. MOF has quite a complicated way to describe metaelement interconnections using References and AssociationEnds. It even has a second way to establish them, in the form of Attributes. Similarly, ECORE also has EReferences and

EAttributes, and KM3 has Reference and Attribute. The iD property in EAttribute seems superfluous from a modeling point of view, but could be useful for referencing elements from the textual domain. GXL contains a crude tagging mechanism in the form of (GXL) Attributes with key-value string pairs. We assume that this concept is included due to the roots of GXL being in describing graphs, which often use attributes for tagging nodes with arbitrary data. Its benefits are not clear for information modeling, especially since a composite edge to a string value would mostly serve the same purpose. This is somewhat similar to the two similar concepts in both MOF and ECORE.

Extensibility is accomplished in slightly different ways in all the frameworks and they work a bit differently: the XMI.Extension of MOF works only on the serialization level, EAnnotations from ECORE work only on the modeling level, and GXL suggests creating a modified DTD. This is a bit strange, given that XML itself employs a standard way to tag elements with extra data, simply by adding new XML elements from a different XML namespace. However, XMI prevents this. Extensibility on the model level and in the serialization format is a two-edged sword. While they certainly have benefits in that they allow a mechanism for arbitrary "tagging" of information, the drawback is that everybody must agree that similar tags are treated similarly, and nonsimilar tags should not be mixed up.

The choice of having a separate (GXL) Graph metaelement for nesting is unusual for modeling information, and the benefit is not very clear. A GraphElement could transitively own other GraphElements, without apparent loss of expressivity. This would simplify the GXL graph model and metaschema since it reduces the number of concepts that must be defined. In fact, since the metaschema already defines ownership via composition edges, GXL has a redundant second construct to express ownership of a subgraph.

Support for aggregation in MOF has been removed in newer versions, which means that there are some information systems that are awkward to describe in MOF. Indeed we have ourselves developed metamodels where aggregation would have been beneficial. To our knowledge, neither ECORE nor KM3 has ever had any support for aggregation. Support for aggregation in GXL among multiple files is not without problems, though. For example, it is not clear which file contains the shared subtree of nodes. On the other hand, several programming language runtimes have discarded the absolute notion of ownership and made the program data into effectively ownership-free object graphs by the use of garbage collectors. Enforcing ownership via associations, aggregations or composition might not be beneficial in the long run. A reason for this is that element ownership using the three alternatives is not complete, with [73] stating that there exists at least as many as 4096 different kinds of ownership when taking overlapping lifetimes, transitivity, shareability, separability et cetera into account. This implies that the frameworks under comparison omit the modeling of the exact semantics, and rely on the users of the individual metamodels or graph schemas to agree on which kinds of ownerships are meant.

GXL has support for hypergraphs whereas the others are restricted to binary edges. We have not found this to be much of an issue when creating metamodels, but it is worth researching further. For example, Steven Kelly from the MetaCase company has stated several times that “the lack of support for n-ary relationships is an astonishing oversight [in MOF/EMF]” [88].

GXL does not have any support for attaching operations to classes at all. This stems naturally from the fact that GXL aims to describe graphs and not objects. MOF and ECORE both support operations well, but KM3 does not. In the context of modeling information, we do not consider this shortcoming to be of much importance. On the other hand, a clear benefit for GXL is the ability to link to external resources using URIs. This can be used to bridge the gap between GXL documents and documents not yet maintained as GXL documents.

From a practical point of view, GXL does have an interesting idea with untyped elements, whereas the others are more stringent since they require that the metamodel is already developed. Nodes and edges without types provide the user the most flexible environment, but with the caveat that type checking might be necessary at a later stage. Similar advantages for dynamic typing have been found in scripting languages such as Python [180] and Perl [201].

GXL has only one serialization format, the GXL DTD, which serializes graphs, schemas and metaschemas. This has its advantages, but does require yet one GXL-specific validator, for example the GXL Validator [68] from the GUPRO project website [67], for validating the schema-graph relationship. XMI defines a serialization format for each metamodel. On one hand there is no extra level of indirection involved, but on the other hand there are multiple serialization formats. So, we do agree with Winter et al. that the “XMI/MOF approach requires different types of documents for representing schema and instance graphs” (p. 8 of [208]) and that this indeed is a drawback, but only because finding the definition of a previously unknown metamodel is impossible, as has been described in Section 2.4. If the metamodel is known, generic XMI reader and writer routines can be created: for example Coral supports reading XMI 1.x and 2.0 as well as writing XMI 1.2 and 2.0 for any known metamodel in a bit over 6 000 lines of C++ code.

Furthermore, Winter et al. claim that “XMI/MOF offers a general, but very verbose format for exchanging UML class diagrams as XML streams (p. 8 of [208]).” Our opinion is that the format is verbose due to the named slots, which GXL allows but does not require. We find the concept itself to be an advantage. For example, a class can own a set of attributes and a set of operations in two different slots. The serialization of these elements are cleanly separated into their own XML nodes. Also, navigation via named slots simplifies the manipulation and query of models.

GXL is itself very verbose when using typed nodes, as the filename of a schema must be repeated every time a type is used, assuming the schema is in a different file. Our anecdotal experiences suggest that due to this, GXL files can be several times larger than their XMI 2.0 equivalents. The choice of GXL or XMI as the underlying serialization format is not necessarily an exclusive one. There are pro-

Structural Features	**GXL**	**MOF**	**ECORE**	**KM3**
Edges between edges	X	-	-	-
Extensibility in Models / Graphs	X	-	X	-
Extensibility in Serialization Format	-	X	X	X
Globally Unique Identifiers	-	X	X	X
Hyperedges	X	-	-	-
Operations	-	X	X	-
Ordered Superclasses	-	X	-	-
Shared Composition	X	-	-	-
Subsets	-	-	-	X
Untyped Elements	X	-	-	-
Practical Features	**GXL**	**MOF**	**ECORE**	**KM3**
Constraint Language	-	X	(X)	(X)
Diagram Support	-	X	(X)	(X)
Language Identification and Acquisition	X	-	-	-
URIs for External Resources	X	-	-	-

Table 2.1: Framework Differences

grams that can transform, at least partially, data from one format to the other. For example GUPRO hosts the XIG [209] tool, and Coral can output GXL files suitable for visualization using Graphviz [65].

Where MOF (or, perhaps, OMG technologies) outperforms GXL is in its handling of constraints using OCL. OCL has become well-established in the modeling community and allows additional arbitrary wellformedness constraints to be added to metamodels and models. Naturally, this does not prevent a constraint language to be added to GXL, but the point here is pragmatic: OCL exists currently and is in wide use, whereas we do not know of a similar effort based on GXL.

We present the main conceptual differences in the frameworks in Table 2.1, split into a structural and a practical part. In the table, an X means that the feature is supported, and a hyphen means that the feature is not supported. Since ECORE and KM3 are so close to MOF, they could easily adopt technologies such as OCL or DI, even though these are technically part of the OMG standards; these are marked with (X). All frameworks lack a single concept for defining the relationship between metaelements, for some reason always opting for two separate concepts (usually designated *references* and *attributes*). Only KM3 provides some of the new concepts from MOF 2.0, namely subsets and derived unions, although their semantics is not explained further. None of the frameworks provide redefinitions and package merges.

The largest differences between the frameworks are found in serialization, diagram support, constraint handling and support for hyperedges and edges between edges. A concise summary is that GXL has the least restrictions of the four frameworks and is perhaps most suitable for rapidly changing requirements or (research) experimentations. However, users must be ready to describe constraints in some query language or natural language to obtain the same static rigor as can be accomplished with MOF and OCL constraints.

2.8 Related and Future Work on Modeling Frameworks

Modeling and metamodeling platforms are becoming more of a commodity all the time. A highlevel view of the current situation is presented by Harald Kühn and Marion Murzek in [97]. Interoperability between metamodeling platforms is becoming more important. We would thus want to find all the necessary concepts for modeling information. Failure to support a concept directly in a framework or by means of a lossless transformation to supported concepts means that the transformation of data to that framework is not possible.

Similar views and ideas on general-purpose metametamodels can be found in [7] and [182]. In contrast with the metacircular definition, the work of Thomas Baar avoids the metacircularity with a set-theoretical framework [10] to describe the abstract syntax of object-oriented / graph languages.

Advances in metametamodels are few and occur seldom, but it is clear that they do occur. In particular, the new concepts of subsets and derived unions from MOF 2.0 are interesting and novel ideas and will be explored more thoroughly in Chapter 3.

Even though it is not necessary to create a metametametamodel, one emerges as a side-effect from a generic modeling platform, as can be seen in Figure 2.10. A sufficiently expressive metametamodel can be used to create metamodels from the other metametamodels, and transformation technology implies that all of this ought to be transparent to the end user. In the figure, the different UML metamodels are equivalent in expressivity, although the metamodels might be used and manipulated with different programming interfaces since they are defined by different metametamodels. The generic metametamodel might even be one of the existing metametamodels.

This opinion is not shared by all. Jean Bézivin, Guillaume Hillairet, Frédéric Jouault, Ivan Kurtev and William Piers state that the idea of a common metametamodel for all artifacts is naïve and that one technical space will never prevail on the other ones [22]. This is true insofar as modeling frameworks continue to evolve. In other words, until modeling is a relatively mature technology, new improvements and ideas will be presented and evaluated, and there is no clear best metametamodel. Bézivin et al. further propose a solution of bridging metamodels and models from different technical spaces by means of several mappings. It is noteworthy that they use KM3 as a pivot for transforming between Microsoft DSL Tools, Ecore and MOF 1.4. This is understandable, since creating separate transformations from/to

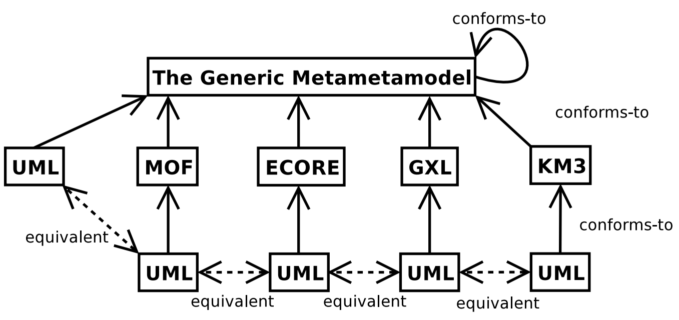


Figure 2.10: The Generic Metametamodel

each metametamodel requires $(N-1)N$ different transformations, where N is the number of metamodels, whereas using a pivot reduces this to linear complexity, $2N$. Due to differences between the metametamodels, some information is lost in the transformation processes. Nevertheless, in our opinion KM3 tries to play the role of a common metametamodel.

There are, naturally, more modeling frameworks than the ones presented. Examples of such are the MOF 2.0 framework, the XMF/XCore system from Xactium [43], Kermeta [90], the Open Modeling Framework (OMF) [172] and the SMILE/MATER project [122]. We think that a long-term goal of metamodeling is to extract the fundamentals in modeling information.

2.9 Conclusions

In this chapter, we have investigated some common modeling frameworks: their intent, differences in expressing structure and some practical aspects of their respective supportive technologies.

There are some fundamental questions that are unanswered in this chapter. The first is which concepts are the most useful ones in a metamodeling language, and how these concepts interact with each other. Experience in software engineering and other practices aids tremendously in evaluating a concept. However, modeling brings us new ideas hitherto unused or badly emulated in mainstream programming languages. An example of such a concept is bidirectionality of properties and slots. This does not exist as a primitive in programming languages, but is often emulated by a sequence of operations. Furthermore, modeling to us means information modeling, whereas programs have execution semantics.

The second is whether or not we want all data to be manipulated uniformly by the common constructs provided by the metamodeling language. The current APIs of modeling frameworks are fairly lowlevel, with primitives such as creation of new elements and linking elements via specific slots. However, most large-scale modeling languages come with several rules for wellformed models, and using primitive operations undermines the validity of these models. A better approach could be to create specific operations for each metamodel to modify models. In this way, models could be built from a series of operations while always being valid, or at least satisfying most wellformed rules. An example of this approach to designing models is being done by Dubravka Ilic, Elena Troubitsyna, Linas Laibinis and Sari Leppänen in [77], which describes a UML 2.0 profile called Lyra. To ensure model consistency, all operations on models are done by formalizing them in the B Method [1], which can formally prove that operations preserve the invariants of a structure.

The drawback from this approach is its rigidity. It is impossible to create partial models that only describe some of the information. In some contexts, only a partial model is necessary instead of a complete model. It is also nontrivial to figure out which sequence of operations must be applied to create some specific model.

In conclusion, the research community is still in the stages of identifying the necessary concepts in the metametamodel, and when to enforce wellformedness. From a formal point of view, more wellformedness checking, e.g., in the form of static strong typing and proven transformations, should lead to better results. However, in practice the trend in programming language design seems to be toward a more loose approach with dynamic typing.

Chapter 3

A Structure of Metamodels and Models: the Simple Metamodel Description Language

3.1 Introduction

In this chapter, we present a set-theoretic formalization of a metamodeling language that supports what we consider to be the core structural features of MOF 2.0 and the UML 2.0 Infrastructure, the two relatively new OMG standards. Our formalization supports multiple class specialization and the new subset properties.

The OMG software and system modeling standards are based on the concept of metamodeling. Most of the concepts found in the modeling frameworks described in Chapter 2 are strikingly similar since they are all based to some extent on the object-oriented (OO) software paradigm. In these approaches, a metamodel is defined as a collection of classes and properties while a model is an instance of such classes and properties.

However, two new metamodeling languages have recently emerged with the advent of the UML 2.0 Superstructure [135]: MOF 2.0 [139] and the UML 2.0 Infrastructure [142]. These two metamodeling languages share most of the features of previous languages such as MOF 1.4, but they also introduce several new concepts not found in traditional modeling and OO programming languages, mainly: *subset* properties, *strict union* properties and property *redefinitions*. These new concepts can be used to define a new modeling language as an extension of an existing one. This is exploited in the definition of UML 2.0 itself, where the language is defined as a relatively small core that is extended and specialized into different metamodeling structures.

Unfortunately, very little is told in the standards [139, 142] about the actual meaning of these new features. This is a critical omission since these concepts are heavily used in the definition of UML 2.0. A precise definition of the UML 2.0

metamodel is necessary in order to ensure interoperability of software modeling tools, such as model editors, model transformation and code generation tools.

The work presented in this chapter can also be applied to the definition of new domain-specific modeling languages (DSML) [58]. Although this chapter presents a theoretical framework, we believe it represents an important contribution that can influence the practical implementation of model repositories and transformation tools for UML 2.0 and other languages. We call this metamodeling language the Simple Metamodel Description Language (SMD). The implementation of it in Coral will be explained in Chapter 5.

This chapter is based on Publication IX. The research presented in this chapter is based on the study of the relevant OMG documents and research papers on related topics and also on the experiences obtained by developing an experimental modeling tool. Also, the experimental modeling tool Coral has been extended to implement and validate the ideas presented in this chapter and it is available as open source.

We proceed as follows: Section 3.2 describes informally the new language extension mechanism and presents the main motivations for our work. Section 3.3 presents the most basic formalism that will be developed and extended during this chapter. Section 3.4 introduces property characteristics such as multiplicity and composition, while Sections 3.5 and 3.6 deal with class and property specialization, respectively. Alternative approaches to property specialization are described in Section 3.7. Finally, Section 3.8 contains related work while Section 3.9 contains some concluding remarks.

3.2 Extension Mechanisms in MOF 2.0 and the UML 2.0 Infrastructure

MOF 2.0 and the UML 2.0 Infrastructure propose mainly four extension mechanisms: class specializations, property subsets and unions, property redefinitions and package merges. Class specialization is identical to class inheritance in OO languages. A specialized class inherits all the properties of its base classes, and it can define new properties. Subset and union properties are two mechanisms to specialize a property defined in a base class. Property redefinition allows us to arbitrarily replace a property with another one. Finally, various package merges allow us to combine different documents describing different (parts of) metamodels into one. These mechanisms allow a metamodel to be developed and extended by different parties. They can also be useful to define large metamodels, such as the UML 2.0 Superstructure, even when the definition is provided by one party.

We will describe class specialization, property subsetting and package merges in the remainder of this section. We also explain the Liskov Substitutability principle. We will leave property redefinitions to Section 3.7.

3.2.1 Class and Property Specialization

Figure 3.1 contains an example of the use of subset properties in a domain-specific modeling language. We abstract electronic components and their interconnecting wires in a digital electronic circuit into three classes, Wire, Pin and Component. An example specialization of Component is Transistor, which represents a transistor connecting to three pins: base, collector and emitter. While a generic component may have an arbitrary number of pins, a transistor may only have three specific pins.

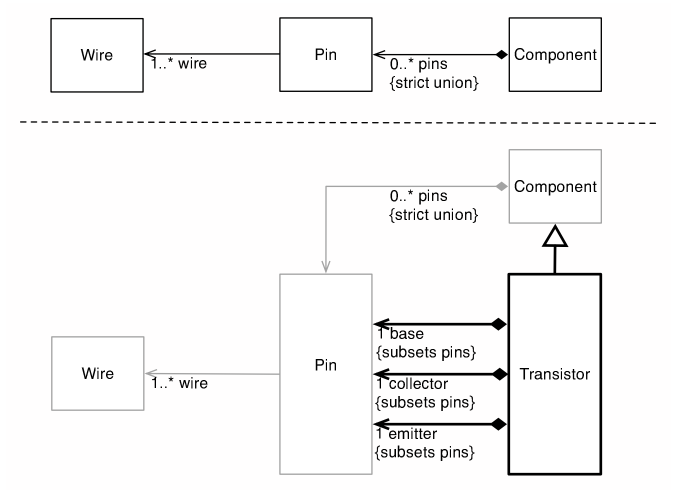


Figure 3.1: (Top) Base Language for Electronic Circuits (Bottom) Example Extension of the Digital Circuit Metamodel by Specialization of Component and its Properties into Transistor

This example also shows that subsetting does not partition the subsetted property with respect to classes; several subset relations can be built between the same two classes. We should also note that subset properties can be useful even when they are not used in combination with class specialization. That is, we can define a property and its subsets in the same class. We should note that the pins property needs to be a strict union. Otherwise we could connect a transistor to other pins that are not the base, collector or emitter.

3.2.2 Criteria for Language Extensions

Most of the artifacts that compose a model-driven development method such as model transformations, model queries and code generators depend on a specific modeling language, such as the UML 2 Superstructure. Since it is now possible to extend and modify metamodels we should consider what is the impact of these extensions in model transformations, model queries and code generators.

Our main criteria for language extensions is that artifacts defined for the original language should still be usable in any of its extensions. This concept is similar to the Liskov Substitutability [108] used in program type systems. This is not a surprise since a modeling language can be seen as a type for a model transformation program. As a consequence, a language extension should not be able to arbitrarily redefine or remove classes or properties from a language since existing artifacts may depend on them.

As an example of Liskov Substitutability, consider Figure 3.1 and a transformation that takes a Component as its input parameter, producing as output the various pins that the Component has. If we were to pass in a subclass of Component, for example Transistor, as a parameter to this transformation, we would still expect it to work as intended. But if the Transistor class could somehow remove the pins property between itself and Pin, our transformation would fail. In that hypothetical case, a Transistor would not be Liskov-substitutable for a Component.

Specialization is a very strict and limiting concept as it implies a tight coupling between classes. Therefore Liskov Substitutability is also equally limiting. However, it does provide a clear mathematical foundation which makes reasoning about programs and models easier.

3.2.3 Package Merge

We consider that package merges, albeit important, only influence the division of a metamodel into different documents. They do not influence the relationship between model elements. A metamodel using package merges can always be transformed into a metamodel without package merges. Thereby its semantics are defined by this transformation operation and as such do not provide a new relationship construct to metamodel developers. This has already been noted by Jim Steel and Jean-Marc Jézéquel in [166]. This does not mean that package merging is not useful, just that it is not necessary to discuss it within the scope of this chapter.

Therefore, we do not study in this chapter the concept of package merges and we focus on the semantics of subset properties since we consider that these are the main novelties in MOF 2.0 and the core mechanism for language extension.

3.3 Metamodels and Models

In this section, we discuss a basic metamodeling language based on classes and properties. This basic language does not include any extension mechanism, but it will serve to explain the most basic concepts that appear in MOF and UML in detail. We will denote our basic language with a subscript B in the names of various structures and functions. In the following sections we will add generalizations (class specializations) and property subsets as successive features in languages G and S, respectively. We proceed in this fashion in order to simplify the exposition

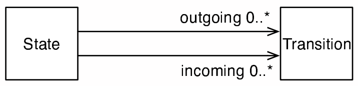


Figure 3.2: A UML Class Diagram Representing a Partial Metamodel for Statecharts

of these concepts in this chapter. Also, we want to show how each new concept in a metamodeling language interacts with the existing ones, sometimes in rather unexpected ways.

3.3.1 Metamodel Formalization

Metamodels are composed of classes and properties. A class represents a kind of abstraction that can appear in a model such as a state or a transition in a Statechart [69], while a property represents a basic relationship between these abstractions such as the fact that each transition has a source state and a target state. Models, on the other hand, are described using elements and slots, where each element conforms to one single class and each slot conforms to one single property.

UML and MOF use the UML class diagram notation to describe modeling languages visually. Figure 3.2 shows a part of a metamodel for a Statechart that we will use as our running example. This metamodel contains two classes: State and Transition, and two properties, outgoing and incoming. These two properties belong to State and have Transition as their type. A property may also contain several annotations that we call property characteristics. In the figure, we can see the multiplicity characteristic of the properties as the label "0..*".

Formally, we define all modeling languages in our basic metamodeling framework as a tuple $ML_B \overset{\text{def}}{=} (C,\ P,\ \text{owner},\ \text{type},\ \text{characteristics})$ where C is a finite set of classes, P is a finite set of properties and $C \cap P = \emptyset$. In our example, the set of classes is $C = \{\text{State}, \text{Transition}\}$, while the set of properties is $P = \{\text{incoming}, \text{outgoing}\}$.

Each property has a class as an owner, and this fact is indicated by the function $\text{owner} : P \rightarrow C$. The function $\text{properties} : C \rightarrow \mathcal{P}(P)$ gives the properties that belong to a specific class such that $(\forall c \in C \cdot \text{properties}(c) \overset{\text{def}}{=} \{\, p \cdot c = \text{owner}(p)\,\})$. In our running example, we have $\text{owner}(\text{incoming}) = \text{State}$ and $\text{owner}(\text{outgoing}) = \text{State}$, while the properties of the classes are $\text{properties}(\text{State}) = \{\text{incoming},\ \text{outgoing}\}$ and $\text{properties}(\text{Transition}) = \emptyset$. Finally, the function $\text{type} : P \rightarrow C$ denotes the type of elements in the property. In our example, both properties have the class Transition as their type, therefore $\text{type}(\text{incoming}) = \text{type}(\text{outgoing}) = \text{Transition}$. We define the characteristics of a property in detail in Section 3.4.

We can now define a specific metamodel L as any nonempty finite subset of the set of classes C, i.e., $L \subseteq C$. From a theoretical point of view, the concept of "one

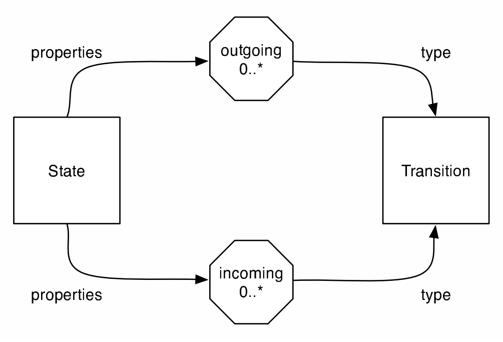


Figure 3.3: Metamodel from Figure 3.2 as a Graph of Classes and Properties

metamodel” is not too interesting. We consider the concept of “all metamodels”, i.e., C, to be more important and interesting.

We can represent ML_B as a labeled directed graph $G = (V, A, l)$. The set of vertices V is the union of the set of classes and properties, $V = C \cup P$. The set of arcs A contains two arcs for each property, one from the owner of a property to it and one from the property to its type: $A = \{\, (\text{owner}(p), p) \,\cdot\, p \in P \,\} \cup \{\, (p, \text{type}(p)) \,\cdot\, p \in P \,\}$. The property characteristics are represented as labels l over the nodes of the graph.

An example of this representation of Figure 3.2 is shown in Figure 3.3. To facilitate the comprehension of the graph, we represent classes as rectangles and properties as octagons. Although this notation is less compact than UML class diagrams, it maps better to the structures defined by ML_B. Additionally, we explicitly refrain from using UML in order to make it clear that UML is not a prerequisite for the metamodeling language described in this chapter.

Understanding modeling languages and models as graphs brings many benefits to our approach since graph theory [160] provides a solid foundation to many modeling approaches and model transformation languages as described for example in [17, 184, 8].

3.3.2 Model Formalization

We define the infinite set of all models as $\mathcal{M} = \{\, M \,\cdot\, M \stackrel{\text{def}}{=} (E, \text{class}, S, \text{property}, \text{slotowner}, \text{elements}) \,\}$. M comprises all the models in a system at some specific time. E is a finite set of elements, S is a finite set of slots and $E \cap S = \emptyset$. Each slot has one element as its owner as represented by the function $\text{slotowner} : S \to E$. For convenience, we can also define the slots of a given element as the function $\text{slots} : E \to \mathcal{P}(S)$, where $(\forall e \in E \,\cdot\, \text{slots}(e) \stackrel{\text{def}}{=} \{\, s \,\cdot\, e = \text{slotowner}(s) \,\})$.

A slot may refer to a number of elements as its contents. This is represented by the function elements. The value of this function is either a set of elements if

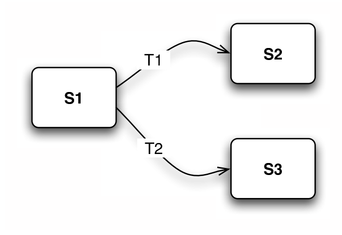


Figure 3.4: Example Statechart Represented in the UML Notation

the order of elements does not matter, and thus elements : $S \rightarrow \mathcal{P}(E)$; otherwise, the function returns a sequence of elements and we say elements : $S \rightarrow (E, \prec)$. We discuss the ordered characteristic in more detail in Section 3.4.2. We define the size of a slot to be the number of elements referenced by that slot: $(\forall s \in S \cdot \#s \overset{\text{def}}{=} \#\text{elements}(s))$.

Figure 3.4 shows an example model based on a Statechart language. In the example, the set of elements is $E = \{S1, S2, S3, T1, T2\}$ and $S = \{in1, out1, in2, out2, in3, out3\}$. Also, slotowner $= \{in1 \rightarrow S1, in2 \rightarrow S2, in3 \rightarrow S3, out1 \rightarrow S1, out2 \rightarrow S2, out3 \rightarrow S3\}$. Since the two properties are not ordered, we have elements$(out1) = \{T1, T2\}$, elements$(out2) =$ elements$(out3) =$ elements$(in1) = \emptyset$, elements$(in2) = \{T1\}$ and elements$(in3) = \{T2\}$.

Models are usually depicted using their own concrete syntax. For example, in a UML Statechart, states are represented as rounded rectangles and transitions as arcs. In this chapter, we use a generic syntax where all models are represent in an uniform way, independent of their modeling language.

We can now define a model B simply as any nonempty finite subset of the set of elements E, i.e., $B \subseteq E$. Analogously to metamodels, the concept of one model is not interesting from a theoretical perspective, and so we would rather work with all elements E.

Formally, we can represent models as a labeled directed graph $G = (V, A, l)$. The set of vertices V is the union of the set of elements and slots, $V = E \cup S$. The set of arcs A contains an arc for each slot and for each element reference in a slot, $A = \{(\text{slotowner}(s), s) \cdot s \in S\} \cup \{(s, e) \cdot s \in S \wedge e \in \text{elements}(s)\}$.

We depict each model element as rounded rectangle and each slot as a circle. Figure 3.5 shows the example model as such a graph.

3.3.3 Model Conformance to a Modeling Language

Each element in a model conforms to a class in a language and each slot conforms to a property. This conformance is represented by the functions class : $E \rightarrow C$ and property : $S \rightarrow P$ in a model. Together these functions link a model to its metamodel, thereby establishing what types the elements and slots have. They also

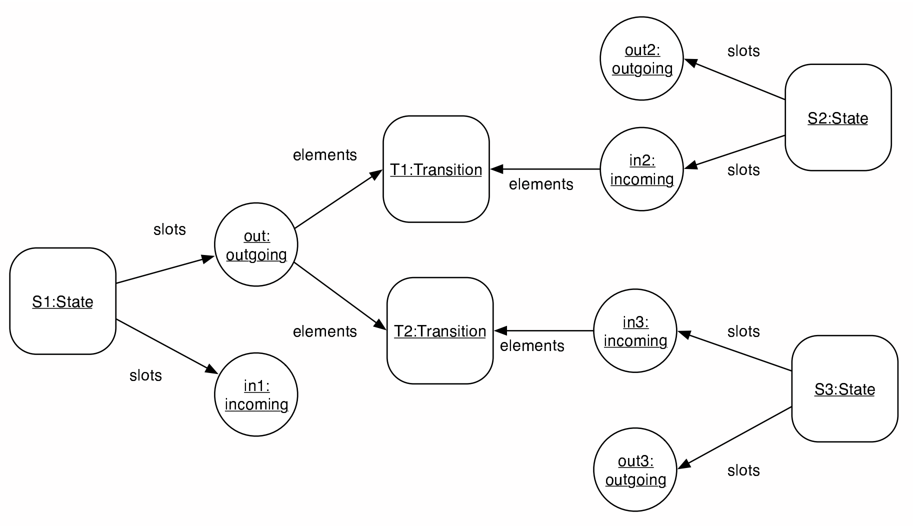


Figure 3.5: Example Statechart Model Represented Using the Generic Model Notation

imply that a certain set of constraints must be satisfied for any valid model, e.g., what slots can be owned by an element, the type of the elements in a slot and the number of elements in a slot.

A model *conforms* to its metamodel if it satisfies all these constraints. There is no reason to separate a model from its metamodel by disregarding the class and property functions. Such a model would merely be a graph of nodes connected by directed edges and, in most cases, would not contain enough information to be understood. Conformance is important because algorithms can then be sure that a certain structure holds.

We can now also define if a given model B is an instance of the modeling language defined by a metamodel L. This question can be asked in two slightly different ways: is B a direct instance of L or of an extension of L? B is a direct instance of L and not of an extension of it if B is a valid model and the type of each element in B is in L. Thus, B is a direct instance of L if $(\forall e \in B \cdot \text{type}(e) \in L)$ and all model constraints for elements in B hold.

On the other hand, B is an instance of either L or an extension of it if B is a valid model and the type of each element in B is a subclass of a class in L. Thus, B conforms to L if $(\forall e \in B \cdot (\exists c \in L \cdot \text{type}(e) \subseteq_c c))$ and all model constraints for elements in B hold.

The generalization of classes and the opposite of a property can be defined across different metamodels. Also, for an element e and a property p, $e.p$ denotes the slot that is owned by e and that conforms to p:

$$e.p \overset{\text{def}}{=} s, \text{ such that } \text{property}(s) = p \land \text{slotowner}(s) = e$$

3.3.4 Model Constraints

We note that constraints over a structure can be expressed using set theory. This has the benefits that set theory is formal and we do not have to declare a separate constraint language such as OCL, but the drawback is that the constraint language is not a metamodel within the framework.

The effective properties of a class is the set of all properties that can be used in an element conforming to that class. The constraints we write use the generic function name effectiveProperties $: C \rightarrow \mathcal{P}(P)$. Depending on the metamodeling language used (ML_B, ML_G or ML_S, for basic, with generalizations and with subsets, respectively), we can substitute different definitions in its stead. We do similarly for other functions as well, but will henceforth omit this explanation.

In a basic modeling language that does not support class specialization, the effective properties are simply the properties defined directly by the class. In Section 3.5, we will review this definition to take into account properties defined in superclasses.

$$\text{effectiveProperties}_B(c) = \text{properties}(c), \quad \forall c \in C$$

The function effectiveProperties$_B$ is specific to the metamodeling language ML_B. The effective properties of a class introduce two constraints over the elements conforming to that class. First, an element cannot have slots that do not conform to the effective properties of its class.

Model Constraint 1 *Valid slots in element (1):* $(\forall e \in E \,\cdot\, (\forall s \in \text{slots}(e) \,\cdot\, (\text{property}(s)) \in \text{effectiveProperties}(\text{class}(e))))$

Second, an element must have exactly one slot for each effective property in its class:

Model Constraint 2 *Valid slots in element (2):* $(\forall e \in E \,\cdot\, (\forall p \in \text{effectiveProperties}(\text{class}(e)) \,\cdot\, (\exists! s \in \text{slots}(e) \,\cdot\, \text{property}(s) = p)))$

The function effectiveType $: P \rightarrow \mathcal{P}(C)$ denotes the effective types of a property, i.e., the set of all allowed types for that property. In the basic language, the effective types of a property is defined explicitly by the type characteristic.

$$\text{effectiveType}_B(p) = \{\text{type}(p)\}, \quad \forall p \in P$$

The set of effective types of a property constrains the class of the elements that can be in a slot conforming to the property:

Model Constraint 3 *Class of elements in a slot:* $(\forall s \in S \,\cdot\, (\forall e \in \text{elements}(s) \,\cdot\, \text{class}(e) \in \text{effectiveType}(\text{property}(s))))$

Figure 3.6 shows the example metamodel for Statecharts together with a part of the example model. We have represented the class and property functions with dashed lines. Since models and metamodels are finite, we can easily check that the previous constraints hold for the example.

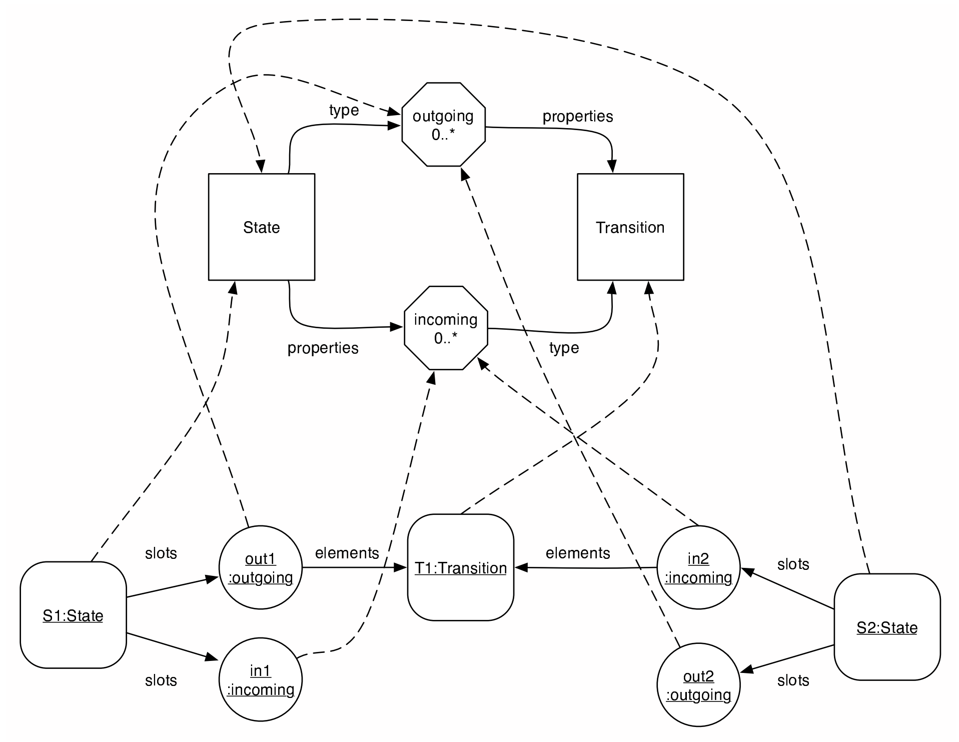


Figure 3.6: Conformance of a Model to its Modeling Language

3.3.5 Metamodel Constraints

Since a metamodel defines a set of constraints over a model, it can be possible to define a metamodel in such a way that there is no nontrivial model that conforms to it. Similarly, it may be possible to define a class so that there is no element that conforms to it, or a property so that there is no nontrivial slot that conforms to it. In these cases, we say that the metamodel, class or property is *void*.

Void metamodels or metamodels with void classes or properties are not useful in software development. For this reason, we will define metamodel constraints in our metamodeling approach. A metamodel constraint is a predicate over a metamodel that should hold in order to exclude void definitions.

3.3.6 Primitive Values

Hitherto, the models that can be described can only consist of elements interconnected via slots. It is often the case that we need to use primitive data values such as strings, integers, floating-point values and enumeration values. However, we consider the expressiveness of a framework to be in the various property characteristics; primitive values are fairly uninteresting. Nevertheless, we will give a brief description of how they can be added to the framework, but we will not consider them any further in this chapter.

We can add various classes that represent primitive data types to C. For example, we can say that $\mathbb{Z} \in C$ and $\mathbb{S} \in C$ denote the class of integers and the class of strings, respectively. Then, we add a partial function from elements to data values, $\text{primitivevalue} : E \nrightarrow \mathbb{Z} \cup \mathbb{S}$. It maps an element to its primitive value if said element is of the correct class. For example, an element e such that $\text{class}(e) = \mathbb{S}$ can be mapped to a string value, and thus $\text{primitivevalue}(e)$ returns a string. Modifying primitive values is done by modifying the function primitivevalue.

Thus, our primitive values are also elements and can technically contain slots as well, referencing other elements. While this is not common in for example programming languages, we feel this arrangement to be conceptually easier as it avoids further constraints.

3.3.7 Universal Identifiers

Some algorithms are made easier by assuming that elements can be identified by some kind of a unique identifier. Such an identifier for each model element allows us to differentiate between two instances of the same element (e.g., two different versions of it) and two elements that are similar. This distinction is fundamental to implement model management operations like comparing two models, merging two models into one or duplicating (parts of) a model. Unique identifiers can also be used to create traceability links between the artifacts in two different models, that may be expressed in two different languages. A solution to this has already been invented: Universally Unique Identifiers (UUIDs) [31].

UUID strings are assumed to be globally unique. They are long and too complex to be generated by hand but this is not an issue for the user since modeling tools should take care of this aspect. They are based on a 128-bit pseudorandom number generated from the physical address of the network interface in the host running the tool and the tenths of microseconds elapsed since the Gregorian reform (October 15th, 1582). The uniqueness of the UUID strings is not questioned. We note that an element retains its UUID across its evolution. UUID strings fulfill all the requirements to uniquely identify elements for model management and transformation traceability. An example of an identifier in the DCE namespace [31] is “DCE:2fac1234-31f8-11b4-a222-08002b34c003”.

We can formalize the concept of UUIDs by denoting the infinite set of UUIDs with $\mathcal{U}$. We can declare a function $\text{uuid} : E \to \mathcal{U}$ that maps an element to its unique identifier. The implementation of this function depends on the modeling environment in question. We will not try to define it further. We assume that any model element created in our framework will also belong to the domain of the uuid function and can be mapped to a UUID string.

From a practical point of view, it is worth noticing that the trend in other areas needing similar strings (e.g., Microsoft DCOM technology, where they are called

GUID) has been that generation is completely pseudorandom with no relation to the host, for security reasons.

3.3.8 Naming

Practical considerations dictate we be able to refer to classes and properties by a string which identifies them. Therefore we introduce a new function name : $C \cup P \rightarrow \mathbb{S}$.

We will not discuss naming conventions and restrictions or requirements of uniqueness of names in this chapter as they are dependent on the serialization technology and possibly even the programming interface that we use. As such, naming is not a part of the theoretical framework, although it is needed when we wish to implement a modeling framework in practice. Since our theoretical framework does not have the concept of one specific metamodel, there is also no possibility to give a name to it.

3.4 Property Characteristics

In the previous section we have studied how properties can be used to relate model elements together. In this section, we discuss how different property characteristics such as multiplicity or composition can be used to constrain even further how elements can be related via slots. We define the characteristics of a property as a tuple:

$$\text{characteristics}_B \overset{\text{def}}{=} (\text{lower}, \text{upper}, \text{ordered}, \text{composite}, \\ \text{opposite})$$

This tuple describes additional features of properties using several functions:

- lower : $P \rightarrow \mathbb{Z}^{0+} \setminus \infty$ represents the lower multiplicity constraint of a property (0, 1, 2, . . . , excluding infinity).
- upper : $P \rightarrow \mathbb{Z}^{+}$ represents the upper multiplicity constraint (1, 2, . . . , ∞).
- composite : $P \rightarrow \mathbb{B}$ is true if a property denotes composition.
- ordered : $P \rightarrow \mathbb{B}$ is true if a property denotes an ordered collection of elements.
- opposite : $P \rightarrow P \cup \{\Omega\}$ denotes the optional opposite of a property in a relation between two classes.

The rest of this section explains the semantics of these functions and how they affect several constraints on models.

3.4.1 Multiplicity

One of the simpler but more important concepts is the multiplicity constraint. The lower and upper characteristics constrain the number of elements that can be referenced by a slot:

Model Constraint 4 *Valid number of elements in a slot:* $(\forall s \in S \cdot \text{lower}(\text{property}(s)) \leq \#s \leq \text{upper}(\text{property}(s)))$

Since the number of elements in a slot is bounded by the multiplicity characteristics of its property, the lower value should be less than the upper value. Otherwise, the multiplicity constraint cannot be satisfied by any slot:

Metamodel Constraint 1 *Property Multiplicity:* $(\forall p \in P \cdot \text{lower}(p) \leq \text{upper}(p))$

Multiplicities are used extensively in the UML and MOF language. These languages support the concepts of multiplicity ranges, and the valid number of elements in a slot is a subset of $\mathbb{Z}^{0+}$. In practice, UML and MOF describe a multiplicity constraint as a set I of intervals (l,u) such that the valid multiplicity subset is equal to $\{x \cdot (\exists (l,u) \in I \cdot l \leq x \leq u)\}$.

3.4.2 Ordering

The ordering characteristic is used to model ordered collections of elements. An example of the usage of an ordered property is the parameters in a method of a class. Another more interesting example is shown in Figure 3.7. The top of the figure shows a metamodel for a modeling language for the ports of an active event-based component. A port accepts a number of events and this is modeled using an ordered property. The bottom of the figure shows an example model where ports *P1* and *P2* accept events *E1* and *E2*. However, port *P1* considers that event *E1* has priority with respect to event *E2*, while *P2* gives priority to event *E2*.

This example remarks the fact that the ordering characteristic does not define an ordering of elements but an ordering of the elements referenced by one particular slot. We should also note that the ordering as such does not introduce new constraints in a model, although ordering should be taken into consideration in all the model constraints.

3.4.3 Opposite Property and Bidirectionality

We have seen that a property can be used to define a UML or MOF 2 relation that is navigable by only one of its participants. However, we can also define bidirectional relations, by defining the opposite of a property.

Formally, the characteristic opposite : $P \rightarrow P \cup \{\Omega\}$ is a function that yields the opposite of a property, or the special constant Ω, which means that no opposite is defined. At the metamodel layer, we require that a property has itself as the opposite property of its opposite (iff it exists):

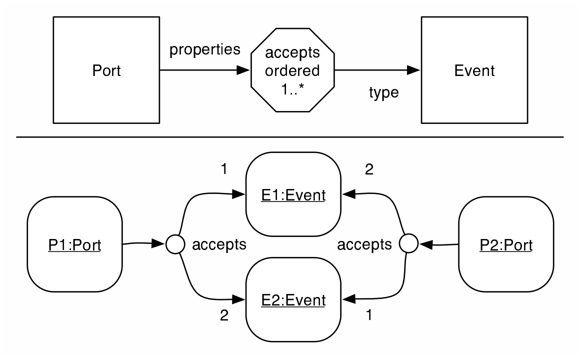


Figure 3.7: (Top) A Metamodel Using the Ordered Characteristic (Bottom) A Model Conforming to this Metamodel

Metamodel Constraint 2 *Opposite properties:* $(\forall p \in P \cdot \text{opposite}(p) \neq \Omega \Rightarrow p = \text{opposite}(\text{opposite}(p)) \wedge \text{opposite}(p) \neq p)$

Figure 3.8 depicts a reviewed metamodel for a statechart. In the example, each property has another property as its opposite: source and outgoing are opposites and form a relation, as well as target and incoming. In a model, this relation means that when a State *s* has a Transition *t* in its outgoing slot, the Transition *t* will have State *s* in its source slot. In Figure 3.9, state *S1* refers to transitions *T1* and *T2* in its outgoing slot, where the transitions refer to *S1* in their source slot.

At the model layer, we need to reflect that the contents of two opposite slots always refer to each other. This is captured in the following constraint for opposite slots:

Model Constraint 5 *Bidirectionality of slots:* $(\forall s \in S \cdot \text{opposite}(\text{property}(s)) \neq \Omega \Rightarrow (\forall e' \in \text{elements}(s) \cdot (\exists! s' \in S \cdot \text{slotowner}(s') = e' \wedge \text{opposite}(\text{property}(s')) = \text{property}(s) \wedge \text{slotowner}(s) \in \text{elements}(s'))))$

Our interpretation of relations is also shared by other authors, such as Génova, Ruiz del Castillo and Lloréns in [60]. However, according to some researchers, bidirectionality of two properties does not imply a bidirectionality requirement at the model layer. That is, model constraint 5 does not need to hold.

To see why this belief does not lead to a useful concept in a modeling language, consider Figure 3.10. It is a valid model according to the statemachine metamodel presented earlier in Figure 3.8, except for the fact that model constraint 5 does not hold. There are two cases where the constraint does not hold. First, the outgoing connection between *S2* and *T1* does not have a source connection from *T1* to *S2*. Second, the target connection from *T1* to *S1* similarly does not have an incoming connection from *S1* to *T1*.

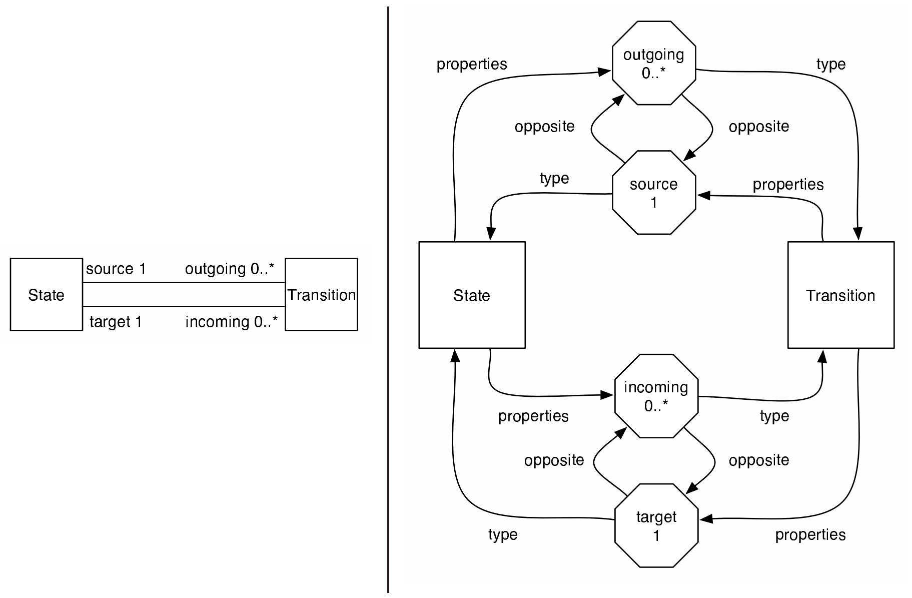


Figure 3.8: (Left) Metamodel for a Statechart Containing Navigable Relations (Right) The Same Metamodel as a Graph of Classes and Properties

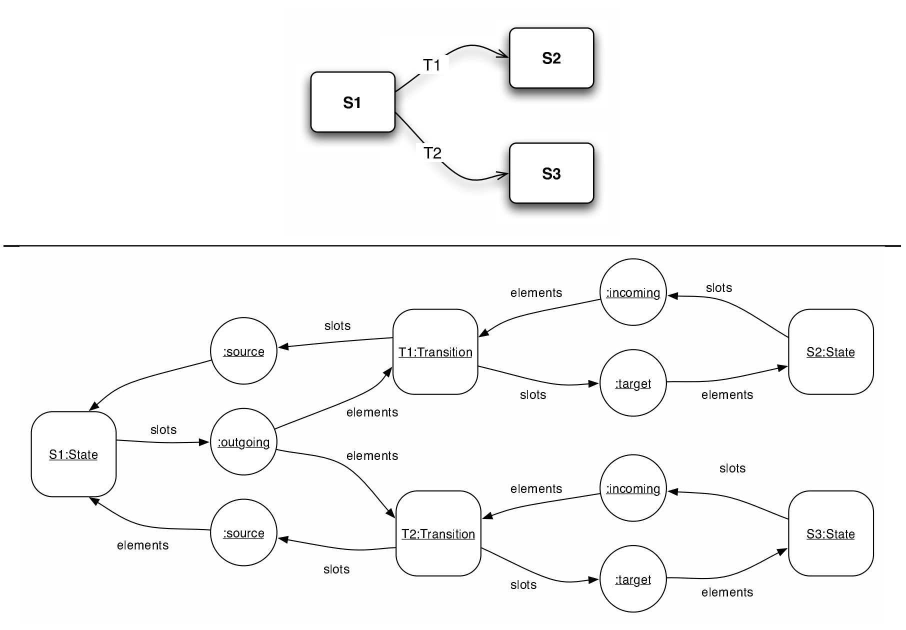


Figure 3.9: (Top) Example Statechart (Bottom) Statechart Represented as Elements and Slots, Conforming to the Metamodel from Figure 3.8

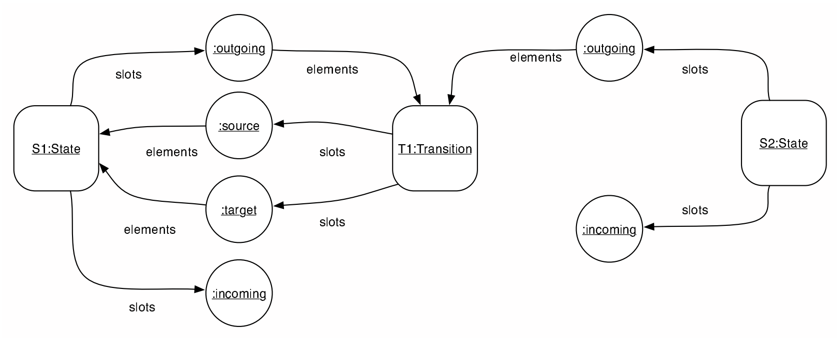


Figure 3.10: A Statemachine Model without Bidirectional Slots

We firmly believe that this example is nonsensical. If for some reason this is the intended interpretation of a modeling language, the metamodel should reflect it, as shown in Figure 3.11. In this new metamodel, the properties source and outgoing are not opposites, and therefore, there is no constraint between their slots.

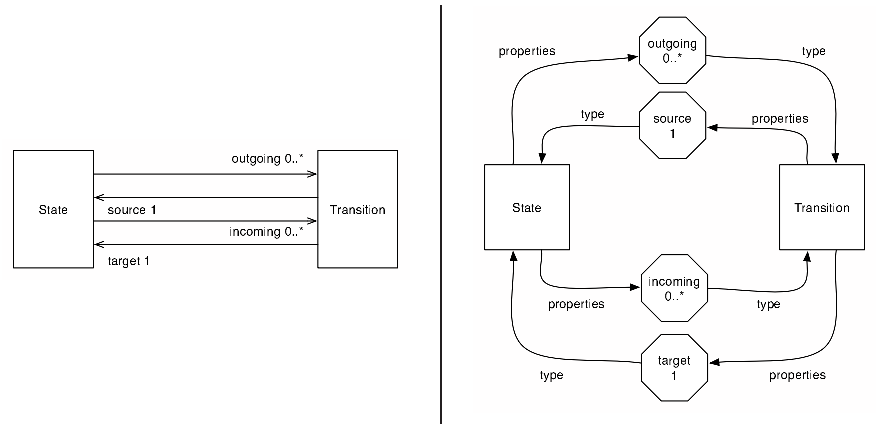


Figure 3.11: (Left) Metamodel for a Statechart with Unidirectional Relations (Right) The Same Metamodel as a Graph of Classes and Properties

3.4.4 Composition

Another important property characteristic is composition. Composition is used to denote hierarchy and ownership in a model. It is a very important concept that aids us in organizing models as a collection of smaller parts. A composition property imposes a rather restrictive constraint over its slots: an element can only be referenced by one composition slot at a time and it should not be possible to create cyclic compositions.

Figure 3.12 shows an example of the use of composition in a metamodel. The top of the figure contains a simplified metamodel, in both UML and our notation, for a class modeling language containing a package and a class. A package may contain classes and each class may contain inner classes. However, a class should not be directly owned by both a package and by a class simultaneously, and neither can an inner class directly or transitively be an inner class of itself. The bottom of the figure contains a model that presents these two cases: class *C1* is owned by package *P1* and class *C3* simultaneously and there is a composition cycle between classes *C1*, *C2* and *C3*. Therefore the model at the bottom of the figure does not conform to the metamodel at the top of the figure.

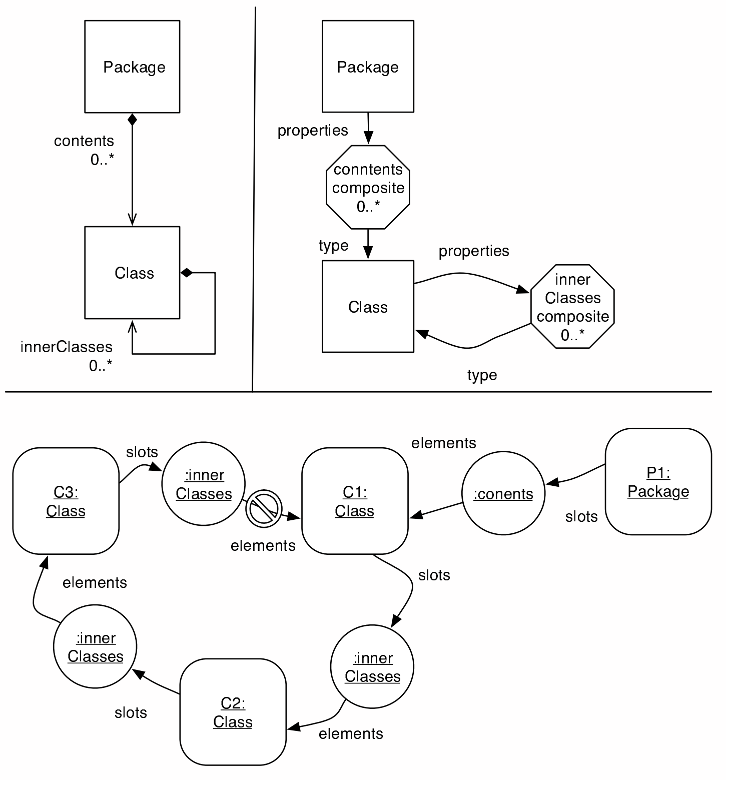


Figure 3.12: Example of Composition

Formally, we say that an element x is the owner or *parent* of an element e if e is referenced by a composite slot s of x. We define the function parent : $E \longrightarrow \mathcal{P}(E)$ to return either the empty set if no parent for an element exists, or a set consisting of all the parent elements. Thus, the size of this set should be at most one.

$$\begin{aligned}\text{parent}(e) \overset{\text{def}}{=} \{x \cdot x \in E \wedge (\exists s \in S \cdot \text{slotowner}(s) = x \wedge \text{composite}(\text{property}(s))\\ \wedge e \in \text{elements}(s))\}\end{aligned}$$

We will also make use of the function parentchain : $E \rightarrow (E, \prec)$, which returns a sequence of parent elements:

$$\begin{aligned}\text{parentchain}(e) \overset{\text{def}}{=} \ & \text{if } \#\text{parent}(e) \ = 0 \text{ then } [\,] \\ & \text{else } [\,p \in \text{parent}(e)\,] \triangleleft \text{parentchain}(p \in \text{parent}(e))\end{aligned}$$

Thus, an element cannot be owned by the same element via two different composite slots:

Model Constraint 6 *Only in one composite slot:* $(\forall e \in E \ \cdot \ \neg(\exists s_1, s_2 \ \cdot$ $\text{composite}(\text{property}(s_1)) \wedge \text{composite}(\text{property}(s_2)) \wedge e \in \text{elements}(s_1) \wedge e \in$ $\text{elements}(s_2))$

As stated previously, an element cannot be the owner of itself, directly or transitively:

Model Constraint 7 *Composition is acyclic:* $(\forall e \in E \ \cdot \ e \notin \text{parentchain}(e))$

This also implies that a relation at the metamodel level cannot be made from two composite properties, since such slots would be void.

Metamodel Constraint 3 *Both properties in a relation cannot be composite:* $(\forall p \in P \ \cdot \ \text{composite}(p) \wedge \text{opposite}(p) \neq \Omega \Rightarrow \neg\text{composite}(\text{opposite}(p)))$

Defining metamodels using both composition and multiplicity range characteristics has an interesting consequence. It is possible to declare a chain of several classes such that it is mandatory for an instance of a class to own at least one instance of the next class, et cetera, until a cycle is created. Thereby all classes would be void, since only an infinite chain of elements could conform to them. We can prohibit this with the following constraint:

Metamodel Constraint 4 *No infinite chain of compositions:* $(\forall c_1, \ldots, c_n, c_{n+1} \in C \ \cdot \ (\forall i \ \cdot \ 1 \leq i \leq n \Rightarrow (\exists p \in \text{effectiveProperties}(c_i) \ \cdot \ \text{composite}(p) \wedge \text{owner}(p) = c_i \wedge c_{i+1} = \text{type}(p) \wedge \text{lower}(p) \geq 1)) \Rightarrow c_1 \neq c_{n+1}), \quad \forall n \geq 1$

Composition is used extensively in the definition of UML and MOF and it also appears in the GXL. Its semantics in the context of UML has been studied by Barbier et al. in [16, 73]. Composition brings many advantages when building tools that need to traverse or transform models. If we only take slots of a composite property and elements into account, the resulting graph forms a tree (or a forest). This allows us to use efficient traversal algorithms. Also, this arrangement maps well to XMI, since an XML document has a tree structure.

In this chapter, we use the concept of *strict composition* or “black diamonds” as described in [73]. Another alternative interpretation of composition is *shared composition*. In this case, the composition links should be acyclic, but an element can have more than one parent. To achieve this interpretation of a modeling language, model constraint 6 should be removed. The resulting graph forms a directed acyclic graph.

3.4.5 Attributes

We should note that our metamodeling language does not have any special provision to model attributes such as in MOF or EMF. This is due to the fact that we can model an attribute by using a combination of the property characteristics that we have already defined. In our approach, we consider an attribute definition equivalent to a property that is a composition and does not have an opposite. This reduces the number of defined concepts at the metamodeling layer and thus simplifies the structure of metamodels and models. We have validated this idea in our modeling tool and found no problems.

3.5 Class Specialization

In this section we introduce the concept of class specialization as a mechanism to organize and simplify large metamodels. Class specialization is the same concept as class inheritance in OO programming languages. A class can be a specialization of one or more base classes and as a consequence it inherits all the properties of its base classes. Without class specialization, the definition of UML would require many additional properties and model transformations would be more complex and cumbersome to create and maintain. Therefore, class specialization is used extensively in the definition of UML and MOF.

As an example, the UML 1.x metamodels use class specialization to model state hierarchy. Figure 3.13 shows a simplified Statechart model where we specialize the State class into CompositeState. In UML, class specialization is represented diagrammatically as an edge between the base class and the specialized class with a triangular arrow head pointing to the base class. In our example, a CompositeState inherits all the properties of a normal state but adds an additional property to model substates. A substate can be a State or a CompositeState.

We should also note that a metamodeling language should support multiple inheritance since it is used extensively in MOF. This has already been noticed by for example Anneke Kleppe [92]. In order to formalize class specializations we will need to extend our definition of metamodels by adding the concept of generalizations of a class. A modeling language supporting class specialization is then defined by the tuple:

$$ML_G = (C, \text{generalizations}, P, \text{owner}, \text{type}, \text{characteristics})$$

We define the generalizations of a class with the function generalizations $: C \rightarrow \mathcal{P}(C)$. We denote by $\subseteq_c$ the extended generalization between classes that is defined as the reflexive transitive closure of the generalization relation: $\subseteq_c \overset{\text{def}}{=} \{ (c,d) \cdot d \in \text{generalizations}(c) \}^*$. Intuitively, given two different classes c and d, we say that c is a subclass of d iff $c \subseteq_c d$. We also note that $\text{characteristics}_G = \text{characteristics}_B$, since no new property characteristics need to be added.

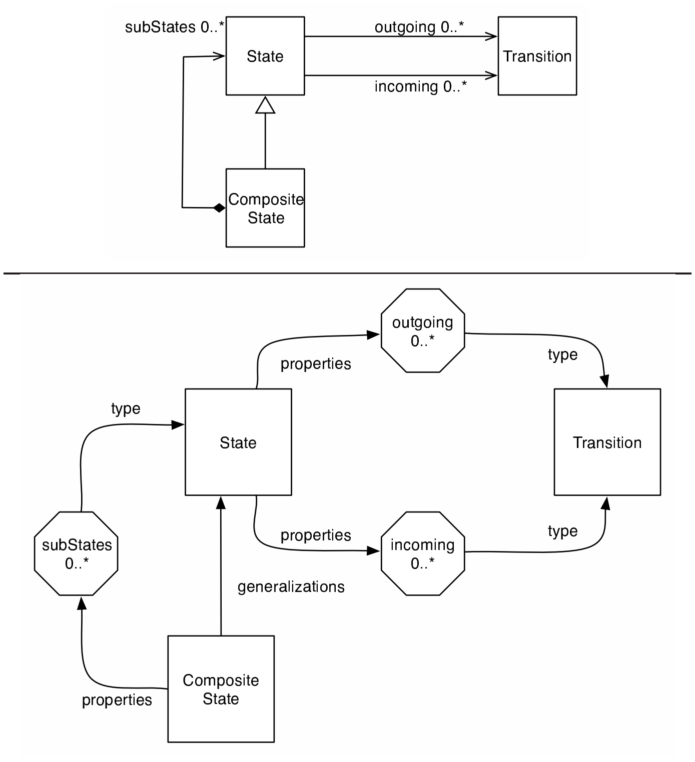


Figure 3.13: A Metamodel Using Class Specialization

We should now review the concept of effective properties of a class for a metamodeling language supporting class specialization. The effective properties of a class shall include all the properties owned directly by that class and all the effective properties of its superclasses:

$$\text{effectiveProperties}_G(c) = \bigcup\{\,\text{properties}(d)\ \cdot\ c \subseteq_c d\,\}$$

We should also review what the effective types of a property in a language supporting class specialization are. In this language, slots are covariant: a slot may contain elements whose class is the basic type of its property or any subclass of that type.

$$\text{effectiveType}_G(p) = \{\,c \in C\ \cdot\ c \subseteq_c \text{type}(p)\,\}$$

We can see an example of these definitions in the metamodel shown in Figure 3.13. We can see that $\text{effectiveProperties}_G(\text{CompositeState}) = \{\text{outgoing}, \text{incoming}, \text{subStates}\}$, i.e., the effective properties of CompositeState are the two properties owned by State and the property directly owned by CompositeState. Similarly, $\text{effectiveType}_G(\text{subStates}) = \{\,\text{State}, \text{CompositeState}\,\}$. An example of a model using this metamodel can be seen in Figure 3.14.

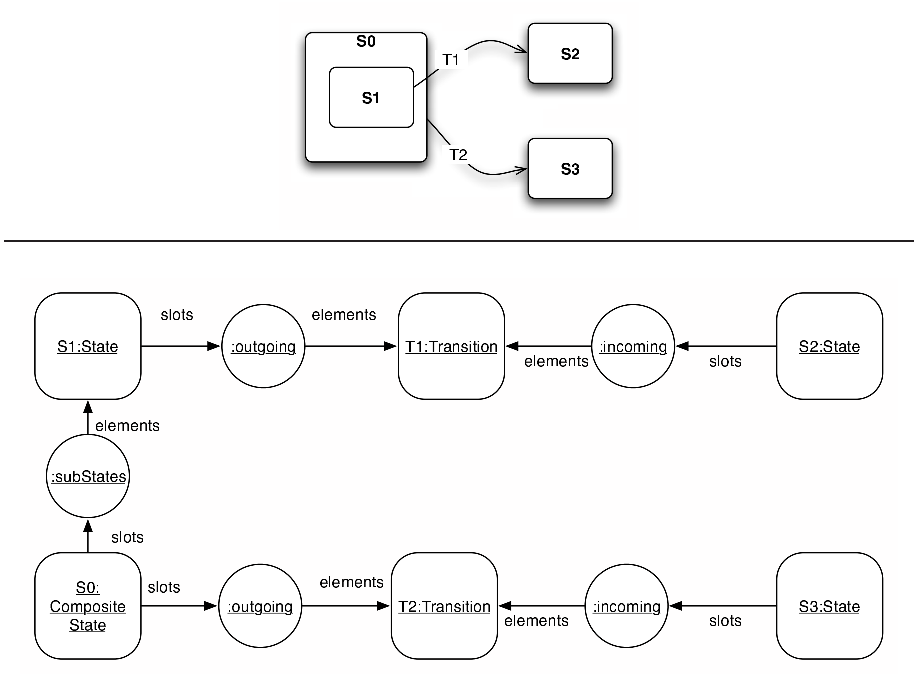


Figure 3.14: Example Model Using Specialization

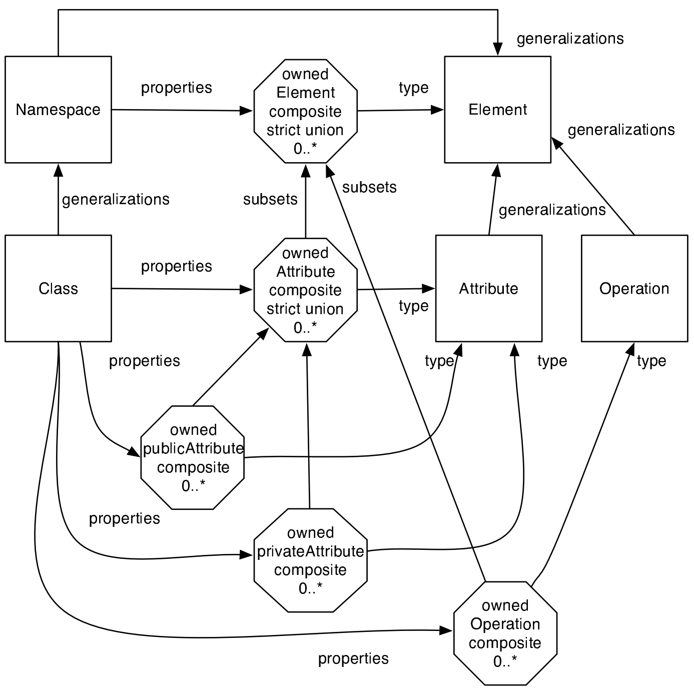
Because the effective properties of a class are defined by itself and its transitive superclasses, we require the generalization relation to be acyclic. Otherwise all classes in a generalization hierarchy would have the same set of effective properties and they would for all practical purposes be indistinguishable from each other. This leads to the following metamodel constraint:

Metamodel Constraint 5 *Generalization is acyclic:* $\neg(\exists e \in C \cdot (e,e) \in \{(c,d) \cdot d \in \mathrm{generalizations}(c)\}^{+})$

3.6 Property Subsetting

In this section we introduce the concept of property subsetting to our metamodeling language, one of the most intriguing concepts introduced in MOF 2.0 and the UML 2.0 infrastructure. Property subsetting allows us to specialize an existing property into a new property with a different basic type and different characteristics, while still retaining the old, existing property. The intuition is that the specialized property is a subset of the original property, meaning that elements in a slot of a subset property should also be included in the slot of the original property.

As an example, we present yet another version of a simplified metamodel for UML class diagrams in Figure 3.15. We first provide a general concept of a con-



Namespace
properties
owned Element composite strict union 0..*
type
Element
generalizations
generalizations
generalizations
subsets
subsets
Class
properties
owned Attribute composite strict union 0..*
Attribute
Operation
generalizations
type
properties
type
type
type
owned publicAttribute composite 0..*
properties
owned privateAttribute composite 0..*
owned Operation composite 0..*
properties

Figure 3.15: Example Metamodel for UML Class Diagrams Using Subset Properties

tainer and its children elements using the Namespace and Element classes. Each Namespace element has a slot named ownedElement representing its contents.

Then we specialize Namespace into a Class and add two *subset properties* called ownedAttribute and ownedOperation to keep attributes and operations. We can say that these properties are *subsetting* the ownedElement property. We also add two subset properties of ownedAttribute to Class, called ownedPublicAttribute and ownedPrivateAttribute. This example also shows how a subset and a union property may be in the same class.

We introduce a new property characteristic to our metamodeling language in order to support subset properties:

- supersets : $P \rightarrow \mathcal{P}(P)$ represents the properties of which a property is a subset.

We will also introduce the characteristic strictUnion later. Thereby, we can define a modeling language supporting property subsetting as the tuple:

$$\begin{aligned} ML_S &\overset{\text{def}}{=} (C, \text{generalizations}, P, \text{owner}, \text{type}, \text{characteristics}_S) \\ \text{characteristics}_S &\overset{\text{def}}{=} (\text{lower}, \text{upper}, \text{ordered}, \text{composite}, \text{opposite}, \text{supersets}, \\ &\quad \text{strictUnion}) \end{aligned}$$

We denote subsetting between properties by the $\subseteq_p$ relation, i.e., $\subseteq_p \stackrel{\text{def}}{=} \{ (p,q) \cdot q \in \text{supersets}(p) \}^*$.

In a model, we say that a slot r is a subset of another slot s if $\text{property}(r) \subseteq_p \text{property}(s)$ and they have the same owner. The slot subsetting relation is thus defined by:

$$\subseteq_s \stackrel{\text{def}}{=} \{ (r,s) \cdot \text{slotowner}(r) = \text{slotowner}(s) \wedge \text{property}(r) \\ \subseteq_p \text{property}(s) \}^*$$

The contents of a slot r subsetting another slot s must be a subset of the contents of s. Also, MOF [139] tells us on page 56 that *"The slot's values are a subset of those for each slot it subsets."* In the case of unordered slots, this is formalized using the following constraint:

Model Constraint 8 *Unordered slots:* $(\forall r,s \in S \cdot r \subseteq_s s \wedge \neg\text{ordered}(\text{property}(s)) \Rightarrow \text{elements}(r) \subseteq \text{elements}(s))$

We can see an example model based on subset properties in Figure 3.16. The model represents a UML class with one public attribute, one private attribute and one operation. The element representing the class has 5 different slots and the elements referenced in each slot is constrained by the subset properties. In the example we have $s_5 \subseteq_s s_1 \wedge s_4 \subseteq_s s_2 \wedge s_3 \subseteq_s s_2 \wedge s_2 \subseteq_s s_1$.

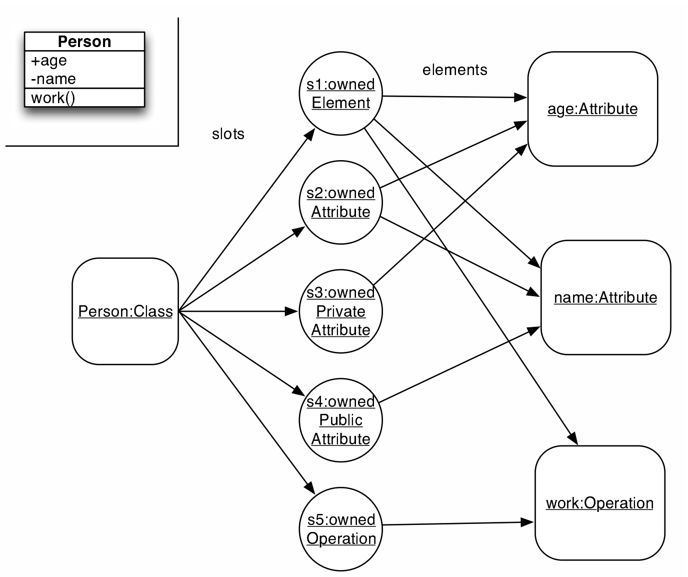


Figure 3.16: Example Model Based on the Metamodel Shown in Figure 3.15

In the rest of this section, we review how the concept of subset properties interacts with the other concepts in our metamodeling language.

3.6.1 Subsets and Ordering

For ordered slots we also wish to preserve order. That is, when elements occur in a specific order in r, they should occur in the same order in s, although s might contain more elements in between.

Model Constraint 9 *Ordered slots:* $(\forall x,y \in E, r,s \in S \cdot x \in \text{elements}(r) \wedge y \in \text{elements}(r) \wedge x \preceq_r y \wedge r \subseteq_s s \wedge \text{ordered}(\text{property}(s)) \Rightarrow x \in \text{elements}(s) \wedge y \in \text{elements}(s) \wedge x \preceq_s y)$

3.6.2 Union Properties

A property is called a union property if it has one or more properties that subset it. In our framework, it is not necessary to declare a property as a union, since a designer of a metamodel cannot know in advance if a new subset property will be defined in the future, possibly in some other metamodel. Another way to express this is to state that all properties are automatically union properties, and some can be declared as strict unions.

3.6.3 Strict Unions

The UML 2.0 Infrastructure also introduced the concept of strict union. The standard states on page 126 that *"This means that the collection of values denoted by the property in some context is derived by being the strict union of all of the values denoted, in the same context, by properties defined to subset it. If the property has a multiplicity upper bound of 1, then this means that the values of all the subsets must be null or the same."*. In other words, a derived union property can be seen as the strict union of its subsets. A slot with a property that is a strict union cannot contain elements that do not appear in any of its subsets.

We introduce a new property characteristic to our metamodeling language in order to support strict unions:

- strictUnion : $P \rightarrow \mathbb{B}$ is true if a property is a strict union.

The UML uses the qualifier "{ union }" to denote a property as a strict union. Since all our properties are unions, we use the qualifier "{ strict union }" instead. In the example, the contents of ownedElement and ownedAttribute slots are strict unions of the contents of the subsetting slots.

The concept of strict unions implies that a new model constraint needs to be defined:

Model Constraint 10 *Strict union:* $(\forall s \in S \cdot \text{strictUnion}(\text{property}(s)) \Rightarrow \text{elements}(s) = \bigcup\{\text{elements}(r) \cdot r \prec_s s\}$

We call model constraint 8, 9 and 10 the *inherent subsetting rules*, or ISR.

3.6.4 Subsets and Substitutability

The rationale for the proposed model constraints is to allow type substitutability in model transformations, queries and code generators. A specialized class has the same properties with the same characteristics as its base classes, while containing new definitions to specialize these properties. Thus, subsetting preserves Liskov substitutability.

As an example, Figure 3.17 shows a model based on the example metamodel for circuits shown in Fig. 3.1. Here, a given transistor can be seen as a generic component with a number of pins or as an element of type Transistor that has three specific pins. The benefit is that it is possible to define algorithms and transformations which work on the base abstract circuit; that is, only considering wires, pins and components, without caring for the details, whereas algorithms that are targeted to specific components can rely on a more refined abstract syntax.

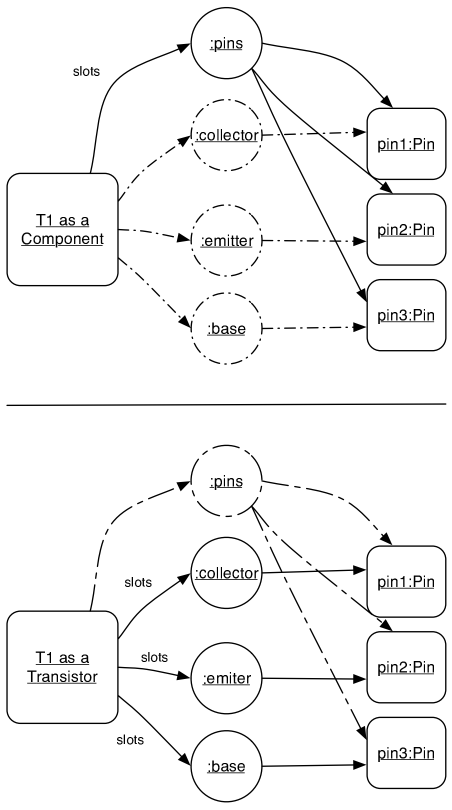


Figure 3.17: (Top) Example of a Model as Interpreted According to the Base Language (Bottom) Same Model Interpreted According to the Extended Language

3.6.5 Subsets and Multiplicity

It can easily be seen that a property subsetting another property should have a lower (or the same) upper limit than the other property. This can be formalized with the following metamodel constraints:

Metamodel Constraint 6 *Upper multiplicity in subset properties:* $(\forall p \in P \cdot (\forall q \in \text{supersets}(p) \cdot \text{upper}(p) \leq \text{upper}(q)))$

The justification for this constraint can be shown with slots r and s such that $r \subset_s s$, $\text{property}(r) = p$, $\text{property}(s) = q$ and $\text{upper}(p) > \text{upper}(q)$, and by filling the slot r with elements so that $\#r = \text{upper}(p)$. Then $\#s \geq \#r = \text{upper}(p) > \text{upper}(q) \Rightarrow \#s > \text{upper}(q)$, which violates the upper limit of q.

Property subsetting does not raise any new restrictions on the lower limits of the properties. This is because more elements can always be inserted into a slot until its size is at least that of the greatest lowest limit in any transitive sub- or superset. Thus model constraint 4 is sufficient for the lower multiplicity constraints.

3.6.6 Subsets and Class Specialization

A property should only subset another property if the effective types of it are a subset of the effective types of the other. The same remark is also stated in the UML 2.0 Infrastructure [142] on page 125: *A property may be marked as a subset of another, as long as every element in the context of the subsetting property conforms to the corresponding element in the context of the subsetted property.*

Figure 3.18 shows a (nonsensical) metamodel, in which property d of class C subsets property b of class A. The type of b is class B and the type of d is class D; however, D is not a subclass of B.

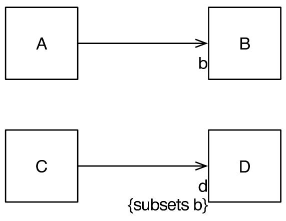


Figure 3.18: Subsetting Only One End of a Relation

We should explore how the existing constraints affect the elements in the slots of a model based on the this metamodel. We can easily find an example which shows that this particular way of using subsets is not useful. Since the elements in a slot s_d conforming to property d should be of type D, the elements in slot s_b conforming to property b should be of type B, and the elements in s_d should also be in s_b, the slot s_d cannot have any elements. This can be proven in the following derivation, based on the structured derivation approach in [14]:

$$
\begin{array}{ll}
 & s_d, s_b \in S \wedge \text{property}(s_d) = d \wedge \text{property}(s_b) = b \wedge \\
 & s_d \subseteq_s s_b \wedge \text{effectiveType}(d) = \{D\}, \text{effectiveType}(b) = \{B\} \\
\Rightarrow & \{ \text{ introduce class of elements in slot, unordered slot constraints, simplify } \} \\
 & (\forall e \in \text{elements}(s_d) \cdot \text{class}(e) \in \{D\}) \wedge \\
 & (\forall e' \in \text{elements}(s_b) \cdot \text{class}(e') \in \{B\}) \wedge \\
 & (\text{elements}(s_d) \subseteq \text{elements}(s_b)) \\
\Rightarrow & \{ \text{ definition of subset, membership in a singleton set } \} \\
 & (\forall e \in \text{elements}(s_d) \cdot \text{class}(e) = D) \wedge \\
 & (\forall e' \in \text{elements}(s_b) \cdot \text{class}(e') = B) \wedge \\
 & (\forall e'' \in \text{elements}(s_d) \cdot e'' \in \text{elements}(s_b)) \\
\Rightarrow & \{ \text{ elements in } d \text{ should also satisfy the constraint for } b \text{ } \} \\
 & (\forall e \in \text{elements}(s_d) \cdot \text{class}(e) = D \wedge \text{class}(e) = B) \\
\Rightarrow & \{D \neq B\} \\
 & \text{elements}(s_d) = \emptyset
\end{array}
$$

Based on this discussion, we consider an additional constraint over a metamodel: the fact that a property can subset another property only from the reflexive transitive superclass closure of its owner:

Metamodel Constraint 7 *Subset only from owner or its superclasses* $(\forall p, q \in P \cdot p \subseteq_p q \Rightarrow \text{owner}(p) \subseteq_c \text{owner}(q))$.

However, this restriction is not strong enough. It would still be possible to cyclically subset a property within the same class. This is not a useful construct since any slots of the properties in the cycle would consist of the exact same elements. Thus property subsetting must be acyclic:

Metamodel Constraint 8 *The property superset relation is acyclic:* $\neg(\exists e \in P \cdot (e, e) \in \{(p, q) \cdot q \in \text{supersets}(p)\}^+)$

Given these constraints, we can conclude the property subsetting is a partial order. In other words, $(P, \subseteq_p)$ is a poset. Thus, on the model level we have several partial orders, one for each slot and its transitive sub- and supersets, owned by some specific element.

It can be noted that multiple inheritance forms very complicated inheritance hierarchies, among them the *diamond inheritance* structure. This leads to a possibility where property subsetting also has a diamond (or even more complicated) structure.

3.6.7 Subsets and Opposite Properties

We should also study the possible interactions between subset and opposite properties. Let us consider the metamodel in Figure 3.19. In this metamodel, the subset property d has an opposite property c which is not a subset.

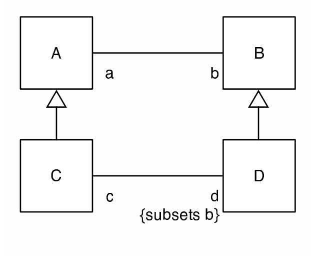


Figure 3.19: Example of the Interaction of Subsets and Opposite Properties

Let us assume that there is a model with two elements e_c and e_d conforming to classes C and D, respectively. According to the metamodel, element e_c has two slots that we name s_d and s_b conforming to properties d and b, respectively. Similarly, element e_d has two slots s_c and s_a. We wish to add element e_d to the slot s_d. Since the property d has an opposite, we also need to add e_c to the slot s_c of e_d in order to satisfy model constraint 5 regarding bidirectional slots. Since $d \subset_s b$, we also need to include e_d in the slot s_b to satisfy model constraint 8 about unordered subset slots. Finally, since property b has an opposite property named a, we should include e_d in the elements of s_a. Thus, e_d is in s_a and in s_c, and the net effect is as if c were a subset of a anyway, even though that was not stated in the metamodel.

The conclusion is that we claim that c needs to subset a, if for nothing else than documentation purposes. There are several faults in MOF 2.0 and UML 2.0 where this rule is violated. Fortunately, the correction is simple by saying that (in our example) c needs to subset a. In any case, this example emphasizes the need for the following constraint:

Metamodel Constraint 9 *The opposite of a subset property must be a subset:* $(\forall p, q \in P \cdot p \subseteq_p q \wedge \text{opposite}(p) \neq \Omega \Rightarrow \text{opposite}(p) \subseteq_p \text{opposite}(q))$

3.6.8 Subsets and Composition

We need to redefine the composition constraints to take into account the subset properties. Due to the subset constraints, an element may be in more than one composition slot at a given time, as long as these slots are not independent. This replaces model constraint 6:

Model Constraint 11 *(Subset) Only in one composite slot:* $(\forall e \in E \cdot \neg(\exists s_1, s_2 \cdot \text{property}(s_1) \,||_p\, \text{property}(s_2) \wedge \text{composite}(\text{property}(s_1)) \wedge \text{composite}(\text{property}(s_2)) \wedge e \in \text{elements}(s_1) \wedge e \in \text{elements}(s_2))$

In Figure 3.20, we see different cases with composite and noncomposite properties. Cases (1) and (2) are legal and quite self-explanatory: in the first case, all

A and *C* elements own their *B* and *D* elements via the *a–b* relation. In the second case the *c–d* relation can be used to own some of the *D* elements and the rest can be referenced via only the *a–b* relation, and can thus be owned by some other element.

Case (3) can be considered legal by discounting the composition at the *c–d* relation without any loss in information, since any elements owned via the *c–d* relation must also be owned via the *a–b* relation. This is what model constraint 11 allows. Case (4) is void since any elements of types *C* and *D* that are connected at the *c–d* relation are also connected at the *a–b* relation, thereby creating a cyclic composition and therefore violating a model constraint. Thereby the following metamodel constraint should be defined:

Metamodel Constraint 10 *No circular transitive composition with subsets:* $(\forall p \in P \cdot \text{composite}(p) \Rightarrow \neg(\exists q \in P \cdot \text{opposite}(q) \neq \Omega \wedge p \subset_p q \wedge \text{composite}(\text{opposite}(q))))$

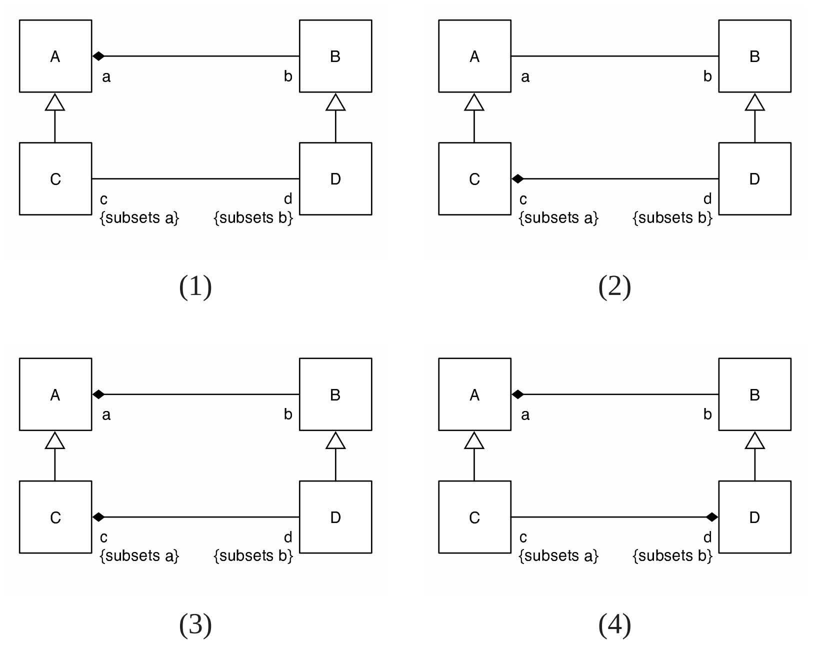


Figure 3.20: Subsetting with Composite and Noncomposite Properties

We can find examples of the three first cases in UML 2.0, all in Figure 11.5 on page 109 of [142]. Case (1) can be found in the association–memberEnd relation, case (2) is found in the owningAssociation–ownedEnd relation and case (3) in the class–ownedAttribute relation.

3.7 Alternative Language Extension Mechanisms

In this section, we briefly go through various additional language extension mechanisms. We also provide a motivation on why we do not include them in our framework.

3.7.1 Covariant Specialization

Covariant specialization is similar to subsetting in that it relates two relations at the metamodel layer. However the semantics are different. In a covariant environment, the specialized relation cannot be modified for elements which are instances of the subclasses.

As an example of covariant specialization, let us assume that e_c is an element of type C shown in Fig. 3.21. It is not possible to insert elements of type B into the b slot of e_c, only elements of type D into the d slot. The c–d relation is a *covariant specialization* of the a–b relation. The a–b relation has been rendered obsolete (or at least read-only) in the context of element e_c.

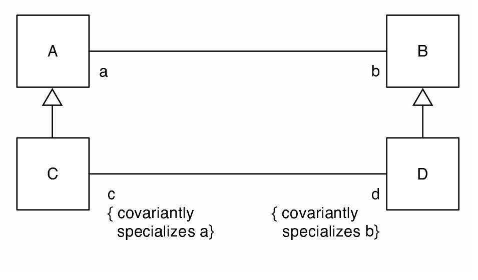


Figure 3.21: Notation for Covariant Specialization

Covariance is a subject that often comes up in the semantics of methods of OO programming. Function parameter type contravariance and return type covariance are rather inconvenient in practical situations and thus a type-unsafe function parameter type covariance is used for specialization. A similar argument also holds for element slots in modeling technology. Property subsetting aims to provide a new way to represent relationships between elements. It must nevertheless be noted that as Giuseppe Castagna has asserted, there are uses for a covariant environment when compared with a contravariant or invariant environment. Thus subsetting and covariance are not opposing but complementing constructs in OO programming and thus in modeling [32].

The major difference between covariance and subsetting is that in a covariant environment, substituting an element of a specific type with an element which is a covariant specialization of that type can result in programs no longer working. Thereby covariant specialization breaks Liskov substitutability and that is the reason we do not include it in our framework, although we realize it is an important concept. On the other hand, subsetting allows the slots defined by the properties in a superclass to be used in an instance of a subclass.

We note that contrary to subsetting, it might be necessary for the metamodel developer to explicitly declare the possibility of covariant specialization, instead of leaving the decision open for the future. This is because metamodel users need to realize that slots conforming to covariantly specialized properties may not be available in the future when their algorithm or function receives instances of the subclasses. That is, in the example it might be necessary to state a priori that the *a*–*b* relation can be covariantly specialized, as a warning mechanism for users and tools. Subsetting can be declared when required, whereas covariant specialization must be planned beforehand.

3.7.2 Property Redefinition

MOF 2.0 introduces the concept of property redefinition. It is our understanding that a property redefinition is an arbitrary replacement of the characteristics of a property in a subclass that overrides the subsetted property and renders it unusable in the subclass.

We can formalize this concept in our framework by introducing a new property characteristic redefines : $P \rightarrow \mathcal{P}(P)$, and by defining the set of effective properties of a class as follows:

$$\begin{aligned}\text{effectiveProperties}(c) = & \bigcup\{\text{properties}(s) \cdot c \subseteq_c s\} \\ & \setminus \bigcup\{\text{redefines}(p) \cdot p \in \text{properties}(s) \wedge c \subseteq_c s\}\end{aligned}$$

Usually, the redefining property has the same name as the redefined property, and exists in a subclass of the class of the redefined property. In data modeling terms, this means that programs will obtain the redefining property when they use an element of the subclass type.

However, we should note that there are no constraints between a property and its redefinition. A redefinition could be covariant or contravariant in some characteristics, and otherwise compatible or incompatible in others with respect to its redefined property. Using property redefinitions in a language extension breaks Liskov substitutability and therefore transformations and tools based on the original language. Therefore, we consider that property redefinition is not a safe construct and it should not be included in a metamodeling language.

3.8 Related Work

Several other researchers have formalized metamodeling languages and model layers. For example, Thomas Baar has defined the CINV language [10] using a set-theoretic approach, but our approach is more general in that we also support generalizations. The benefits of a set-theoretic approach is that it avoids a metacircularity whereby one (partially) needs to understand the language to be able to learn the language. José Álvarez, Andy Evans and Paul Sammut describe such a static OO metacircular modeling language [7], and the Metamodeling Language Calculus [42] by Tony Clark, Andy Evans and Stuart Kent is another very sophisticated one. Jan Nytun, Andreas Prinz and Andreas Kunert present in [122] a modeling framework in which all model layers are represented uniformly.

Akehurst, Kent and Patrascoiu present in [46] the structure of a metamodel and its semantics using OCL. Our rationale for not using OCL to define the model and metamodel constraints is that the definition of the navigation in OCL expressions actually depends on the metamodeling framework.

More recently, Frédéric Jouault and Jean Bézivin have presented KM3 [84], a metamodeling language targeted towards domain-specific modeling languages. This is one of the most influential works for this chapter since the notion of model conformance presented here is based on it.

However, the main contribution of this chapter comes from the definitions of property subsets in a language with multiple inheritance, which neither metamodeling nor traditional OO language descriptions explain. Several authors use relation inheritance without defining exact semantics, and some say that it denotes covariance. An example of this covariant specialization is the multilevel metamodeling technique called VPM by Varro and Pataricza [183], which also limits itself to single inheritance. We argue that property subsetting is not the same concept as covariant specialization, and requires different semantics. However, both subsetting and covariance specialization have their uses, and are thus complementing rather than competing constructs.

Carsten Amelunxen, Tobias Rötschke and Andy Schürr are the authors of the MOFLON tool [8] inside the Fujaba framework [121]. MOFLON claims to support subsetting, but no description of the formal semantics being used is included. It is not clear if their tool works in the context of subsets between ordered slots, or with diamond inheritance with subsetting. Markus Scheidgen presents an interesting discussion of the semantics of subsets in the context of creating an implementation of MOF 2.0 in [162]. To our knowledge, this has so far been the most thorough attempt to formalize subsetting.

The OO and database research communities are also researching a similar topic, although it is called relationship or association inheritance, or first-class relationships. In [23], Bierman and Wren present a simplified Java language with first-class relationships. In contrast with our work, they do not support multiple inheritance, bidirectionality or ordered properties; all of these constructs are common in modeling and in the UML 2.0 specification. However, relationship links are explicitly represented as instances, and they can have additional data fields (just like the AssociationClass of UML). As the authors have noticed, the semantics of link insertion and deletion is not without problems. Albano, Ghelli and Orsini present a relationship mechanism for a strongly-typed OO database programming language [6]. It also handles links as relationship instances, but without additional data fields. Multiple inheritance is supported, but ordered slot contents are not.

Finally, we should note there is an important ongoing discussion on the conceptual role of metamodeling and metamodeling languages in articles such as [63, 9]. These works describe the conceptual relationship between different metamodeling levels or layers. Our work focuses on the concrete constraints between two specific levels and it clearly exhibits the two metadimensions described in [9], where every model element *logically conforms* to a given metamodel class while it is *physically represented* as an element.

3.9 Conclusions

In this chapter we have explored the main concepts used in a metamodeling approach that supports class specialization and property subsetting. We have achieved this by building the metamodeling framework from the ground up using successive set-theoretic definitions of the structural semantics. Each definition adds a concept to our modeling framework: multiplicities, bidirectionality, ordering, composition, class specialization, subsetting, unions and strict unions.

We have also briefly discussed other language extension mechanisms. We have argued that covariant specializations make classes nonsubstitutable, that arbitrary property redefinitions are not a safe extension mechanism and that package merges do not provide anything fundamentally new as they can be described in terms of previous mechanisms. Therefore we have not included these concepts in our approach.

The contribution of this chapter is important because it emphasizes the need for all new metamodel language concepts to form an integrated whole and because it defines the new property characteristics of subsets and unions from MOF 2.0. We realize that all new metamodeling constructs interact with all the old metamodeling constructs. We have to ensure that the semantics of all combinations of these constructs make sense by declaring suitable metamodel and model constraints.

The OMG modeling standards do not describe subset and union properties in detail, not even informally, and therefore they cannot be applied in practice. In this chapter, we have formalized a simple modeling framework that supports subsets and derived unions. It discusses the relevant model constraints that must be upheld by any valid model. We will explore this area further in the next chapter, where we define operations that modify models while keeping the model constraints.

There is a limitation in the work presented in this chapter. The framework and especially subsetting as proposed is restricted to slots with unique elements. Slots where the same element can occur several times (bags, multisets) are not considered. Although bags can be defined in MOF 2.0, they are not used in the definition of the UML 2.0 Superstructure. Supporting ordered and unordered bags is not a fundamental problem when considering the static constraints, but is rather an issue of more work. However, supporting model operations on ordered bags with subsetting is problematic, as will be shown in Chapter 4. It must also be stressed that we do not cover several important aspects of MOF 2.0, such as association end ownership or navigability. These aspects create no important limitations from a formal point of view.

We have implemented the metamodeling language defined in this chapter in our modeling tool Coral, which will be discussed further in Chapter 5. Unfortunately, we know of no modeling tools that support subsets as extensively as discussed in this chapter. At the time of writing, the Eclipse EMF model repository does not implement subset properties, although the feature is planned. It is not clear what the semantics will be, though.

In conclusion, we consider that there is a need in the modeling community to standardize on one intuitive explanation and a rigorous formalization of subset properties and derived unions, so tools based on MOF 2.0 and UML 2.0 can be implemented and be interoperable. This chapter presents a proposal in the direction that we hope can help other researchers and tool developers to define a common understanding for MOF 2.0 and UML 2.0.

Chapter 4

Model Operations

4.1 Introduction

In the previous chapter we described the static structure of metamodels and models, and how models conform to their metamodel. From a conceptual perspective, this might be sufficient, in that we can give different model structures as input to a conformance testing function to see whether or not they are valid. However, from a tool development perspective, we are interested in modifying said conformant model structures in some particular manner while retaining conformance. In this chapter we will provide these operations.

These modifications are of different complexity: an interactive model editor requires small, incremental changes, whereas a transformation engine can create a whole model. Any larger change can in any case be split into a sequence of smaller changes, which we will call *basic edit operations*.

We have identified the following basic edit operations on the model structure *M* described in Chapter 3:

- Element creation.
- Element deletion.
- Inserting an element into a slot.
- Removing an element from a slot.

We need to consider whether these operations preserve the constraints defined in the previous chapter. We should note that a valid model transformation usually involves a sequence of many operations, of which some particular one cannot preserve for example the multiplicity constraint. Therefore, we always consider a model transformation as a sequence of basic edit operations. As an example, let us assume that we want to create a Generalization *G* between two classes *C1* and *C2* in a model based on UML. This requires three basic operations: create *G*, insert *C1* in the parent slot of *G* and insert *C2* in the child slot of *G*, assuming that the slots *C1*.specialization and *C2*.generalization are implicitly modified due to bidirectionality. The element *G* is invalid just after the create operation since a Generalization should always be connected to exactly one super- and one subclass. However, the model should be wellformed after executing all the basic operations.

This chapter is based on Publication VIII. We proceed as follows: in the next section we give pre- and postconditions and implementations for element creation and deletion. In Section 4.3 we will define unidirectional slot operations using a pre- and postcondition specification, as well as provide an implementation for each operation. For ease of understandability, we split the case of inserting an element into a slot into two separate operations, one for inserting into an unordered slot, and the other for inserting into an ordered slot. In Section 4.4 we extend these operations with bidirectional ones. We discuss validation of the research, related work and future work in Section 4.5. We finally conclude in Section 4.6.

4.2 Element Creation and Deletion

The operation create : $\mathcal{M} \times C \to \mathcal{M} \times E$ such that $(M', e) = \text{create}(M, c)$ creates a new element of type $c \in C$ and has no preconditions. The new element will also be a root element, i.e., it will not have any parent. The returned value is a tuple of the new models and the new element. In any pre- or postcondition, the old models are denoted $M = (E, \text{type}, \text{slots}, S, \text{property}, \text{elements})$. In postconditions, the new values of variables are denoted with tick marks, otherwise the old values before the execution is assumed. Thus, the new models are denoted $M' = (E', \text{type}', \text{slots}', S', \text{property}', \text{elements}')$. The primary postcondition is that there must be exactly one new element in the set of elements. The various model constraints mean that the sets and functions in M must be updated in M' to reflect this change; this leads to more postconditions.

1. $(\exists! e \in E' \cdot E' \setminus \{e\} = E \wedge \text{type}'(e) = c)$
2. $\text{type}' \cap \text{type} = \text{type}$
3. $\#S' = \#S + \#\{p \cdot p \in P \wedge (\exists! e \in E' \setminus E \wedge \text{type}(e) \subseteq_c \text{owner}(p))\}$
4. $S' \cap S = S$
5. $\text{slots}' = \text{slots} \cup \{e \to s \cdot e \in E' \setminus E \wedge s \in S' \setminus S\}$
6. $\text{property}' = \text{property} \cup \{s \to p \cdot s \in S' \setminus S \wedge p \in P \wedge \{\exists! e \in E' \setminus E \cdot \text{type}(e) \subseteq_c \text{owner}(p)\}\}$
7. $\#\text{Range}(\text{property}' \setminus \text{property}) = \#\{p \cdot p \in P \wedge (\exists! e \in E' \setminus E \wedge \text{type}(e) \subseteq_c \text{owner}(p))\}$

8. $\text{elements}' = \text{elements} \cup \{s \to \{\,\} \cdot s \in S' \setminus S \wedge \neg\text{ordered}(\text{property}'(s))\}$
$\cup \{s \to [\,] \cdot s \in S' \setminus S \wedge \text{ordered}(\text{property}'(s))\}$

The only relevant postcondition is the first one, the rest are implicit or informally understandable from the various model constraints. To avoid too much repetition, we assume that the new values of any variables not mentioned are kept identical to their previous values and that only the necessary changes to fulfill the postconditions are made. We will refrain from listing obvious postconditions and concentrate on the important ones.

Element creation can be defined by inserting a new element into the set E and correctly updating the various functions that comprise the models. The operation is shown in Figure 4.1. We assume there is a programming language-dependent way to create a new element and new slots.

```
create$(M,c)$ :=
    $M = E, \text{type}, \text{slots}, S, \text{property}, \text{elements}$
    Create an element $e$.
    $\text{type}' := \text{type}[e \to c]$
    $E' := E \cup \{e\}$
    $\text{property}' := \text{property}$
            $\cup$ { create a slot $s$ and return $s \to p \cdot p \in P \wedge \text{owner}(p) \subseteq_c c\}$
    $S' := S \cup (\text{Dom}(\text{property}') \setminus \text{Dom}(\text{property}))$
    $\text{slots}' := \text{slots} \cup \{e \to s \cdot s \in S' \setminus S\}$
    $\text{elements}' := \text{elements} \cup \{s \to [\,] \cdot s \in S' \setminus S \wedge \text{ordered}(\text{property}'(s))\}$
            $\cup \{s \to \{\,\} \cdot s \in S' \setminus S \wedge \neg\text{ordered}(\text{property}'(s))\}$
    return $((E', \text{type}', \text{slots}', S', \text{property}', \text{elements}'), e)$
```

Figure 4.1: Implementation of Creating a New Model Element

The operation $\text{delete} : \mathcal{M} \times E \to \mathcal{M}$ deletes an element. We require the element being deleted to have no connections to other elements via its slots. Therefore the precondition for deleting an element e is:

1. $(\forall s \in \text{slots}(e) \cdot \#s = 0)$

The postcondition is that the element must no longer be in the set of elements:

1. $E' = E \setminus \{e\}$

The implementation for element deletion is given in Figure 4.2.

4.3 Unidirectional Edit Operations on Models

We begin by describing in detail how to unidirectionally insert to or remove an element from a slot. These operations are the basic edit operations for models that are necessary to implement a model repository and a model transformation system.

```
$\text{delete}(M,d) :=$
    $M = E, \text{type}, \text{slots}, S, \text{property}, \text{elements}$
    $\text{type}' := \text{type} \setminus \{e \to c \cdot e = d \wedge e \to c \in \text{type}\}$
    $E' := E \setminus \{d\}$
    $\text{property}' := \text{property} \setminus \{s \to p \cdot \text{slotowner}(s) = d \wedge s \to p \in \text{property}\}$
    $S' := S \setminus (\text{Dom}(\text{property}) \setminus \text{Dom}(\text{property}'))$
    $\text{slots}' := \text{slots} \setminus \{e \to s \cdot e = d \wedge e \to s \in S\}$
    $\text{elements}' := S' \lhd \text{elements}$
    return $(E', \text{type}', \text{slots}', S', \text{property}', \text{elements}')$
```

Figure 4.2: Implementation of Deleting a Model Element

First, we describe the case of insertion into ordered or unordered slots and then the case of removing elements from slots. The pre- and postconditions are described as separate enumerated clauses. All of the clauses in the precondition must hold for the operation to succeed, and all the clauses of the postcondition must be guaranteed by an implementation.

Unfortunately, the semantics provided by this chapter have one caveat. We assume that the ordering characteristic of a property must be the same in its subset and/or superset properties. That is, the following additional metamodel constraint must hold:

Metamodel Constraint 11 *Ordering characteristics are same in property poset:* $(\forall p \in P \cdot (\forall q \in \text{supersets}(p) \cdot \text{ordered}(q) = \text{ordered}(p))$

We note that a slot and its transitive subset and superslot slots form a poset with respect to the $\subseteq_s$ relation, as has been explained in Chapter 3. Then these slots can be drawn as a Hasse diagram.

4.3.1 Element Insertion into an Unordered Slot

Consider an operation $\text{insert}^u : \mathcal{M} \times S \times E \to \mathcal{M}$ such that $\text{insert}^u(M,s,e)$ inserts element e into slot s. The intuition behind the insertion operation is that all supersets of s must contain the new element e for the ISR constraints to hold. The clauses for the precondition for element insertion into an unordered slot are thus:

1. $\neg\text{strictUnion}(\text{property}(s))$
2. $\neg\text{ordered}(\text{property}(s))$
3. $e \notin \text{elements}(s)$.
4. $\text{type}(e) \subseteq \text{owner}(\text{opposite}(\text{property}(s)))$
5. $(\exists t \in S \cdot s \subseteq t \wedge \text{composite}(\text{property}(t)) \Rightarrow \text{parent}(e) \setminus \{\text{slotowner}(t)\} = \emptyset)$

6. $\mathrm{composite}(\mathrm{property}(s)) \Rightarrow e \notin \mathrm{parentchain}(\mathrm{slotowner}(s))$

The clauses state that (1) we are not modifying a derived read-only slot, (2) the slot is unordered, (3) the element must not yet exist in the slot, (4) that we obey the rules of strong typing, (5) we do not create a connection to a second parent for *e* and (6) we do not create a circular composition.

The postcondition for element insertion is simple. We wish element *e* to be found in the slot *s* and all its transitive supersets. All the model constraints except for the multiplicity constraints must also hold as a postcondition.

1. $(\forall t \in S \cdot s \subseteq t \Rightarrow \mathrm{elements}'(t) = \mathrm{elements}(t) \cup \{e\})$ (Note $s \subseteq s$)

An example of element insertion into an unordered slot can be seen in Figure 4.3. In case (1) of the figure, we have a poset of unordered slots. Suppose we insert an element *c* into slot *q*. This requires an insertion of *c* into slots *p* and *r* as well, to preserve the ISR, with the end result shown in case (2). After this, inserting *c* into slot *t* also inserts it into slot *s*, again to preserve the ISR, resulting in case (3). Slots *p*, *q* and *r* are not modified because *c* already existed in those slots.

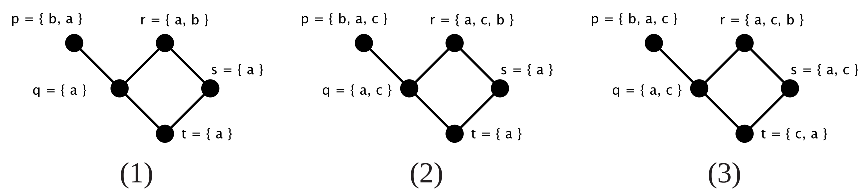


Figure 4.3: Example of Inserting an Element into Unordered Slots

It can be noted that in our semantics, an insertion into a slot never modifies any subset of that slot.

The implementation of element insertion into an unordered slot is given in Figure 4.4.

```
$\mathrm{insert}^u(M, s, e) :=$
    $M = (E, \mathrm{type}, \mathrm{slots}, S, \mathrm{property}, \mathrm{elements})$
    $\mathrm{elements}' := \mathrm{elements}[\{x \rightarrow \mathrm{elements}(x) \cup \{e\} \cdot x \in S \wedge s \subseteq x\}]$
    $\mathrm{return}\ (E, \mathrm{type}, \mathrm{slots}, S, \mathrm{property}, \mathrm{elements}')$
```

Figure 4.4: Implementation of Element Insertion into an Unordered Slot

4.3.2 Element Insertion into an Ordered Slot

Subsetting with ordered slots is more complicated than with unordered slots, due to the need to preserve an order between the elements in different slots. We define the operation $\text{insert}^o : \mathcal{M} \times S \times E \times \mathbb{Z}^{0+} \to \mathcal{M}$ such that $\text{insert}^o(M,s,e,i)$ inserts an element e into a slot s at index i.

We assume there is a function $\text{index} : E \times S \to \mathbb{Z}^{0+}$ which returns the zero-based index of an element in the contents of an ordered slot. A function lower_index : $\mathbb{Z}^{0+} \times S \times S \to \mathbb{Z}^{0+}$ is such that $\text{lower\_index}(i,x,y)$ returns the index in x where $y[i]$ should be inserted to preserve the subset $x \subseteq y$. It is shown in Figure 4.5 and is used to calculate which restrictions from supersets apply to subsets when inserting an element. As an example, consider what the restriction given by element c (at index position 2) in the superset $[a,b,c,d]$ is to its subset $[a,d]$. Then $\text{lower\_index}(2,[a,d],[a,b,c,d])$ returns 1 since c should be inserted between a and d.

```
lower_index$(i,s,t)$ :=
  if $t[i] \in s$ then return index$(t[i],s)$
  do
    if $t[i] \in s$ then return index$(t[i],s)+1$
     else if $i = 0$ then return 0
     else $i := i-1$
  od
```

Figure 4.5: The lower_index Function

A function lift_interval : $S \times S \times R \to R$, where R denotes integer intervals, is such that $\text{lift\_interval}(s,t,[v..w])$ "lifts" the interval $[v..w]$ from s as superimposed on t (when $s \subseteq t$). It is shown in Figure 4.6 and is used to calculate which restrictions from subsets apply to supersets and works as the dual of lower_index. As an example, consider the ordered sets $s = [c]$ and $t = [b,c]$. If we were to insert element a at index 0 in s, the corresponding interval for s would be $[0..0]$. This interval is superimposed onto t as the interval $[0..1]$, meaning that the same element can be inserted either before or after b in t without violating the ISR. Thus, $\text{lift\_interval}(s,t,[0..0]) = [0..1]$.

```
lift_interval$(s,t,[v..w])$ :=
  if $v > 0$ then $v' :=$ index$(s[v-1],t)+1$
     else $v' := 0$
  if $w = \#s$ then $w' := \#t$
     else $w' :=$ index$(s[w],t)$
  return $[v'..w']$
```

Figure 4.6: The lift_interval Function

The function $\text{indices\_ok} : \mathcal{P}(S) \times (S \to R) \to \mathbb{B}$ returns true if when executing $\text{indices\_ok}(T,F)$ there is a possible way to insert an element into every slot in T

such that the constraints in F are satisfied. Here, $F : S \to R$ is a map from slots to integer intervals $[v..w]$ such that $v \leq w$ where e can be inserted. The function is shown in Figure 4.7. Using the lift_interval and lower_index functions we restrict the possible intervals where e can be inserted into the slots.

$$\text{indices\_ok}(\emptyset, F) := (\forall t \in \text{Dom}(F) \cdot F(t) \neq \emptyset)$$

$$\begin{aligned}&\text{indices\_ok}(T, F) :=\\ &\quad (\exists t \in T \cdot (\forall u \in T \cdot t \not\supset u)\\ &\qquad \wedge R \overset{\text{def}}{=} \cap\{\text{lift\_interval}(c, t, [v..w]) \cdot (\forall c \cdot s \subseteq c \prec t \wedge F(c) = [v..w])\}\\ &\qquad \Rightarrow \text{indices\_ok}(T \setminus \{t\}, F[t \to R \cap F(t)]))\end{aligned}$$

Figure 4.7: The indices_ok Function

The precondition is otherwise identical to the case when inserting into an unordered slot, except for the check for an ordered slot. and that there exists an extra clause which calculates if the insertion into the slot and its transitive supersets is at all possible without violating the ISR.

1. $\neg\text{strictUnion}(\text{property}(s))$
2. $\text{ordered}(\text{property}(s))$
3. $e \notin \text{elements}(s)$
4. $\text{type}(e) \subseteq \text{owner}(\text{opposite}(\text{property}(s)))$
5. $(\exists t \in S \cdot s \subseteq t \wedge \text{composite}(\text{property}(t)) \Rightarrow \text{parent}(e) \setminus \{\text{slotowner}(t)\} = \emptyset)$
6. $\text{composite}(\text{property}(s)) \Rightarrow e \notin \text{parentchain}(\text{slotowner}(s))$
7. $\text{indices\_ok}(\{t \cdot s \subset t\},$
   $\{s \to [i..i]\}$
   $\cup \{t \to [\text{lower\_index}(\text{index}(e,u),t,u)..\text{lower\_index}(\text{index}(e,u),t,u)] \cdot s \subset t \wedge t \subseteq u \wedge e \in \text{elements}(u)\}$
   $\cup \{t \to [0, \#t\ ] \cdot s \subset t \wedge \neg(\exists u \cdot t \subseteq u \wedge e \in \text{elements}(u))\}$
   $)$

The intuition behind the last clause in the precondition and the definition of the indices_ok function is that we calculate the range restrictions of e which exist in any super- or subsets onto the other slots. The F function is initially created by describing constraints from supersets. F is created from three different clauses. The first, $s \to [i..i]$, constrains e to be inserted at exactly index i. The second does similarly for supersets which have a superset that already has e, whereas the third initially allows all indices to be candidates for insertion. This initialization makes sure that F is restricted by the the elements e that already exist in any supersets of s. Note that any slot o such that $o \subset t \wedge s \subset t \wedge o \,||_s$ is outside of the transitive superset closure of s and any restrictions from it will already be visible in t and thus it is not necessary to include o in F.

Then, indices_ok calculates the constraints from subsets and does set intersection to calculate whether an insertion is possible. The actual function takes all supersets T and picks one $t \in T$ which is a bottom element, which must exist since the slots in T are part of a poset. It then imposes all intervals from subset slots c (such that $s \subseteq c \prec t$) onto t, also including the initial constraint on t. It then recurses with a modified F until T is empty.

We claim, without proof, that if the final mapping F contains only nonempty intervals, it is possible to successfully insert e into s at index i. The postcondition is:

1. $\text{elements}'(s)[i] = e$

2. $(\forall t \in S \cdot s \subseteq t \wedge e \notin \text{elements}(t) \Rightarrow \text{elements}'(t) \setminus \{e\} = \text{elements}(t)$
   $\wedge e \in \text{elements}'(t))$

The current definitions do not tell us the exact index where to insert e into any superslot of s, only that a combination of indices exists; an index i_t for a superslot t of s must exist somewhere in the range given by $F(t)$.

An example of element insertion can be seen in Figure 4.8. Case (1) is the initial configuration of the slots w, x, y and z. Let us assume an insertion of element c into slot w at index position 0 occurs. The returned slot ranges where c should be inserted raises the possibilities in cases (2) to (5), depending on whether c is inserted onto the left or right side of either a in slot y or b in slot z. Cases (2), (3) and (4) are correct solutions and our postcondition does not prefer any particular one over the another. Case (5) is not legal, because slot x cannot preserve the superset relationship as enforced by both slots y and z, as element c should occur both before a and after b in the ordered set. It is up to the implementation to choose one of the correct solutions, perhaps with guidance from the user.

As we have shown, insertion into an ordered slot is rather complicated. The precondition could only tell us whether or not there is at least one solution, not what the exact combination of indices in different slots should be for a particular solution. Naturally, we must avoid the combinations that do not preserve the ISR.

Hence, we believe that the notion of an *insertion strategy* is important. Depending on the effect the developer wishes to obtain, such a strategy will mechanically calculate a particular solution and execute the actual insertion operation. At the moment we use only one strategy, that of always using the last index position possible.

The context in which the insertion strategy has to work is the final function F when T has been exhausted, as can be seen in Figure 4.7.

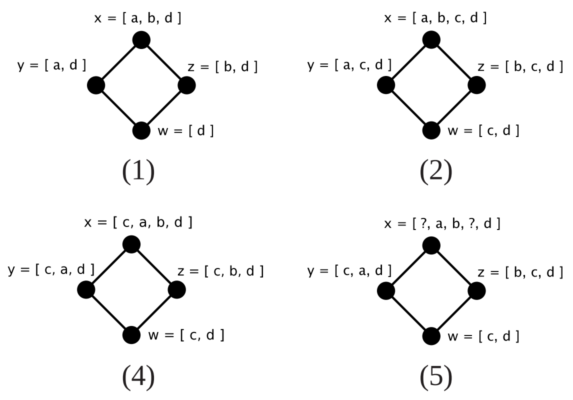


Figure 4.8: Example of Inserting an Element into Ordered Slots

Insertion Strategies

Our implementation assumes that a correct combination of indices occurs if we always choose the last index (i.e., w of $F(t) = [v..w]$ for a slot t). This has worked perfectly in our experiments. Given our assumption, the implementation in Figure 4.9 is simple.

```
$\text{insert}^o(M, s, e, i)$
    Calculate the final $F$ as in Figure 4.7.
    $M = (E, \text{type}, \text{slots}, S, \text{property}, \text{elements})$
    $\text{elements}' := \text{elements}[\{t \rightarrow \text{elements}(t)[0:w] \triangleleft [e] \triangleleft \text{elements}(t)[w:\#t]$
                                        $\cdot\ t \in S \wedge s \subseteq t \wedge e \notin \text{elements}(t) \wedge [v..w] = F(t)\}]$
    return $(E, \text{type}, \text{slots}, S, \text{property}, \text{elements}')$
```

Figure 4.9: Implementation of the Insert Operation for Ordered Sets, Using the Last Index Strategy

4.3.3 Element Removal from a Slot

The operation remove : $\mathcal{M} \times S \times E \rightarrow \mathcal{M}$ is defined such that $\text{remove}(M, s, e)$ removes the element e from s and all its subsets, as well as from those supersets which would not acquire e via some other subset which is not comparable to s. Element removal from an ordered slot is identical to element removal from an unordered slot since removing a specific element from an ordered slot does not alter the relative position of the other elements in the slot.

The precondition requires that a derived slot is not being modified and that the element must exist in the slot:

1. $\neg\text{strictUnion}(\text{property}(s))$
2. $e \in \text{elements}(s)$

The postcondition:

1. $(\forall r \in S \cdot r \subseteq s \Rightarrow \text{elements}(r) = \text{elements}'(r) \cup \{e\} \land e \notin \text{elements}'(r))$
2. $(\forall t \in S \cdot s \subset t \land \neg(\exists m \in S \cdot m \subset t \land m \,||_s\, \land e \in \text{elements}(m))$
   $\Rightarrow \text{elements}(t) = \text{elements}'(t) \cup \{e\} \land e \notin \text{elements}'(t))$

Both clauses in the postcondition are interesting. The first clause states that a removal from a slot triggers a removal from any subset, so that the ISR can hold. This can be contrasted with the insertion operation, which does not modify any subsets. The second clause states that a removal from a slot triggers a conditional removal from any superset. An interesting feature of the clause is shown in Figure 4.10. If we have an initial setting as in case (1) and remove *a* from *z*, the clause requires that *a* is removed from *x* as shown in case (2), although this is not necessary to preserve model consistency. However, we believe that this feature is the intended usage by the modeling standards. Inserting into a subset triggers insertion in all supersets, and so dually a removal from a subset ought to trigger a removal from all supersets. A similar chain of reasoning has been reported by Markus Scheidgen [162].

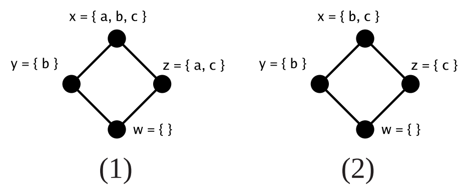


Figure 4.10: Removing *a* from an Unordered Slot *z*

As an example where the second clause is necessary, consider Figure 4.11 with the initial setting as in case (1).

Assume we wish to remove *a* from *y*. An incorrect approach is the removal of *a* from supersets and subsets would leave *x* without *a*, but *z* with *a* intact, violating the ISR, as shown in case (2). A correct option would be to remove *a* also from *z*, as shown in case (3), but our opinion is that this "snowball effect" of removing *a* reduces the usefulness of subsets; slot *y* should affect slot *z* as little as possible, since they are not comparable in the Hasse diagram. Our postcondition ensures that *a* must be removed from *w* and *y*, but not from *x*, because *z* still contains *a*; this is seen in case (4).

Another interesting case is the ISR rule for derived slots. If (and only if) *z* is marked as derived, we must remember that its elements must be found in the union of its subsets. In case (5), *a* is removed from *y* which leads to it being removed

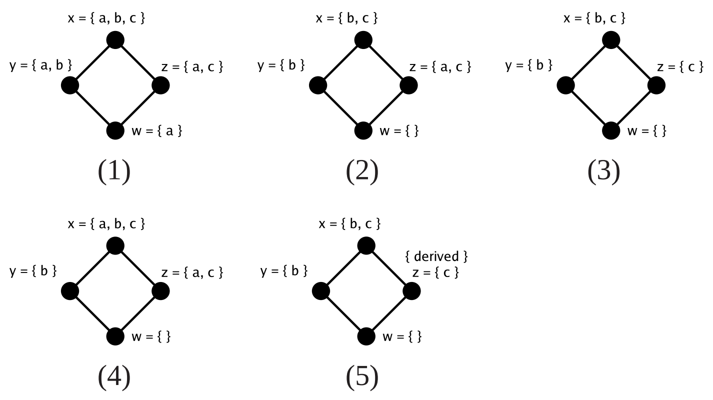


Figure 4.11: Different Scenarios for Removing *a* from an Unordered Slot *y*

from *w* as well. As *z* is marked as derived, *a* must also be removed from it, since *z* does not have any other subset containing *a*. This in turn leads to *a* being removed from *x*.

The implementation for the removal of an element for an ordered or unordered slot is shown in Figure 4.12.

```
$\text{remove}(M, s, e) :=$
    $M = (E, \text{type}, \text{slots}, S, \text{property}, \text{elements})$
    $\text{elements}' :=$
        $\text{elements}[\{t \to \text{elements}(t) \setminus \{e\} \cdot t \subseteq s\}$
                    $\cup \{t \to \text{elements}(t) \setminus \{e\} \cdot s \subset t$
                        $\wedge \neg(\exists m \cdot m \subset t \wedge m \,||\, s \wedge e \in \text{elements}(m))\}]$
    $\text{return } (E, \text{type}, \text{slots}, S, \text{property}, \text{elements}')$
```

Figure 4.12: Implementation of Element Removal from a Slot

4.4 Bidirectional Edit Operations on Models

In the previous two sections, we showed how to create and delete elements, and how to unidirectionally modify one slot and its transitive subset and superset slots. We emphasize the importance of unidirectionality; a consequence of this is that the operations did not modify any opposite slots. While the unidirectional operations are useful for low-level manipulation of models, they cannot be used directly in a high-level context, such as a model transformation, if we are to preserve model constraint 5, the bidirectionality constraint on slots, from the previous chapter. However, we can define another set of bidirectional slot operations based on the unidirectional ones.

We note that given a slot s such that $e \in \text{elements}(s)$, the opposite slot that is owned by element e is the unique slot:

$$\text{slotopposite}(e,s) \stackrel{\text{def}}{=} r,\ \text{such that slotowner}(r) = e \wedge \text{opposite}(\text{property}(s)) = \text{property}(r)$$

Due to the evident use of an index parameter for insertion into ordered slots and no parameter for insertion into unordered slots, we can conclude that a bidirectional operation on a slot s and element e such that $e \in \text{elements}(s)$ can be split into four cases, depending on whether they are ordered or not:

- $\neg\text{ordered}(s) \wedge \neg\text{ordered}(\text{slotopposite}(e,s))$
- $\neg\text{ordered}(s) \wedge \text{ordered}(\text{slotopposite}(e,s))$
- $\text{ordered}(s) \wedge \neg\text{ordered}(\text{slotopposite}(e,s))$
- $\text{ordered}(s) \wedge \text{ordered}(\text{slotopposite}(e,s))$

We create four functions that implement the above cases for inserting an element, and a single function that implements element removal. The preconditions of the following functions are the conjunction of the preconditions of the individual operations of which they consist. Similarly, the postconditions are the conjunction of the postconditions of the individual operations of which they consist. Thus there is no need to repeat them here.

4.4.1 Element Insertion into Unordered/Unordered Slots

The first function $\text{insert}^{uu} : \mathcal{M} \times S \times E \to \mathcal{M}$ describes insertion into an unordered slot with an unordered opposite slot. The implementation of it can be seen in Figure 4.13.

```
$\text{insert}^{uu}(M,s,e) :=$
    return $\text{insert}^{u}(\text{insert}^{u}(M,s,e), \text{slotopposite}(e,s), \text{slotowner}(s))$
```

Figure 4.13: Implementation of Element Insertion into Unordered/Unordered Slots

4.4.2 Element Insertion into Unordered/Ordered Slots

The function $\text{insert}^{uo} : \mathcal{M} \times S \times E \times \mathbb{Z}^{0+} \to \mathcal{M}$ describes insertion into an unordered slot with an ordered opposite slot. The implementation of it can be seen in Figure 4.14.

```
$\text{insert}^{uo}(M, s, e, i) :=$
    $\text{return insert}^{o}(\text{insert}^{u}(M, s, e), \text{slotopposite}(e, s), \text{slotowner}(s), i)$
```

Figure 4.14: Implementation of Element Insertion into Unordered/Ordered Slots

4.4.3 Element Insertion into Ordered/Unordered Slots

The function $\text{insert}^{ou} : \mathcal{M} \times S \times E \times \mathbb{Z}^{0+} \to \mathcal{M}$ describes insertion into an ordered slot with an unordered opposite slot. The implementation of it can be seen in Figure 4.15.

```
$\text{insert}^{ou}(M, s, e, i) :=$
    $\text{return insert}^{u}(\text{insert}^{o}(M, s, e, i), \text{slotopposite}(e, s), \text{slotowner}(s))$
```

Figure 4.15: Implementation of Element Insertion into Ordered/Unordered Slots

4.4.4 Element Insertion into Ordered/Ordered Slots

The last insertion function $\text{insert}^{oo} : \mathcal{M} \times S \times E \times \mathbb{Z}^{0+} \times \mathbb{Z}^{0+} \to \mathcal{M}$ describes insertion into an ordered slot with an ordered opposite slot. The implementation of it can be seen in Figure 4.16.

```
$\text{insert}^{oo}(M, s, e, i_1, i_2) :=$
    $\text{return insert}^{o}(\text{insert}^{o}(M, s, e, i_1), \text{slotopposite}(e, s), \text{slotowner}(s), i_2)$
```

Figure 4.16: Implementation of Element Insertion into Ordered/Ordered Slots

4.4.5 Element Removal from Slots

Bidirectional element removal can be described by the function $\text{remove}^{\star} : \mathcal{M} \times S \times E \to \mathcal{M}$ as given by the implementation in Figure 4.17. It works regardless of whether the slot or its opposite is ordered or unordered.

We note that the unidirectional operations do not preserve model constraint 5, i.e., bidirectionality, whereas the bidirectional operations preserve it. We claim that the other model constraints, except the multiplicity constraint, are preserved.

Then, only model constraint 4, that of restricting the number of elements in slots to the multiplicity limits of the property, can be violated by these operations. We do not preserve this constraint because of the atomicity of the operations. Consider a slot with a lower multiplicity greater than zero. When creating an element with such a slot, we should immediately populate the slot with enough elements

```
remove$^\star(M,s,e)$ :=
    return remove(remove$(M,s,e)$, slotopposite$(e,s)$, slotowner$(s)$)
```

Figure 4.17: Implementation of Bidirectional Element Removal

for the multiplicity constraint to hold. But this is not possible with the operations that we have described.

Although we could extend our operations further, there is a different solution. We can use a sequence of basic operations Our anecdotal experience is that a sufficient solution is to add a *transaction mechanism* into our modeling tool of choice.

Indeed, the implementations of the bidirectional operations are merely a sequence consisting of two unidirectional operations, and thus we could claim that a transaction mechanism is already necessary at this level. However, we feel that bidirectionality is such a fundamental part of manipulating and navigating models that the bidirectional operations that have been described in this section are the basic operations in a modeling framework, and more complicated operations should employ the aid of the transaction mechanism.

4.5 Related and Future Work

Apart from the related work as mentioned in Chapter 3, we are only aware of the work of Markus Scheidgen [162] on operations for subset properties. In his approach to formalizing model edit operations on subsets, a slot modification creates an *update graph* of slots, so that a later modification at some other slot in the update graph actually updates all the associated slots. The actual operational semantics are unfortunately not described in detail. In comparison, we do not have to create or maintain any update graphs. Furthermore, our contribution not only discusses but also defines pre- and postconditions and implementations for the operations for ordered and unordered sets. It is also not clear if the work by Scheidgen supports diamond subsets or ordered sets, both of which are used in e.g. the UML 2.0 Infrastructure. However, our semantics are different and it is not clear which formalization is better suited for modeling purposes.

As already mentioned in Chapter 3, the basic edit operations involving subsets have been implemented in the Coral tool. We have found no errors or inconsistencies in our implementation.

There are several different theoretical tasks for future work.The first and foremost task is to prove the correctness of the pre- and postconditions as well as the implementations. Second, throughout this chapter, we have been clear that the elements in a slot are an unordered or ordered set. It might be of interest to formalize the framework for *bagness*, whereby a slot may contain the same element several times. It is fairly straightforward to extend this framework for unordered bags, even with subsetting, but our initial experiments with ordered bags and subsetting have not been as successful.

Third, we might wish to incorporate covariant specialization into the framework, as so many authors are implicitly using covariance, and experience from the object-oriented community is that covariance is a useful concept.

Fourth, the MOF characteristic of redefinition is still fairly arbitrary, badly defined concept, and could certainly take advantage of a more rigorous definition.

Furthermore, we have not discussed metamodel evolution, where properties and classes are redefined, and how models must be updated accordingly, especially with respect to subsetting. We have followed a basic assumption that metamodels are static, but we should note that there are object-oriented frameworks where class updates or redefinitions are possible, for example Common Lisp [48].

4.6 Conclusions

There are several new property characteristics described in MOF 2.0: subsets, (derived) unions and redefinitions. However, these standards do not describe these concepts in detail, not even informally, and therefore cannot be applied in practice. In this chapter, we have presented basic operations for element creation and deletion and slot modification, taking into account subsets and derived unions. We have given usage examples where subsetting provides a new, fundamental approach to language extension. Several authors have used property subsetting informally, almost always referring to covariant specialization. Formalization of covariance leads to a different result than the one presented in this chapter. Both subsetting and covariance specialization have their uses, however, and are thus complementing rather than competing constructs.

There are limitations in the work presented in this chapter, except for the ones already discussed in the previous chapter. It would be useful if the precondition on checking indices related to ordered insertion into a slot could be removed. We also assume that a subset property should have the same ordering characteristic as its union property. However, we notice that it is also possible to mix these characteristics such that an ordered slot may be a subset of an unordered slot. The extension is trivial since it weakens the precondition because we do not need to preserve any indices in the unordered slot. The UML 2.0 Infrastructure uses this in the association–memberEnd relation in Figure 11.5 of [142].

However, the opposite case where an unordered property subsets an ordered property is problematic. Insertion into an ordered slot requires an index, but the initial insertion into the unordered slot does not tell which index or indices to use in any supersets which are ordered. We do not see any benefits in pursuing semantics for this construct. However, it must be noted that Figure 11.7 of [142] does show an example where an ordered property is subset by an unordered one. We believe this example, to be erroneous. We have not found such a usage pattern in UML 2.0.

Furthermore, ordered bags are not considered in our formalization. The reason for this can be shown with an example. Consider a slot $[a]$ subsetting another slot $[a, b, a]$. Inserting an element into the subset slot is problematic, because we have to match element a to one of the a in the superset slot, but it is not possible to deduce which one.

As we have shown, the formalization presented in this chapter can be implemented in a straightforward manner in a model repository. We plan to use our new definitions in implementing the UML 2.0 metamodel with diagram editors in Coral. In doing so, we strive to acquire experience in using subsets and derived unions in large models.

In conclusion, we consider that this work is important because there is an imminent need in the modeling community to standardize on one formalization of subsets and derived unions, so that tools implementing MOF 2.0 and UML 2.0 can be interoperable. The semantics described in this chapter is one proposal and we hope it spurs further interest and discussion. Furthermore, the idea of subsetting is intriguing, since it is a new construct for modeling relationships between classes and objects, and thereby brings a novel idea to the modeling and object-oriented community.

Chapter 5

Implementation of the Simple Metamodel Description Language

5.1 Introduction

In this chapter we present the definition and implementation of a modeling framework called Coral. Coral is based on a simple metamodeling language that supports the concepts defined in Chapter 3 and some primitive datatypes.

We call the metamodeling language the Simple Metamodel Description Language (SMD). We explain it and show how metamodels and models are described internally in Coral. We also show how metamodel files described using the SMD language are transformed into metamodels inside the framework.

The work in this chapter is important because it allows us to validate our ideas in practice. Even though it is crucial to have a wellfounded theoretical basis for the metamodeling language, as discussed in the above-mentioned chapters, it is far too easy to overlook details without a working prototype. For example, we notice that supporting serialization of models affects SMD and its implementation.

This chapter is based on experiences from all our publications. The Coral tool has been presented in Publication V. We proceed as follows. We describe the SMD language in Section 5.2 In Section 5.3 we discuss the implementation of models and metamodels in the framework, along with relevant known limitations. We discuss related work in Section 5.4 and conclude in Section 5.5.

5.2 The SMD Language

The Simple Metamodel Description language (SMD) is our modeling language to define models that define modeling languages. In other words, it is a metamodeling language or a metametamodel. SMD is analogous to MOF, but there are several reasons why we have chosen to use our own language instead of MOF. Primarily we have wanted to explore which concepts are really required from a metamodeling language, and as a result SMD is based on fewer concepts than MOF and therefore it is easier to implement and understand. Additionally we are interested in experimenting with new metamodeling constructs or new semantics for old constructs. Our experiences in implementing and using a working tool led to the formalization of SMD as described in Chapter 3. It can be seen in Figure 5.1.

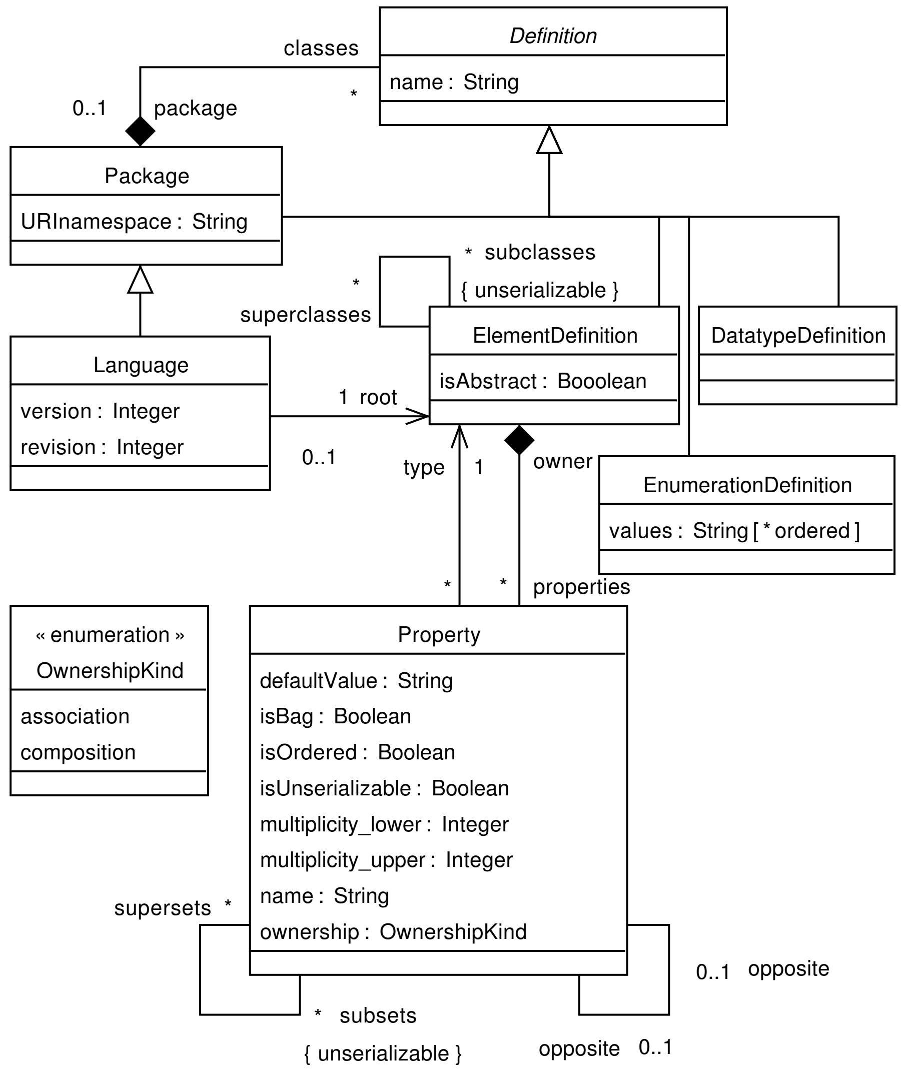


Figure 5.1: The SMD Language

It must be noted that although SMD has been developed independently from ECORE, it still is very similar to it, and some features of ECORE were added to SMD to support ECORE models. SMD consists of a Language metaclass which describes a modeling language. It has a name, URInamespace, version and revision. The latter three are intended for humans, e.g., the tuple $(\mathrm{UML}, 1, 4)$ describes the UML 1.4 language. However, URInamespace is the XML namespace declaration for serializing any models written in a specific modeling language. This concept will be explained more extensively in Chapter 6. The root property describes which class should be instantiated to create a rudimentary model of the modeling language. For example, the UML 1.4 language has the Model class as its root.

Package instances are otherwise like Language instances but are assumed to be a owned by a Language instance, since they lack the version and revision information. A hierarchy of Package and other Definition elements can be created with the Package.classes property. It can be noted that neither Language nor Package are part of the formalization from Chapter 3. They are merely convenient containers of the other definitions.

Various DatatypeDefinition elements are defined but cannot be created by the metamodel designer. They are hardcoded primitive types in the SMD language. New EnumerationDefinition classes can be created to define new enumerations. For example, a UML 1.4 Attribute has a VisibilityKind enumeration which defines the values public, protected, private and package. The first value is assumed to be the default value, unless Property.defaultValue overrides it. A limitation of SMD when compared with the formalization in Chapter 3 is that primitive types cannot have properties. This is a very common limitation in metamodeling frameworks.

ElementDefinition is used to define new classes in a modeling language. Its most important concepts are properties and superclasses. A Property defines one endpoint of a relation. It can be connected to some other Property element or to itself; this later part is not allowed in the formalization from Chapter 3, but it is for convenience. It defines the various property characteristics as given in Section 3.4 and Section 3.6. It also adds the Property.isUnserializable characteristic; slots of a property with the isUnserializable characteristic set to true are not serialized. This is very useful for some transient modeling data which is not persistent. For example, references to subclasses and subset properties from the superclass and superset property, respectively, should not be saved since we wish modeling languages to be independent of their extensions.

We note that akin to MOF 2.0 and EMF, SMD also supports only two kinds of ownerships: association and composition. Multiplicity restrictions can only be given by a single range, with an upper multiplicity of -1 denoting infinity. An interesting detail is that during the implementation of the Coral core, Package.URInamespace was renamed from Package.xmlnamespace, due to an XML restriction which is explained in Section 6.3.2.

As our experience in metamodeling grew, we added more features deemed necessary to SMD. The default values for primitive obligatory properties, i.e., those with a multiplicity of 1..1, are an example where a feature was copied for compatibility reasons from EMF. Property subsetting is a feature that has hitherto been unavailable in most (if all) other modeling frameworks.

The SMD metametamodel can be described using itself. As presented, there is an interesting problem with it. It uses primitive types like integers and strings. These can be seen in Figure 5.2. The most important element is the AnyElement class, which is the superclass of any other class. Its role is to serve as the toplevel definition in any language hierarchy, similar to java.lang.Object in the Java programming language [64].

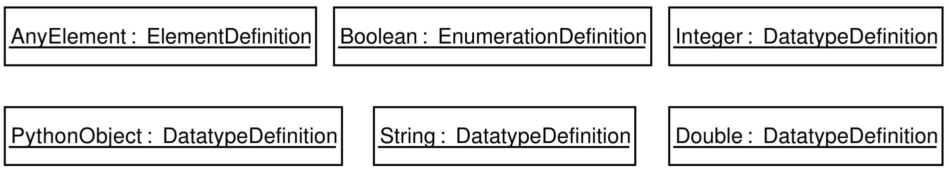

AnyElement : ElementDefinition
Boolean : EnumerationDefinition
Integer : DatatypeDefinition
PythonObject : DatatypeDefinition
String : DatatypeDefinition
Double : DatatypeDefinition

Figure 5.2: The Primitives of the SMD Language

This presents an interesting conundrum, as AnyElement has a type of ElementDefinition, and the superclass of ElementDefinition is (implicitly) AnyElement. In addition, this definition crosses metalevel boundaries. However, this is merely an artefact of the metacircularity and does not effectively present any problems. An implementation must bootstrap the SMD language in some implementation-specific way.

There is also a set of predefined primitive types: integer, string, double floating-point, the boolean enumeration type and PythonObject. They can be used when an entity cannot be broken into smaller constituent parts, or when such primitive values are a natural way to express their contents. The Python [180] object type enables us to attach any arbitrary Python object such as graphical widgets to a model element and it is usually used in unserializable Slots. The enumeration named Boolean has the values *false* and *true*, and is predefined for the sole reason that it is so frequently necessary.

5.3 Implementation

As stated previously, our implementation of SMD is a modeling framework called Coral. It has been developed by our research group and can be seen as an implementation and validation of the ideas presented in this thesis. Its current version consists of over 23 000 lines of C++ [170] source code with bindings for the Python [180] language.

The SMD language defined in Figure 5.1 requires the concepts of classes, enumerations and primitive types itself. This is the bootstrapping problem that all implementations of metamodeling tools need to cope with in one way or another. Therefore, although the SMD language can and is defined as an SMD metamodel, it cannot be loaded into a metamodeling tool based on SMD in order to define

SMD. Rather, it must be bootstrapped in-place. The details of this is not important as it is dependent on too many internal implementation artifacts in a metamodeling tool.

Even though SMD cannot be loaded from a file in order to define itself, other languages can be defined as SMD models. Thus, the Coral core is an implementation of SMD itself, which allows us to load additional metamodels defined as SMD models. Additionally, there is an interface for creating models based on these metamodels.

5.3.1 The Metamodel and Model Layers

There are different ways to realize a metamodeling tool, and thus there are several ways to represent metamodels and models internally in a modeling tool. Some tools present each ElementDefinition in a modeling language as a class in the programming language used by the tool and each element in a model as an instance of such a class. For example, such an implementation of the UML metamodel in Java would have a Java class named State that will represent the definition of a UML 1.4 State. New elements in a model are created by instantiating the Java classes.

There are several problems with this approach. The most obvious is that it is not possible to define new modeling languages dynamically without compiling or loading new source code, i.e., the compilation environment must be accessible. Also, the definition of a modeling language may inherit some of the limitations of the programming language. For example, in Java, a class cannot have a space in its name and neither can it use multiple implementation inheritance conveniently, so some kind of cumbersome or complicated tricks must be applied which might make the final system less clean resulting in more maintenance work.

The lesson to take from this is that one should not use the type and inheritance mechanism of the host language to define a metamodeling tool. In effect, a metamodeling tool provides a lightweight virtual machine with its own type system. Models and modeling languages should be seen as dynamic data structures that are created and updated regardless of the host language. This was implicitly established in Chapter 3, in which a dynamic environment of several languages and language extension mechanisms was described.

The metamodel layer as implemented in Coral can be seen in Figure 5.3. It bears a strong resemblance to the definition of the SMD language in Figure 5.1, although some small differences due to the C++ concept of strong ownership can be seen. The gray classes Atom and Slot represent how the model layer is connected to the metamodel layer. The Property.isAnonymous characteristic will be explained later.

The internal representation of models in Coral can be seen in Figure 5.4. We can create a model by instantiating suitable Element objects and connecting them via their Slot objects. Every Element is linked to its type, which is an Element-Definition object. The Slot objects of an Element are given by the Property objects

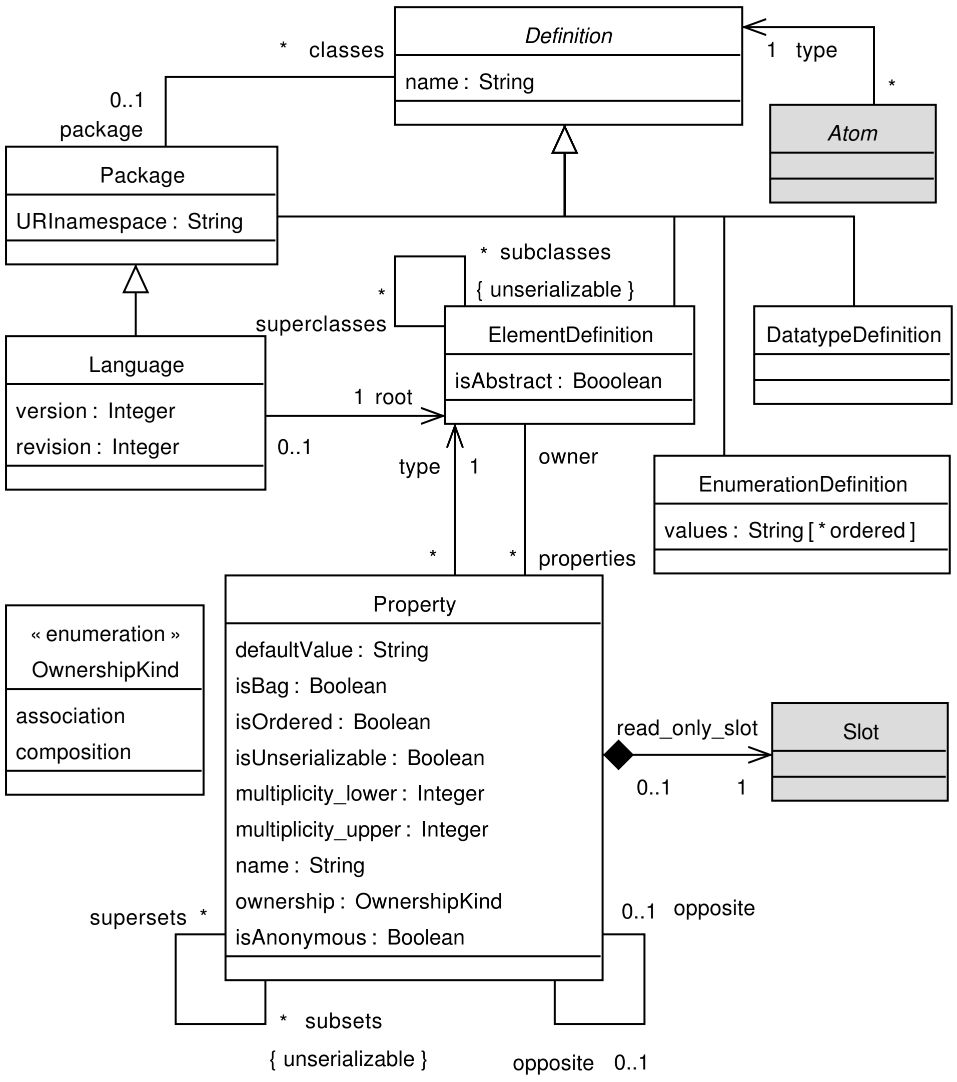


Figure 5.3: The Metamodel Layer in Coral as a C++ Class Diagram

of the type of the Element. Thus, each definition in a modeling language is implemented as an object instance of the ElementDefinition C++ class, while each element in a model is implemented as an object instance of the Element C++ class. That is, the definition of a UML State is an object in the Coral runtime while an instance of a UML State is also an object. This allows us to be fully dynamic, as new languages can be created at runtime. Also, Coral has complete control over the inheritance and property mechanism of the modeling languages. That is, the inheritance mechanism in modeling languages defined by SMD is completely independent of the inheritance mechanism in C++.

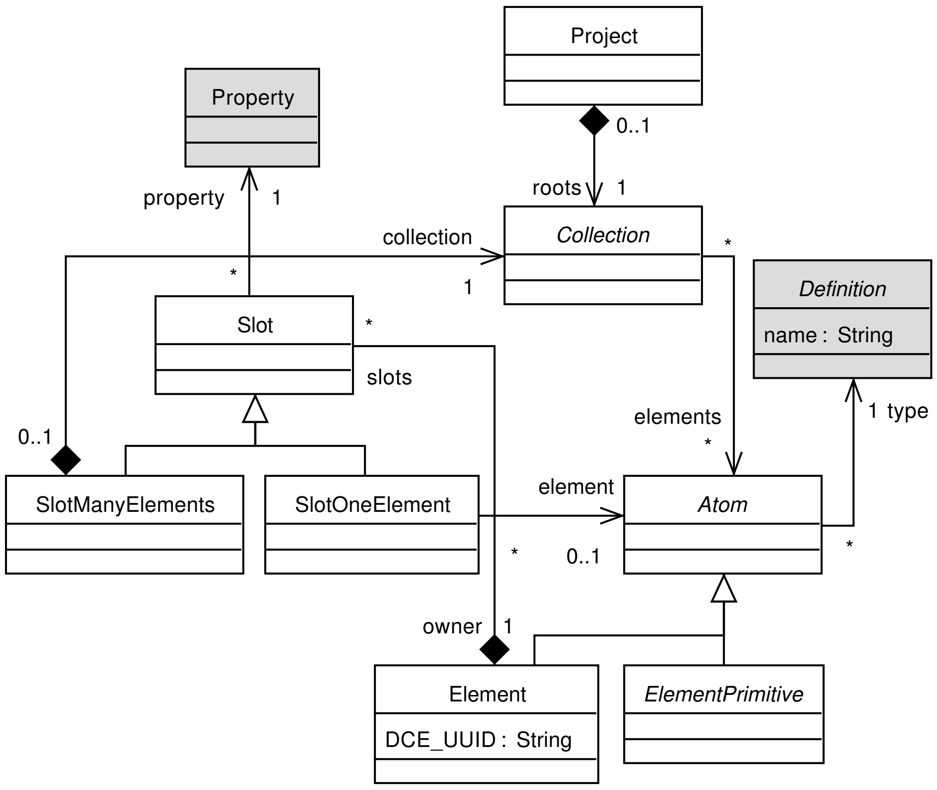


Figure 5.4: The Model Layer in Coral as a C++ Class Diagram

The model layer in a model manager is responsible for managing models. Modification of the data is performed via a program interface, which provides suitable interfaces for valid modification of models. A discussion of different features of such interfaces can be found in [152]. A detailed description of one of these interfaces is the Java Metadata Interface (JMI) Specification [79].

Slots are implemented using two different classes: one class for slots with only one (possibly optional) element and another for slots with an arbitrary number of elements. The latter case requires different containers. Our implementation of containers supports the cross product $\{$*unordered*, *ordered*$\} \times \{$*set*, *bag*$\}$. These are the well-known four different OCL collection types [131] and are implemented as four different classes with a common Collection superclass. Currently, bags are not supported in slots, so the only possible subclasses that can be used are Set and OrderedSet. However, bags can otherwise be used in other collection objects.

However, while collections are simple element containers, a slot is also responsible for maintaining bidirectionality. Coral automatically takes care of bidirectionality, i.e., modifying a Slot modifies also the opposite Slot. The interface does not allow the user to modify the underlying Collection of the Slot directly.

The primitive element types are implemented as separate C++ classes, where each class wraps the corresponding primitive C++ type. This can be seen in Figure 5.5. The reason for using classes instead of the primitive C++ types directly is that this makes some data processing internally easier, as both the Element and ElementPrimitive classes have a common superclass called Atom. The strings used by Coral are internally encoded as UTF-8 [178], meaning that any string data can be represented, and for example Scandinavian or Asian characters are supported.

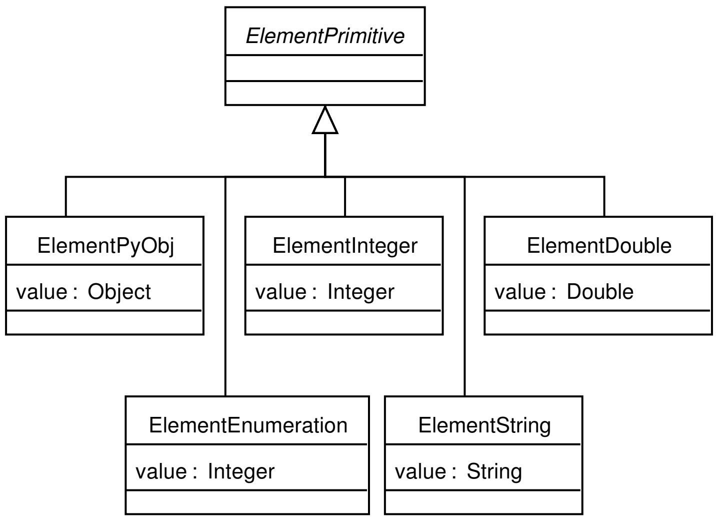


Figure 5.5: Primitive Element Types

5.3.2 Registering New Metamodels

After bootstrapping Coral, it only knows of two languages: SMD itself and a helper language called XMI 1.2 which handles XMI.Extension elements for serializing models into the XMI format. Serialization using XMI will be discussed extensively in Chapter 6.

An important property of a metamodeling tool is the ability to load new metamodels on demand. In Coral, this requires a facility for recognizing which metamodels exist and where they can be found, and a routine for loading SMD models and interpreting them as metamodels.

The first task is done by searching through the filesystem in predefined directories for short description files which map the XML namespace declaration or the tuple (name, version, revision) to the corresponding SMD model file. When Coral is told to load a model from an XMI file, that file must have an XML namespace declaration which describes which metamodel has been used. The loading of the model is temporarily suspended and the metamodel is loaded. In Coral, this metamodel is further converted into a language.

The second task is a special routine called *metamodel2language* that interprets a SMD model consisting of Elements and Slots as a modeling language consisting of ElementDefinitions and Properties. Once the new language has been configured in the kernel it is possible to continue loading the original model, or create models based on this language. This arrangement can be seen in Figure 5.6.

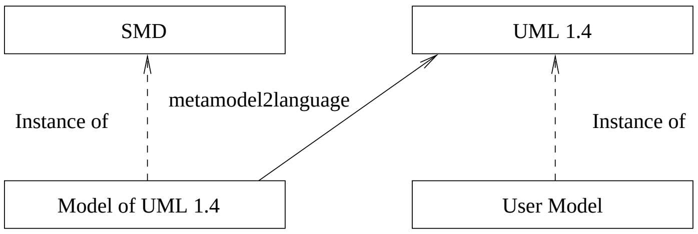


Figure 5.6: Lifting an SMD Model to the Metamodel Layer

Similarly, there also exists an opposite operation, *language2metamodel*, which transforms a language back to an SMD model. This is portrayed in Figure 5.7. Using this technique, we can manage our languages using the same interfaces as we use to manage models.

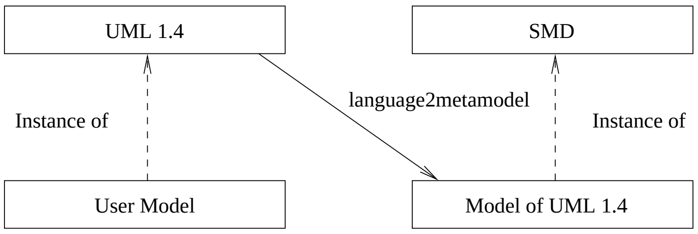


Figure 5.7: Lowering a Language to the Model Layer

A model manager only needs to provide built-in support to convert models from one specific metamodeling language. In the case of Coral, we only provide a mechanism to convert an SMD model into the internal representation of a modeling language. This decision is arbitrary but sufficient, since it is possible to define model transformations from other languages into SMD. Examples of these transformations, which are available in Coral, are:

- From UML class models to SMD.
- From ECORE models to SMD.
- From a human-readable textual notation to SMD (similar to the KM3 [84] notation), made by Ivan Porres.

We consider that it is more cost-effective to define such conversions as a normal model transformation by a separate script than to provide support in the model manager for all these languages. Other users can define as many of these transformations as required using a high level transformation language without increasing the complexity of the model manager.

Metamodels are relatively small. Even languages as large as UML 2.0 only have a few hundred definitions. This allows us to perform the conversion from metamodels to languages on the fly, i.e., when a model based on one of these languages is loaded by the kernel.

5.3.3 Enforcing Bidirectionality

Coral enforces bidirectionality according to the model constraints defined in Chapter 3, with the behavior as defined in Chapter 4. There is an interesting addition for aiding navigability, whereby unidirectional relations are always and automatically made bidirectional in case both classes are nonprimitive types. This automatically created property is called an *anonymous* property in our terminology, meaning that its isAnonymous characteristic is set. It requires a careful explanation. Anonymous properties are useful when we want to extend a modeling language while maintaining compatibility with other tools. Imagine that we want want to create a new modeling language that combines UML classes and Petri nets. In our new language we want to be able to navigate from a class to a Petri Net and vice versa. The problem is that if we extend the definition of a UML class with a property definition that points to the Petri net, the resulting models will not conform to standard UML. Other tools may have problems loading our extended UML models, since the models will contain references to Petri nets. Even UML tools based on our model manager may panic if they can reach a Petri net from a UML class, for example, using reflection.

The solution is to hide this property by specifying that the property that links a UML class with a Petri net is anonymous. The property is then only accessible by its opposite property. In this way, only tools that explicitly know about the Petri nets can reach these models.

An anonymous property is different than a unidirectional relation. A unidirectional relation is only navigable from one property while a relation with an anonymous property is navigable from both properties, provided we are aware of its existence. In Coral, the only unidirectional relations are the ones with a primitive type. From an ElementPrimitive object, Coral cannot access the Slot objects which point to it.

An anonymous slot is one which conforms to an anonymous property. It cannot therefore be accessed directly. However, an anonymous slot can be accessed via its opposite slot. Since anonymous properties do not have proper names, slots conforming to them cannot conflict with existing slots in a model element or any future extension. On the other hand, such slots cannot be serialized into an XMI document and are thus automatically unserializable. Only one the properties of a binary relation can be anonymous. It is possible to simulate the concept programmatically, using a separate data structure. This is prone to errors, since a developer might forget to update it correctly. Using anonymous properties and slots is more convenient, with usually a marginal increase in memory and processing time consumption due to the anonymous slots.

It can be noted that OCL [131] explicitly permits tools to support the concept of anonymous slots.

5.3.4 Optimizations

Performance is an issue in a model manager since industrial-size models may contain millions of model elements across several files or databases. Unfortunately, there are no standard benchmarks tests to compare Coral with similar tools such as EMF or NMR. Thus we only have anecdotal experience in using the interface for various tasks and can conclude that the Coral model manager is sufficiently fast for interactive uses for tens of thousands of elements.

An interesting optimization is the delayed creation of slots. For example, given that a UML 1.4 Class element has over 40 slots but usually only a relatively small subsets of them are used in some specific instance, it makes sense to create slots lazily, i.e., the first time they are accessed. In the common case of using only some slots of an element, this makes significant heap space savings, thus even giving a net speed increase as less work has to be done.

Another optimization can be seen in Figure 5.3, where each Property object has a corresponding empty Slot object in Property.read_only_slot. We have observed that many routines only query the slots of elements without modifying them. For example, saving a model queries all slots. Thus delayed creation of slots would not be very useful, since a save operation would request all slots of all elements that were saved, thereby creating all the slots. But, it is possible to recognize read-only purposes in C++ with the *const* keyword, meaning a logical and shallow constant view on an object [171]. Thus, when querying a const Element object for Slot objects, we can return the corresponding read-only Slot object of the Property object in question. This has considerable heap space savings, but the benefit is lost by language bindings without the same concept of logical constness as the one in C++, such as Python. However, algorithms written in C++ all benefit from this optimization almost without effort from the developer.

5.3.5 Known Limitations

This section explains the known limitations of SMD or its implementation, Coral. They are based on our experience of using the implementation, and emphasizes the relevance of trying out an idea in practice.

No Explicit Relations

The binary relations provided by the SMD language are formed by two cooperating Property objects. These constitute the underlying bidirectionality that is prevalent in Coral and other modeling frameworks. However, there is no explicit concept for the relation itself. This is unlike MOF 1.3 or GXL, where there are Association and RelationClass elements to keep explicit track of relations.

There are certain small drawbacks in not using an explicit object for the relation. For example, the data in a Property object must conform in a very specific way to the opposite Property object in a relation. For example, if one Property object is set as a composite, the opposite cannot be composite, as explained in Section 3.4.4 by metamodel constraint 3. This information could be encoded in the relation, and thus no extra metamodel constraint would be necessary.

However, the biggest drawback is that sometimes we would like to refer to a relation although we do not have an explicit object for it. In other words, we have a need in our modeling technologies to refer to either elements or some particular connection between two elements (or classes, or metaclasses). We could reify a connection into an explicit object, but we can always take this problem one step further: is there a need to refer to the concept between an element and a relation? And when do we stop? Andreas Prinz, Jan P. Nytun, Leiming Chen and Sun Wei also mention this complication in [154]:

> It is possible to reify (i.e., view as objects) links and associations so that they can be modeled as objects and clabjects [sic] respectively... The difficulty in reifying links is not in working out how to view them as objects, but in knowing when to stop viewing them as objects...To break this potentially infinite regression it is necessary to identify certain kinds of links as implicit or primitive links which will not be stored as objects.

In hindsight, we would have benefited from an explicit relation object for describing the connection between two SMD Property objects. Our current graphical modeling layer built on top of the Coral model manager cannot show a link as an edge on screen because the relation is implicit. This might be merely an issue with the graphical tool itself, but it might also indicate a more fundamental issue. Unfortunately we have relatively little experience in understanding this concept, but we feel that this might be much more important than what we currently realize.

SMD is Hardcoded in the Tool

We have a lot of experience in actually using the Coral model manager, and we feel that we can draw some conclusions on SMD itself, and on using SMD. SMD is implemented as C++ code and therefore any change in SMD requires a change its underlying implementation. A language such as SMD contains a few but very important concepts that cannot be changed easily. Also, changes in SMD may require changes in any other tool component based on SMD, for example our graphical modeler. For example, adding shared composition to the interpretation of ownership will require a major revision of the model manager. Also, most tools must be reviewed and updated accordingly.

Having a fixed language to describe modeling languages has the implication that users cannot develop new metametamodeling techniques and languages without developing a new tool or extending the tool they are using. In contrast to this, an interesting idea is being developed by the Open Systems Development Group at Agder University College in Grimstad, Norway. The Semantic Metamodel-Based Integrated Language Environment project (SMILE) [153] led by Andreas Prinz strives to create a modeling framework in which the metamodels and models have a uniform representation and different semantics can be attached [122].

At the metamodel layer, the distinction between a Package and a Language is not very well motivated, and we could have removed the former one without any significant loss. At the model layer, the Project class is outside the metamodel/model abstraction as a separate C++ class. We feel that this should have been a class in a (perhaps special) metamodel, and then individual Project elements could be instantiated from it. Now, there are different interfaces for handling Project and Element objects, which is confusing. Also, we believe that having both SlotOneElement and SlotManyElements is an optimization made too early. Several internal routines have to take both subclasses into account, which leads to more cumbersome programming.

The current implementation of the model layer assumes throughout that elements are kept in memory. There is no support for on-demand loading and unloading of models. In our research, we have not had problems with this limitation.

Primitive Datatypes

Another consequence of SMD being hardcoded in the tool is that there is no simple way to add new user-defined datatypes to SMD. This would require explicit support from the Coral model manager to function properly. A special case are EnumerationDefinition objects, which the developer is allowed to use to define new enumerations.

Another known limitation with the current numeric primitives is that no magnitude or range limits are defined; there is no metamodel-independent way to know if a numeric value models small or big values, and the model manager cannot be allowed to round numbers to, for example, produce prettier output. A generic tool has to display and store a floating-point value with the precision provided by hardware or use a numeric library that supports arbitrary-precision (floating-point and integer) numbers, both resulting in long awkward numbers. Coral currently uses the precision of the underlying hardware for both floating-point as well as integer numbers, as this question has not been interesting to us.

Languages and Metamodels are Distinct

Languages and metamodels-as-models are distinct concepts. Even though there is a one-to-one correspondence between a metamodel and a language, it would be more convenient in practice, albeit slightly less efficient, if these were the same objects in C++.

This is a part where we believe that Coral should have been developed as most virtual machines like common C#, Java and Python implementations, where ElementDefinition (or its equivalent) is a subclass of Element, instead of being a completely separate concept. Reusing or modifying an existing virtual machine to accommodate the additions that modeling technology brings us could have been a faster path to a good implementation.

5.4 Related Work

Related work in this field can be split into two partially overlapping areas. One area contains articles about metamodeling languages and de facto standards for metamodeling languages, published articles on suitable algorithms and structures. The other area includes practical implementations of these metamodels as software tools.

Among the more popular languages to define modeling languages are the MOF and EMF metametamodels. Coral has several common and similar traits with both MOF and EMF. Eclipse itself is a full-fledged pluggable development environment and utilizes EMF to provide a good middle ground between two extremes, programming and modeling. EMF provides a subset of UML in the form of class diagrams. It generates Java classes with get/set methods for individual slots as well as providing a generic API for accessing all the slots.

Varro and Pataricza presented in [183] a multilevel metamodeling technique called VPM. VPM uses covariant specialization of elements definitions and single element inheritance.

The KM3 metamodeling language [84] and Kermeta [90] are implementations built on top of the Eclipse platform. Microsoft has relatively recently begun their DSL Tools effort in the context of Software Factories [66]. Essentially there are very few differences between any of them, or with Coral. The DSL tools have more integrated support for describing the concrete syntax of the constructs of a new metamodel.

Coral does not generate code for elements in a similar vein as Eclipse EMF. Instead, Coral keeps everything as models, thus primarily positioning itself as a metamodel-based modeling tool rather than an integrated development environment for source code with round-trip capabilities.

Most modeling tools only support one modeling language. Previously this was UML 1.3 or UML 1.4, nowadays it is usually UML 2.0. Due to this, we do not consider them as metamodeling tools. Although they are certainly useful, they are limited in scope as they cannot be used for creating new domain-specific languages.

Comparing SMD with the modeling frameworks from Chapter 2, we can note that SMD is slightly smaller than ECORE—mostly because ECORE has more functionality in the form of operations—and provides the anonymous and superset property characteristics.

5.5 Conclusions

The discussion about Model Driven Engineering is quite often centered on modeling languages and model transformation languages. However, we consider that it is equally important to discuss the features and development of model managers model transformation engines and model transformation tools that support such modeling languages and transformations. The Coral tool is an attempt to seek practical and theoretical issues in these topics and provide a useful working solution.

All existing model managers are based on a specific metamodeling language and all the existing metamodel kernels implement at least a subset of MOF. However, MOF as such is not a model manager as such since it is not a software tool. In our approach we use SMD as the metamodeling language. SMD is similar to MOF but it is based on fewer concepts.

In this chapter we have described how the metamodel and model layer are implemented in the Coral model manager, based on the formalization in Chapter 3. We note that even though a formalization is extremely important, it is still difficult to ask all the right questions when operating within a theoretical framework. Examples of such questions are what a good serialization format is, and how usable the framework is.

Coral has been used as a library by several other projects within our research group, such as a UML editor, MICAS, SOCOS, and several other smaller tools. We again stress that these projects are accomplished by other people and do not constitute a part of this thesis, except as to serve as validation of the viability of Coral as a library for metamodeling.

Coral is distributed as open source under the terms of the GNU GPL version 2 license [59] at `http://mde.abo.fi/tools/Coral/`. The current version available is 0.9.3.

Chapter 6

Model Serialization and Interchange Using XMI

6.1 Introduction

In this chapter we discuss issues surrounding model serialization and using XMI in particular as the solution to model serialization. One of these challenges of modeling is how to represent models in a machine-independent format to allow model interchange between tools and systems. This exchange format should be well-documented, stable and supported by different tools from different vendors.

It is important to realize why a serialization format is necessary in the first place. The reason is rather simple. Our current global computing environment and infrastructure consists of several weakly connected computers. What we mean by this is that a computers is not inherently dependent on other computers, and that we do not have a global namespace for reliably addressing and acquiring all data that the computers contain. To be able to communicate between computers, we cannot in general keep our data in databases or in the memory of a computer; instead the data must be transported to the other computer. This transportation must ultimately be accomplished as a serial stream of bits. Hence, there is a clear need for a serialization format.

OMG proposes the use of XMI [129, 132, 137] to enable model interchange. One of the strong points of XMI is that it is an XML [188] application, as XML has been successfully used to support many document and model representation standards. XML is well-documented, machine-independent and there exist plenty of tools supporting it. Thus, we could assume XMI files should be portable and easy to parse. According to Perdita Stevens, XMI could revolutionize the use of models in software engineering [168] by making it easy to create programs that analyze and modify models. We fully agree with this idea. However, we must make sure that XMI is a suitable technology for model serialization and interchange. The contents of this chapter are devoted to solving this matter.

There is at least one tempting alternative to defining a serialization format explicitly for models. *Automatic object serialization* refers to the technology of automatically serializing objects according to a welldefined format. However, our anecdotal experience is that such serialization is tied to the internal implementation of the application in question. Modifying the application too easily modifies the serialization format. Our current computing environment also verifies this understanding. We have almost no automatic serialization technology widely used in a environment with multiple platforms and programming languages. Instead we are basing our intercomputer communication formats increasingly on well-defined, publically available documents, such as the TCP/IP, SMTP and HTTP protocols, and the OpenDocument, PNG and HTML content formats.

The question we raise is how suitable the XMI standards are for model interchange in practice. We will review different scenarios for model interchange, how suitable the current standards are to implement these scenarios and how different tools implement the standards. We will also propose different improvements for the standards. The results presented in this chapter have been obtained using three different approaches. First, we have studied the OMG documents that describe the XMI and related standards, as well as several W3C standards. We have also performed practical experiments to test model interchange between different tools, including commercial UML tools such as Rational Rose and open source model repositories such as Eclipse EMF. Although the OMG standards are supposed to be the authoritative definition of the model interchange formats, we have observed some discrepancies on how different tools implement model interchange features.

Finally, we have implemented different research tools that include XMI support. We have also developed a prototype multiuser model repository based on XMI. This work has allowed us to obtain first hand experience on all the issues that appear when implementing XMI in real applications.

This chapter is based on Publication VI. We proceed as follows. Next, we enumerate some requirements for a suitable serialization format, and give some usage scenarios for model interchange. In Section 6.3, we examine the XMI standards in sufficient detail in both a theoretical and practical context for us to discover the problems with them. In Section 6.4 we provide some validation of our research. We finally go through some related work in Section 6.5 and conclude in Section 6.6.

6.2 Scenarios for Model Interchange

This section lists some basic functionality required by any serialization technology, and four highlevel usage scenarios for model interchange.

6.2.1 Basic Functionality

We have already discussed some of the basic functionality described in this section in Chapter 2. First of all, any model serialization should be able to serialize the actual contents of a model. This includes the various concepts defined by the metamodel, such as ordered slot contents and the associations between elements, and it must also be especially noted that primitive values can contain arbitrary data as strings, numbers, et cetera.

Second, we must be able to realize which metamodel has been used, and what the type and identity of each element serialized is. As we would like to transfer parts of our (potentially huge) model, it should be possible to use multiple files and connect elements from different files together.

According to the OMG standards, diagrams are expressed as Diagram Interchange models. Thus, a serialization technology needs to be able to deliver a model described using several modeling languages (e.g., UML for the abstract data and DI for the diagram data). Various extensibility mechanisms for arbitrary nonmodel data could be of interest as well. For a viable collaboration environment, various model difference and related algorithms are certainly important.

Although these criteria might not exhaustively list all necessary features, they serve as a set of base cases against which a serialization technology can be compared.

6.2.2 Scenarios

We draw the scenarios from our experience in software engineering. We have tried to list them in an increasing order of difficulty from the point of view of the serialization technology. Although this list is not exhaustive, we believe it addresses several important issues that any serialization technology should address.

Migration from One CASE Tool to Another

Our first scenario describes the need to move all our models from one CASE tool to another, probably from a different vendor. To avoid vendor lock-in, we use our serialization technology to provide seamless, robust and consistent ways to migrate from one CASE tool to another. We plan to migrate the models once and not use the current CASE tool any more.

This is the simplest scenario. In this case, the size of the files or the speed of the serialization export and import process is not too important. Also we may tolerate small defects in the migration, if they can be easily found and corrected in the new CASE tool.

Model Interchange within a Desktop

Another scenario for model interchange is to use inter-application communication mechanisms such as cut-and-paste and drag-and-drop to share models or fragments of a model from one application to another. This feature is present in many commercial UML tools. For example, it is possible to select a part of a UML class diagram, copy it to the clipboard and paste it into a Microsoft PowerPoint presentation. We should note that in this case, the modeling tool is not placing a model to the clipboard but a picture of a model in a graphical format such as PNG, since presumable Powerpoint does not understand models. However, nothing prevents the use of a standard model interchange format between applications running within the same desktop.

We should note that in this scenario we will interchange just a small part of a model. Therefore, our small file will probably contain links to other files.

A Model Driven Engineering Tool Set

In this scenario, different tools are used to create and maintain the models. Each tool may be specialized in a certain task, such as requirements engineering, analysis modeling, or model implementation. The tools are not necessarily used in sequence; they could be used for forward, backward and round-trip engineering.

Since we use many different tools, the files representing our project may include elements that are not supported by every tool. However, all the tools should respect and leave intact those unsupported elements, i.e., they should be able to load and save the elements even though they do not understand anything about them.

A Multiuser Model Repository

In this scenario, we plan to store all the models in a central or distributed repository, easily accessible by different developers using heterogeneous tools. The serialization technology is used as the exchange format between the repository and the developers' tools; in this chapter, we ignore the transport protocol used. This is the most demanding scenario for serialization. The files may include version information or represent differences between models. Also, as within the desktop scenario, we might only interchange parts of a model.

6.3 XMI

In this section, we examine the XMI 1.x standards as well as the relatively new XMI 2.x standards. We will not go into the details, but rather give an example description of how example models serialized in these contents formats look like, and how well XMI tackles the scenarios and issues described in Section 6.2. We conclude with some ideas on how XMI could be improved.

6.3.1 Using XMI

To understand how models can be interchanged using the XMI standard we need to understand how models are organized according to a modeling language. We reuse our knowledge from Chapters 2 and 3. We can conclude that the composition hierarchy in a model forms a tree. Shared composition is not used in OMG standards.

As an example, we demonstrate a finite state machine (FSM) model encoded using the two different versions of XMI. Figure 6.1 represents a particular model in the FSM language, rendered in its own concrete syntax. The states are drawn as circles with their names inside, and transitions between states are drawn as arrows. Each transition also has a token which is the input required for the transition to be taken.

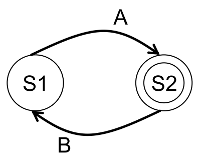


Figure 6.1: Example Finite State Machine Model

In Figure 6.2 we show a simple modeling language to describe such FSMs.

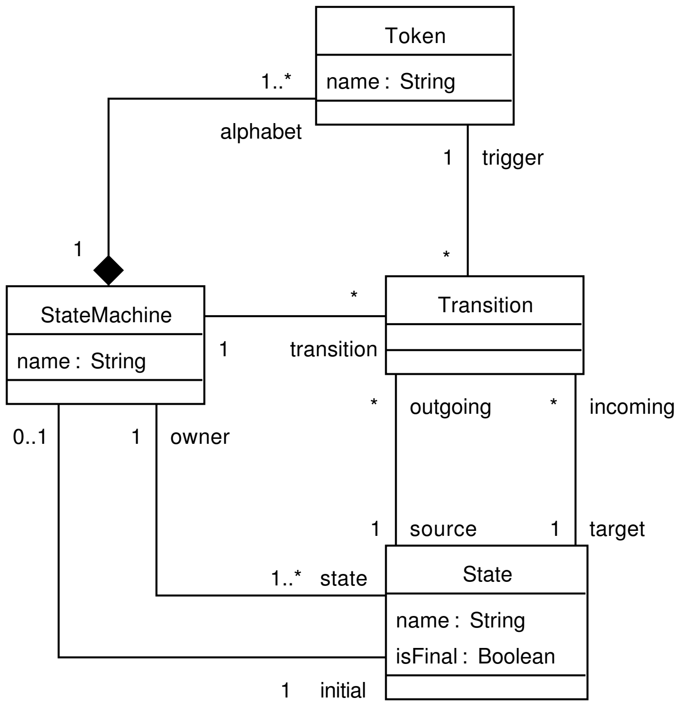


Figure 6.2: Example Finite State Machine Metamodel

Since the XMI standard defines a series of rules to serialize any MOF-based modeling language, we can also represent the example abstract model in XMI using these rules. The resulting XML file using XMI 1.2 is shown in Figure 6.3, and using XMI 2.0 in Figure 6.4. We note especially that the concept of composition in modeling maps to the concept of hierarchy in XML.

We can see in the XMI files that a model element can refer to other elements using the *xmi:id* of those elements. This is necessary since the underlying structure of a model is a graph while an XML file is a tree. It is also possible to link across XMI files (not shown in the figures), i.e., a model element may refer to elements that reside in a different file. This feature enables us to use XMI in larger projects.

In the context of a modeling tool, an XMI filter is a tool component that can create and retrieve XMI files based on one or more given metamodels. Since a metamodel is based on a model, it can also be represented as an XMI file. We should note that although it is possible to create a DTD or XML Schema [193, 194] to define the structure of an XMI file this is not strictly necessary, since an XMI filter can obtain all the necessary information about the structure of an XMI file from a metamodel. Actually, a metamodel contains additional information that cannot be expressed in a DTD. An example of this are relations which are unordered such as a UML Package owning a set of UML Classes; in XML, the order is always important.

Furthermore, XMI has facilities for describing model differences using the XMI.difference XML element. Also arbitrary XML fragments can be added to any model element using XMI.Extension, or to the whole model using XMI.Extensions.

It is also import to remark what XMI is not. XMI cannot be used to define the structure of a modeling language. This is the role of the MOF standard. Actually, XMI can be used to serialize any MOF-based modeling language including, but not restricting to, UML and MOF itself. The fact that XMI is independent of a modeling language is at the same time one of its strong points and major weaknesses. Two tools that use two different versions of the UML standard, for example UML 1.3 and UML 1.4 will not be able to exchange models even if they use XMI.

Surprisingly, the UML and MOF metamodels do not contain information about the diagrammatic representation of models. A UML model may state that there is a class named "Person" in a model, but it cannot state that this class is represented in a diagram by a rectangle in a certain position, size and color. To remedy this situation the DI [136] standard has been proposed. DI is not a model interchange format but a metamodel to describe diagram information.

We should also remember XMI is neither an application programming interface to retrieve information from models, such as the Java Metadata Interface (JMI) [79], or the Eclipse EMF [28], nor a communications protocol to transport models between systems such as HTTP [56] or WebDAV [62]. This implies that there are no standard software components to create or retrieve XMI files from a filesystem or multiuser repository since the API and communication mechanism for such components has not been standardized.

```
<?xml version='1.0' encoding='UTF-8'?>
<XMI xmlns:XMI="http://schema.omg.org/spec/XMI/1.2"
       xmlns:FSM="http://www.example.com/FSM/1.0"
       xmi.version='1.2' timestamp='Sun, 01 Jan 2006 11:08:44 +0200'>
  <XMI.header>
    <XMI.documentation>...</XMI.documentation>
  </XMI.header>
  <XMI.content>
    <FSM:StateMachine xmi.id="e1" name="Example" initial="e6">
      <FSM:StateMachine.alphabet>
        <FSM:Token xmi.id="e2" name="A" transition="e3">
        </FSM:Token>
        <FSM:Token xmi.id="e4" name="B" transition="e5">
        </FSM:Token>
      </FSM:StateMachine.alphabet>
      <FSM:StateMachine.state>
        <FSM:State xmi.id="e6" name="S1" incoming="e5" outgoing="e3"
                     isFinal="false">
        </FSM:State>
        <FSM:State xmi.id="e7" name="S2" incoming="e3" outgoing="e5"
                     isFinal="true">
        </FSM:State>
      </FSM:StateMachine.state>
      <FSM:StateMachine.transition>
        <FSM:Transition xmi.id="e3" source="e6" target="e7" trigger="e2">
        </FSM:Transition>
        <FSM:Transition xmi.id="e5" source="e7" target="e6" trigger="e4">
        </FSM:Transition>
      </FSM:StateMachine.transition>
    </FSM:StateMachine>
  </XMI.content>
</XMI>
```

Figure 6.3: Example Finite State Machine Model as an XMI 1.2 File

6.3.2 Assessing XMI Suitability

In this section we study some of the inherent problems that exist in the XMI standard. Many of these problems are a consequence of the rather fast development of the standard, with multiple versions appearing in a relatively short time. Also, the initial versions of XMI (and UML) were probably created taking into account only some of the scenarios from Section 6.2

```
<?xml version='1.0' encoding='UTF-8'?>
<xmi:XMI xmlns:xmi="http://schema.omg.org/spec/XMI/2.0"
         xmlns:xlink="http://www.w3.org/1999/Xlink"
         version='2.0' timestamp='Sun, 01 Jan 2006 11:08:44 +0200'>
  <documentation>...</documentation>
  <FSM:StateMachine xmlns:FSM="http://www.example.com/FSM/1.0"
                    xmi:id="e1" name="Example" initial="e6">
    <alphabet xmi:id="e2" name="A">
      <transition xmi:idref="e3" />
    </alphabet>
    <alphabet xmi:id="e4" name="B">
      <transition xmi:idref="e5" />
    </alphabet>
    <state xmi:id="e6" name="S1" isFinal="false">
      <incoming xmi:idref="e5" />
      <outgoing xmi:idref="e3" />
    </state>
    <state xmi:id="e7" name="S2" isFinal="true">
      <incoming xmi:idref="e3" />
      <outgoing xmi:idref="e5" />
    </state>
    <transition xmi:id="e3" source="e6" target="e7" trigger="e2" />
    <transition xmi:id="e5" source="e7" target="e6" trigger="e4" />
  </FSM:StateMachine>
</xmi:XMI>
```

Figure 6.4: Example Finite State Machine Model as an XMI 2.0 File

Arbitrary Data

An important and most likely unintended consequence of basing XMI on XML is how arbitrary (binary) data is handled. Surprisingly, XML cannot easily express any binary string of data, e.g., the null byte. There are at least two viable workarounds for this. The first one is to Base64-encode the binary data [83], although this increases the size. The second one is to support the relatively new XML-Binary Optimized Package (XOP) [196] standard. Unfortunately, both of the workarounds require the cooperation of all tools. Otherwise, XML supports data using any encoding. Especially, Unicode [178] encodings are supported.

There are other limitations given by XML as well. For example, properties from the modeling technical space are sometimes encoded as XML node attributes; however, XML attribute names that start with the letters *xml* are reserved by the XML specification.

Too Many Versions, Options and Optimizations

One of the main impediments to use XMI in practice is the large number of different versions of the XMI standard and UML metamodels, and the already-existing different implementation variations. At the moment of writing this text, there are five versions of XMI (1.0, 1.1, 1.2, 2.0 and 2.1) and seven versions of UML (1.0 to 1.5 and 2.0). This means that a file containing a UML model can actually be serialized in 35 different combinations of XMI and UML versions. The diversity of different XMI implementations has already been studied in [82], in which Jiang and Systä devised a method to explore differences between XMI formats using DTDs.

As an example, the name and isActive slots of a UML 1.4 Class in XMI 1.x has been serialization in several different ways by various tools, as can be seen in Figure 6.5. Special code is required to support all the different constructs.

```
<UML:Class name=”MyName”>...</UML:Class>

<UML:Class ...>
  <UML:ModelElement.name>MyName</UML:ModelElement.name>
</UML:Class>

<UML:Class ...>
  <Foundation.Core.ModelElement.name>MyName
  </Foundation.Core.ModelElement.name>
</UML:Class>

<UML:Class ...>
  <UML:Class.isActive xmi.value="false"/>
</UML:Class>

<UML:Class ...>
  <Foundation.Core.Class.isActive xmi.value="false"/>
</UML:Class>
```

Figure 6.5: Example Serializations of name and isActive Slot for a Class in XMI 1.x

The fast evolution of XMI and UML gives little room for implementers to try it out in practice. We consider that the solution to this problem is to include in each new version of XMI or UML, a mechanism to transform old files into the new version. In the case of XMI, this could be achieved at the XML level by including an XSLT [187] transformation file with the new version of the standard. In the case of a new version of UML, the transformation could be defined using QVT.

Some of the new features introduced in the latest version of XMI are questionable. A new concept related to XMI appears in the mapping of MOF models to XMI [134], where XMI files can be made smaller in size by removing some redundant information and instead serializing the derived information “because the derived form is more compact”. However, the problem is that they seem of little use. They make it more difficult to implement working a XMI reader since the variation in input that the reader must accept is greater. We have performed anecdotal experiments that suggest that these optimizations provide a minimal benefit with respect to file size, whereas standard compression tools, such as gzip, can obtain compression ratios of up to 95% on XMI files. A “binary XML” format has even been discussed [186, 191, 195] by the W3C, which would obviate the need for any application-specific (i.e. XMI) compression solutions.

Additionally, very loose directions are given by the standard itself: “to allow import, derived properties should only be made serializable if...it is possible to reverse-derive the base information from the derived form.” The rationale of the gain these optimizations bring is thus questionable, as they add more variability to the XMI format.

There might be arguments for a loose model interchange specification. Jon Postel, a well-known Internet persona, used to say “an implementation must be conservative in its sending behavior, and liberal in its receiving behavior”, i.e., maximum interoperability is obtained by being as strict as possible to published standards in the output, but accepting also slightly malformed input when the meaning is still clear. This Robustness Principle [78] is often quoted as “Postel’s Law”. But that does not mean that the standard itself should be loosely specified. Furthermore, Postel discussed network protocols, not content formats, and even then the principle has been questioned [158]. As an example of where not following standards leads to, the infamous abuse of HTML has forced web browsers to support rendering in “quirks mode”, where the browser makes an educated guess on the document’s real structure, making different browsers produce different output. We believe this is not a path software modeling should follow.

Metamodel Identification

To be able to process a model, a tool must know its metamodel. However, XMI lacks of a reliable mechanism to identify which metamodels are used in a file. The problem is complicated since XMI 1.x and XMI 2.0 use two completely different approaches.

The XMI.metamodel element can be used in XMI 1.x to identify the metamodel used in a file (Pages 3-12 and 3-20 of [129]). An example of this approach is as follows:

```
<XMI.header>
  <XMI.metamodel name="UML" version="1.3" href="UML.xml"/>
```

In XMI 2.0, the XMI.metamodel element has been removed. Instead, each metamodel is assigned one or more XML namespaces. A namespace specification may be any arbitrary string, that according to the standard “provides a permanent global name for the resource. An example is *http://schema.omg.org/spec/UML/1.4*. There is no requirement or expectation by the XML Namespace specification that the logical URI be resolved or dereferenced during processing of XML documents”. In page 1-16 of [132] we can see the following example:

```
<xmi:XMI version="2.0"
          xmlns:UML="http://schema.omg.org/spec/UML/1.4"
          xmlns:xmi="http://schema.omg.org/spec/XMI/2.0">
```

The new mechanism is more correct from an XML point of view, and certainly enables one to mix models of different metamodels with ease. However, it is inconvenient as there is no published mapping between namespaces and existing versions of UML and other modeling languages. Even if that mapping exists, it is merely an opaque string to a tool. No information can be deduced from it, and thus a generic modeling tool cannot work with unknown metamodels; it must know, in advance, about the namespace mapping and the metamodel. Our solution in the Coral modeling tool is a user-editable table with mappings between namespaces URIs and actual metamodels. However, this solution requires manual work by the user and is not viable in the long term.

Another problem in creating a robust XMI import component is that old XMI files may combine namespaces with header information. One commercial tool produces the following header when exporting a model to XMI:

```
<XMI xmi.version=’1.1’ xmlns:UML=’//org.omg/UML/1.3’>
  <XMI.header>
    <XMI.metamodel xmi.name=’UML’ xmi.version=’1.4’/>
```

The namespace declaration may suggest that the file contains a UML 1.3 model, while the header states that it is a UML 1.4 model.

The problem of metamodel identification is an important issue and its final consequence is that it is not possible to load new metamodels based on their description from the namespace URI. Therefore the current definition of namespace declarations in XMI cannot be used as such in a metamodel-based modeling tool that should support any user-defined modeling language.

Element Identification

Element identification is the process of obtaining a unique reference to a model element that distinguishes it from the rest of model elements that compose all the artifacts required in a project.

The need for unique element identification was already tackled in XMI 1.x. Those standards provide two attributes to uniquely identify an element: the *xmi.id* and *xmi.uuid* attributes. The *xmi.id* attribute is used to reference elements inside an XMI file. They are unique arbitrary strings in the context of a given file. Although they must be retained by tools, *xmi.id* might need to be changed if the element is moved to another file, since some other element in the new file might have the same identifier. Therefore *xmi.id* can be a bit awkward in practice.

On the other hand a UUID is assumed to be a globally unique string, as has been discussed in Chapter 3. UUIDs may be assigned immediately, or the first time that an element is exported to an XMI file. Later, any standard-compliant open tool that imports the XMI file should not change or remove the assigned UUIDs, as per the XMI specification for open tools.

Unfortunately, many of the existing UML tools do not generate or preserve UUID strings and cannot be used rigorously to create links between different files. This is further examined in a small survey in Section 6.3.2.

It is possible to use other attributes instead of UUID to track the elements in a project, for example the name of a model element. However, this has some drawbacks. First, the name of a model element is a property of a modeling language, such as UML, not of XMI. It is possible to define a modeling language where the elements do not have a name. Also, even if we restrict XMI to UML, not all UML model elements have proper names. For example, a Generalization relationship is usually not named. The same applies to transitions in a statechart, links in a sequence diagram and many other minor but equally important elements. Furthermore, names can easily be changed by the designer, whereas UUIDs are supposed to be persistent information.

The *xmi.label* attribute is of special mention. It can be used to give elements an arbitrary name by the designer. But the standards do not explain why it should be used instead of a *uuid* or *id* attribute—so why use it?

A relatively recent addition to the XML family of standards is the xml:id Version 1.0 [198] by the W3C which is very similar to *xmi.id*. Indeed when XML provides similar or superior alternatives for element identification and element linking, we believe they should be used in lieu of the XMI standards. This allows the usage of general-purpose XML tools for navigation in models, but would also mean that XMI must be constantly updated to match the new standards by W3C. This also means that the XMI 2.x standards should not invent any own syntax and instead only incorporate the usage of XLink [189], XPath [197] and XPointer [192].

Tool Compatibility

Besides developing our own XMI tool component, we have also performed a simple study of the XMI files generated by two open source and six commercial CASE tools. There are many tools in the market that support UML in one way or another. The tools studied were chosen because they are popular, they can import and export XMI files and the vendor offers a trial or free version available on the Internet. We have created a simple model with each of the tools and exported it to an XMI file.

Then we have examined the generated file in a text editor to observe the header of the file to determine the XMI and UML version of the file. The objective of this study is to determine the effort of creating a new component that can import existing XMI files. If we would like to develop a new tool that can load and transform models generated by the studied tools, the XMI import filter of that tool should support all the XMI and UML versions used by the CASE tools.

A second experiment was performed to determine if a CASE tool preserves UUID strings. First, we have created a simple model and saved it as XMI. Then we have edited the XMI file with a text editor and added a UUID identifier to the Model element, the main element in a UML model. For example, in an XMI 1.x file we will add the following string shown in bold face:

<UML:Model xmi.id='1' name='Example Model'
xmi.uuid = '123' isRoot='false' isLeaf='false'
isAbstract='false' isSpecification='false'>

Then we have loaded the modified XMI file in the CASE tool. The modified XMI file should be imported without problems. Afterward, we exported the model again to an XMI file and opened it with a text editor. The Model element should contain the same UUID as introduced by us. If the CASE tool modified the UUID string or removed it completely then it does not preserve UUID strings.

The results from a small sample of tools are not too encouraging, as can be seen in Table 6.1. UUIDs are discarded by 6 tools out of 8 tools. Also, each tool seems to use a different combination of XMI and UML version. Of special mention is Together 6.0, that can generate 7 different kinds of XMI files from the same model, using different XMI versions (1 or 1.1), UML metamodels (1.1, 1.3 or 1.4) or extensions introduced by different XMI components (Unisys, IBM, or plain OMG). XMI compatibility between several modeling tools, both commercial and open source, have also been researched by Anna Persson et al [148, 149, 109]. Their results tell that compatibility is not achieved in almost all cases for XMI 1.x, but XMI 2.x fares better.

Advanced Concepts

Shared composition is a concept that existed in MOF 1.x but was not kept for the MOF 2.0 specification. It means that a model with shared composition forms a directed acyclic graph; with composition, a model forms a tree. XMI does not support shared composition in a meaningful way, especially when considering elements shared between multiple files. However, this might be a fundamental limitation in the design of our filesystems, and not a fault of XMI per se. We infer from the lack of composition in MOF 2.0 that the concept itself has been deemed to troublesome to support.

Subsets and unions are such a relatively new idea that they have received almost no adequate examination. As far as we know, our Coral tool is the only one to

Tool	Exporter	XMI Version	UML Version	Supports UUID	DI
Coral	Coral	1.2,2.0	1.x	yes	yes
EMF UML2	(no header)	2.0	2.0	no	no
Magic Draw 7.1	Unisys.JCR.2	1	1.3	yes	no
Poseidon 3.0	Netbeans	1.2	1.4	no	yes
Rational Rose	Unisys.JCR.1	1.1	1.3	no	no
Together 6.0	TogetherSoft	1, 1.1	1.1,1.3,1.4	no	no
Visual Paradigm 3	(no header)	1.1	1.4	no	no
Visual UML 3.24	Visual UML	1	1.3	no	no

Table 6.1: Tool Compatibility

support subsets and XMI serialization. There are some issues: a composite slot subsetting another composite one, and unserializable slots.

The first issue is relatively straightforward. Instead of serializing the same element twice (once for each composite slot) we only serialize it once, with any remaining composite slots serialized as associations. While this is not technically according to the XMI standard, we feel this to be a rather good and viable solution.

The second issue of unserializable slots is not as clear. If a slot is not serializable, we could assume that its contents can be derived from any opposite slots that are serialized. However, if both slots in a bidirectional relation are unserialized, we could also try to derive the contents (partially) from other subset slots; needless to say, this quickly becomes very complicated. Exactly how complicated deductions a tool should support is not clear. Upon loading a model, Coral makes an extra pass through the structure of the model to make sure that it is at least internally consistent with respect to bidirectionality, subsetting and multiplicity. This solution is not able to reconstruct all possible cases, but it must be deemed good enough until the XMI standard itself is improved.

6.3.3 Conclusions

After exploring the details of XMI, we should be ready to assess in what way XMI supports the different scenarios given in Section 6.2.

Our experience as users of different case tools suggest that even the simplest scenario, migration from one tool to another, can be problematic. In most cases, it is better to concentrate on one tool and its file format, than the standard XMI file format.

Model interchange within a desktop is a rather complicated subject, and therefore demands a lot of cooperation between different tools adhering to common usage practices and protocols. In the X Window System [163], the Inter-Client Communication Conventions Manual (ICCCM) [159] defines how the desktop clipboard functions. Especially it is required that data is given a MIME type [119]. However, XMI does not define a specific MIME type. In different tools, we have seen uses of the nonstandard *application/x-uml* and *text/x-xmi+xml* MIME types. Without this small deficiency, XMI could be used for model interchange within a desktop.

We consider a model driven engineering tool set to be a set of tools by different vendors, cooperating using several models of different metamodels. It is clear that XMI could be used for all this, but software modeling has not become mature enough so the industry has not established which metamodels to use. The lack of metamodels of good quality is quite similar to the problem of reusable components and libraries in text-based programming languages.

A multiuser model repository will be discussed extensively in Chapter 7. However, an important conclusion is that model difference calculation algorithms are important and that XMI.difference is not a suitable structure for them.

When analyzing the various versions of the XMI, one notices how XMI 2.0 has taken a very different road from previous versions by aiming for better XML and XML namespace [185] compliance. Indeed, the change is so dramatic that there seems to be little reason to support XMI 1.x in new tools. Unfortunately, there are many older tools in the market than only support XMI 1.x so in many cases it is necessary to support backwards compatibility reasons.

In light of the arguments given in this chapter, we primarily suggest a compliance test suite for checking XMI compatibility. It should consist of a set of files with successively more advanced XMI features. The current compliance points in appendix A of [132] are quite coarse-grained, and it is not always possible for vendors to know whether or not their tool provides the support they claim. Detecting successful loading could be done by evaluating OCL constraints on the loaded models as suggested by Stefan Haustein [72], although this requires an OCL or similar interpreter for automation. The constraints would verify that all the model elements, their interconnections, and the information stored in them are correct. This alone would mean that customers can check the official XMI compliance test scores and tremendously benefit from the interchange format. "Full compliance" as so often touted by vendors would in fact be split into several compliance levels of XMI features, such as support for interfile linking, XMI.extensions, XMI.difference and multiple models using multiple metamodels in one file. We expect that a compliance suite will improve the implementation of the XMI standards in modeling tools. However, we consider that XMI should also be improved in these directions:

- A standard mechanism for element identification. This includes a deprecation of *xmi.label* and enforcement of the *xmi.uuid* identification mechanisms. This means that if a tool imports an element that contains a UUID, the same UUID should appear in any future serialization of that element. We consider that a tool should support both *xmi.id* and *xml:id* according to user request.

- A standard MIME type.
- A standard mechanism to retrieve the metamodel used by a certain model, including future metamodels. This enables the creation of generic metamodel-based modeling tools.
- A thorough clarification on how subset slots should be serialized and interpreted, according to the problems discussed in Section 6.3.2.
- Each new version of XMI should include the definition of a XSLT transformation to convert files created using the previous version. Once these transformations has been defined for all versions of XMI, the XMI 1.x standards should be deprecated due to their lack of XML compliance.

The OMG should require that each new proposal to the XMI standards should include a compliance test suite and a transformation definition from old versions to the new one. The result would be that end users have a guaranteed and standard transition path for their models.

In January 2006, OMG partnered with Unisys Corporation in creating an XMI compliance testing effort [126]. Its success cannot yet be evaluated due to its novelty.

6.4 Validation of Research

We have built tools that include XMI 1.2 support (SMW [11, 152]) or XMI 1.x and XMI 2.0 support (Coral). Since several vendors release new versions of their software, it is not possible to give an accurate figure on how compatible our tools are with respect to XMI. However, a lot of effort has gone into making Coral quite resilient for errors in XMI, understanding several "dialects" of it. Coral has at some point been able to read MagicDraw, Poseidon, ArgoUML and Eclipse EMF files, for some versions of these tools, and sometimes only partially due to major inconsistencies (e.g. using composition where an association should have been used). Most of the tools have not supported the diagram interchange standard, instead opting for their own proprietary diagram information; Coral is not able to read these diagrams.

It can be noted that implementation in Coral uses a bit over 6 000 lines of C++ to support the following for any metamodel:

- Reading XMI 1.x and writing XMI 1.2.
- Reading and writing XMI 2.0, also including some EMF compatibility code.
- Reading and writing ZUML (packed XMI 1.x) files.
- Writing GXL 1.0.

Due to an unfortunate misinterpretation of the standards when writing the input/output routines of Coral, it currently does not retain *xmi.id* after loading a model. There is however no particular technical or scientific challenge in making it work.

Coral also supports the X Window System clipboard, making it possible to paste DI diagrams into graphics programs, or the corresponding XMI into text editors. Also drag-and-drop between multiple instances of Coral is possible. XMI is used as an exchange format for these functionalities, with a MIME type of *text/xml;charset=UTF-8*.

We have built a prototype model repository [5] that aimed to use XMI as far as possible. However, a repository becomes much more efficient by using model differences calculation algorithms, and we found the XMI.difference element and documentation to be inadequate. Difference calculation and the repository will be discussed extensively in Chapter 7.

6.5 Related Work

There are several model interchange formats available. Perhaps the most well-known proprietary one is the undocumented Petal format used in the Rational Rose modeling tool, and lately the XML format used by Microsoft DSL Tools. Another interesting format, discussed in Chapter 2 is GXL [208]. It could certainly be used for serializing models, although it provides only a modest improvement over XMI.

Anna Persson, Henrik Gustavsson, Brian Lings, Björn Lundell, Anders Mattsson and Ulf Ärlig have examined the usage of XMI 1.x in the context of open source development tools [148, 149], and later Björn Lundell, Brian Lings, Anna Persson and Anders Mattson analyzed XMI 2.x [109]. Their analysis strongly suggests that while XMI 1.x is not well-established as an interchange format between different modeling tools, XMI 2.x has fared better. This is probably because several vendors base their tools on Eclipse EMF and therefore on the serialization technology and compatibility provided by it.

It is interesting to note that there are nowadays other technologies than XMI for model persistence. The Teneo library [54] supports creating an object-relational mapping from EMF models to a relational database. It also provides persistence using Hibernate [74]. Adaptive provides a MOF-based repository called the Adaptive Repository [2] and has versioning capabilities according to their own information. The problem with using a database for persistence was already stated in the introduction; databases cannot easily be transported between two different users.

6.6 Conclusions

We consider that in order to apply MDE in practice, model interchange between tools should be as frequent and simple as, for example, source code exchange between text editors, compilers and build tools. In this chapter we have first discussed several common usage scenarios for models that a serialization technology should address. Then we have analyzed the XMI standards and how well they support these scenarios. We have also discussed important drawbacks in these standards. We do not present these drawbacks as a criticism to the work performed by the OMG and its contributors but as an opportunity to improve the state of the practice. Finally we provide solutions to some of the problems.

The current abysmal state of interchange in practice using XMI cannot be explained alone by complications in the standards themselves. We know from our own experience that the standards are usable as such and that it is possible to implement working XMI input and output routines with a relatively small effort. Still, basic interoperability is lacking between tools from different vendors.

Chapter 7

Version Control of Models

7.1 Introduction

In this chapter we study how to store and manage large models during the lifetime of a software project. The first generation of UML editors assumed a single developer working on a single model. This approach assumes that once a model is ready there will be no major changes and it can be distributed to the programmers as documentation. Programmers use the model as a reference design or blueprint for the code to be developed, but the model is not updated any longer. In this scenario, software evolution and maintenance reverts over to the program source code, not to the UML model.

However, this approach is not satisfactory if we plan to use models instead of source code as the main and most important description of our software. This requires that any model should always be up to date. In this context, there will be different developers working simultaneously on the same models. There will be different versions of the same model, targeted to different platforms or customer requirements, and evolution and maintenance will be carried out over the models. This implies that we need to use a proper configuration management system to keep track of the models.

Software configuration management (SCM) is a well-studied topic and there are many tools available on the market. It involves several different subtopics such as version control as well as change, build and release management. Configuration management is a key element in the management of any software development project. However, most of the existing tools are designed to manage either program code or informal documents in natural language. Yuehua Lin et al. have argued that we need new tools and methods customized to the idiosyncrasies of the modeling standards for model transformations and model comparisons [105]. Model comparisons are especially important for SCM.

We center our study to what we consider to be the central element of a configuration management system for models: a model repository with version control capabilities. The objective of this chapter is to raise different issues that appear when we try to create an efficient and practical version control system for models. We also provide solutions to some of the problems, and recognize which problems still exist. Here, we use the term efficient loosely; the most performant solution (for some particular metric) is not our goal, but we want to avoid the pathological cases, such as transferring the whole model over the network when only a part of it will suffice.

Thus, we discuss only a subset of the problems of software configuration management of models, namely that of the technicalities of versioning: calculating the difference between models, applying a difference to an old model to create the new one, and of merging differences created in parallel. We also consider the storage of models in a SQL server.

This chapter is based on Publications IV and VII. We are indebted to the work of Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina and Jennifer Widom, who described a generic routine for detecting change in hierarchically structured information [37]. This chapter applies that method in the context of modeling, with edit distance calculation (e.g., by Eugene W. Myers [120]) for handling ordered sets.

We proceed as follows: in Section 7.2, we explain the problem of versioning models. In Section 7.3, we explain the algorithm for calculating the difference between two models and for applying said difference to one model, producing the other. These algorithms alone can be used for saving transmission time or storage space. Section 7.4 explains how these algorithms are used to create a union algorithm between two models and their base model. Conflict situations in such cases are a fact, and a closer look of what kinds of conflicts can be avoided are studied. We give an outline of a model repository with versioning capabilities with a SQL backend in Section 7.5. We describe how the work of this chapter has been validated in the context of working tools in Section 7.6, discuss related work in Section 7.7 and conclude in Section 7.8.

7.2 Problem Exposition

Asset, version and configuration management are important activities in any large software development project. This is still true if we use models as the main description for our software. These models may represent different designs of the same subsystem, different subsystems created in parallel by several designers, nonexecutable project data, or combinations of these cases. If the models are large, we need special tools to compare and merge several different models into a new one that contains all the changes proposed by all the developers.

We may illustrate these problems as follows. Let us assume that the original model shown at the top of Figure 7.1 is edited simultaneously by two developers. One developer has focused his work on the classes A and B and decided that the subclass B is no longer necessary in the model. Simultaneously, the other developer has decided that class C should have a subclass D. The problem is to combine the work of both developers into a single model. This is the model shown at the bottom of Fig. 7.1.

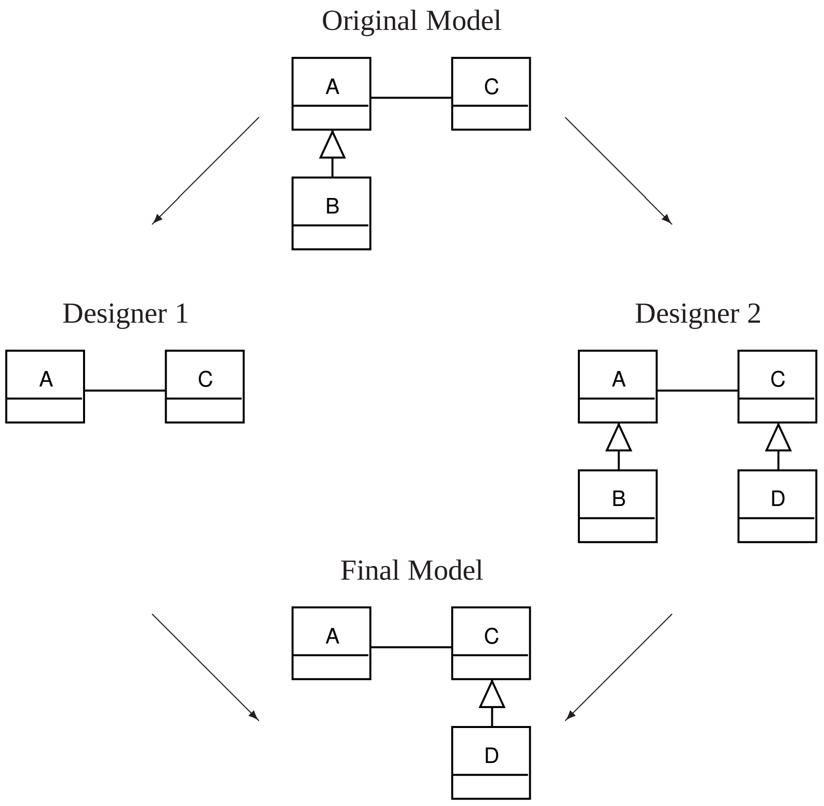


Figure 7.1: Example of the Union of Two Versions of a Model

In this chapter we study how to perform this operation in a generic way. The problem is not trivial. In the example, if we would just perform a straightforward union of the models, i.e., take all elements that appear in the two versions of the model, we would obtain a model that does not contain the changes proposed by the first developer. Our solution is based on calculating the final model based on the differences from the original model. Figure 7.2 shows an example of the difference of two models, in this case the difference between the models edited by the developers and the original model. The result of the difference is not a model conforming to the metamodel used by the original model. An example of this is shown in the bottom part of Fig. 7.2. In this case, the difference of the models contains *negative* model elements, i.e., elements that should be removed from a model.

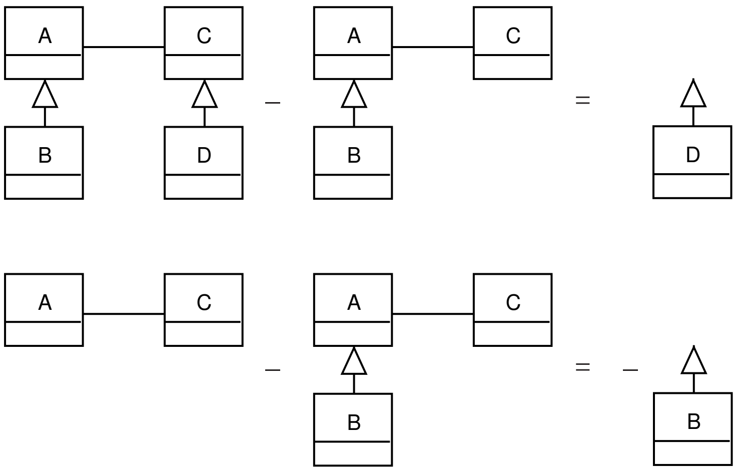


Figure 7.2: Example of the Difference of Models

We first show two algorithms to calculate the difference between two models and to merge a model with a difference. Once we know how to operate with differences between two models, we would like to solve our original problem by computing the union of two versions of a model as follows:

$$M_{\text{final}} = M_{\text{original}} + (M_1 - M_{\text{original}}) + (M_2 - M_{\text{original}})$$

Figure 7.3 shows this operation intuitively. In practise, the implementation of this operation is complicated by the fact that the two developers may have changed the same subset of the model. This situation can lead to *conflicts* and it may not be possible to apply all changes into the final model. We will discuss conflicts in more detail in Section 7.4.

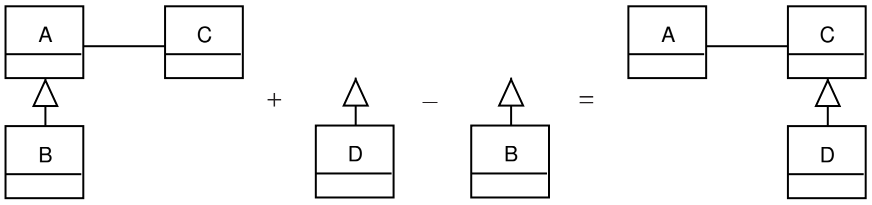


Figure 7.3: Example of the Union Based on Differences

The presented algorithms are implemented in a generic way, i.e., the algorithms are not defined in terms of a specific metamodel. However, these algorithms crucially rely on the existence of a universally unique identifier for each model element.

These algorithms are useful in many problems that appear in model management, especially in a distributed setting. An obvious application is a version control system with optimistic locking that allows many developers to work on a model simultaneously. Also, a model repository that stores different revisions of a model may store the difference between revisions instead of complete models, saving storage space. Similarly, two computers connected by a slow link may interchange a difference between models, saving bandwidth and communication time.

In the next sections, we reuse the formalization of models and metamodels as given in Chapter 3.

7.3 Difference Between Models

This section describes how to calculate the difference between two models, M_{old} and M_{new}. We represent the result of this operation as Δ. When we apply Δ in some specific way to M_{old}, we will acquire M_{new}.

We denote all differences to be the infinite set $\mathcal{D}$. Thus we would like to define the operation $\text{mdiff} : \mathcal{M} \times \mathcal{M} \to \mathcal{D}$ for calculating the difference between two models, and $\text{mpatch} : \mathcal{M} \times \mathcal{D} \to \mathcal{M}$ for merging a difference to a model, such that

$$\text{mpatch}(M_{\text{old}}, \text{mdiff}(M_{\text{old}}, M_{\text{new}})) = M_{\text{new}}$$

holds for two arbitrary models M_{old} and M_{new}. For clarity of presentation, we use plus and minus signs for these functions, although they do not share the usual properties of plus and minus on integers, as follows:

$$\begin{aligned} M_{\text{new}} - M_{\text{old}} &\overset{\text{def}}{=} \text{mdiff}(M_{\text{old}}, M_{\text{new}}) \\ M_{\text{old}} + \Delta &\overset{\text{def}}{=} \text{mpatch}(M_{\text{old}}, \Delta) \end{aligned}$$

In practical terms, one might consider the functions as the modeling equivalents of the Unix tools diff and patch on text files [111]. Considering our previous experience in using line-based version control systems such as CVS [34] or Subversion [150], an important operation is to be able to merge a difference *in reverse*, i.e., to apply a difference previously calculated as $\Delta = M_{\text{new}} - M_{\text{old}}$ to M_{new} in some way in order to acquire the old model M_{old}. Obviously $M_{\text{new}} + \Delta$ will not produce the desired result, since a prerequisite is that Δ only be applied to M_{old}. However, by suitably transforming Δ into its inverse $\tilde{\Delta}$, the straightforward equation $M_{\text{new}} + \tilde{\Delta} = M_{\text{old}}$ holds. The function $\text{diffinverse} : \mathcal{D} \to \mathcal{D}$ accomplishes this, and it can be noted that $\text{diffinverse}(\text{diffinverse}(\Delta)) = \Delta$ holds.

To support all this, we thus need to define the mdiff, mpatch and diffinverse functions.

7.3.1 Requirements on Metamodels

This section describes the requirements of the algorithms. In particular, we note that these requirements are supported by MOF and by our formalization from

Chapter 3, where we have covered the model and metamodel layer extensively. An important property characteristic concerning the algorithms in this chapter is *ordering*. An example of an ordered property is the sequence of parameters in a method.

We assume that each primitive datatype used in a metamodel has a default value, called the “zero” of the datatype. The default value of an integer is the value 0, the default value of a string is the empty string, and the default value of an enumeration is its first value (assuming a sequence of enumeration literals) or the empty (invalid) string, whichever is applicable in some specific modeling environment. The default value for the contents of an unordered and ordered set of elements are the empty set and empty sequence, respectively.

7.3.2 Description of a Difference

We have shown informally in the introduction that the result of the difference between two models may contain negative elements, i.e., information that should be removed from a model. We consider that it is intuitive to represent Δ in operational terms; not as a set of elements and negative elements but as a sequence of transformations that add, remove or modify elements in a model.

We have identified seven elementary transformations in three different categories that will be used as the basis for defining a Δ. These transformations are almost the same as have been discussed in Chapter 4, where we called them *basic edit operations*. The most important change here is that we need some additional information in them to be able to calculate the inverse, and that we do not maintain bidirectionality or subsetting at this level. They are maintained at a higher level in the actual difference algorithm. We also assume that it is not possible to change the type of a model element, e.g., a UML Class cannot become a Package, and an element cannot change its UUID. The different operations that can exist in a Δ are:

- Element creation and deletion.
  - create(u,t) : Create a new element of type t with the UUID u. By default, a new element has all its slots set to their default values.
  - delete(u,t) : Delete an element of type t with the UUID u. An element may only be deleted if all its slots are set to their default values, and its primitive value—if any—is a zero.
- Primitive value modification.
  - set(e,o,n) : Set the primitive value of element e from o to n. For ease of presentation, we assume all primitive values can be represented as strings.

- Modification of a slot.
  Modification of a slot of type p of an element e, either by inserting or removing another element e_t. Depending on p, this might mean one of the following modifications:

  – $\text{insert}(e, p, e_t)$: Add a link from $e.p$ to e_t, for an unordered slot.

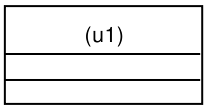
  – $\text{remove}(e, p, e_t)$: Remove a link from $e.p$ to e_t, for an unordered slot.

  – $\text{insertAt}(e, p, e_t, i)$: Add a link from $e.p$ to e_t, at index i, for an ordered slot.

  – $\text{removeAt}(e, p, e_t, i)$: Remove a link from $e.p$ to e_t, which is at index i, for an ordered slot.

These operations have two properties. First, the positive operations (create, set, insert and insertAt) are complete in the sense that they can be used to represent any model. Second, each operation has a dual operation with the opposite effect; this will be discussed further in Section 7.3.6.

An example of a difference between two models is given in Figure 7.4, in which the old model is on the left and the new model is on the right. In the Δ, two new elements u_2 and u_3 are created. They are connected to the root Model element u_0 (not shown) via their namespace association, and the Model connects them to its ownedElement composition, due to bidirectionality constraints. The new class u_2 is connected to the old class u_1 via the new Generalization element u_3, using its specialization and generalization slots.



$$\begin{aligned}\Delta = [\quad & \text{create}(u_2, \text{Class}), \\ & \text{create}(u_3, \text{Generalization}), \\ & \text{insert}(u_3, \text{namespace}, u_0), \\ & \text{insert}(u_3, \text{parent}, u_1), \\ & \text{insert}(u_3, \text{child}, u_2), \\ & \text{insert}(u_1, \text{specialization}, u_3), \\ & \text{insert}(u_0, \text{ownedElement}, u_2), \\ & \text{insert}(u_0, \text{ownedElement}, u_3), \\ & \text{insert}(u_2, \text{namespace}, u_0), \\ & \text{insert}(u_2, \text{generalization}, u_3) \quad]\end{aligned}$$

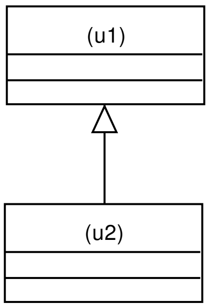


Figure 7.4: Difference Between Two Simple Models

7.3.3 Serialization

Operations cannot as such be saved to a disk or transferred over a network connection. We need to serialize them somehow. The solution is simple: instead of the actual element, we transfer its UUID. Also, properties and classes are encoded by their name. Thus, for example the command for inserting into a slot is not insertAt(e, p, e_t, i) but insertAt$(\text{uuid}(e), \text{name}(p), \text{uuid}(e_t), i)$ instead. This information must be suitably encoded depending on the actual serialization format used.

A serialization format with support for differences is XMI. It has facilities for representing arbitrary differences between two models, using an XML element called XMI.difference. The positive operations create, insert and insertAt can be described in an XMI document using the XMI.add element, while the negative operations delete, remove and removeAt can be specified using the XMI.delete element. Depending on the contents of a particular set operation, it will be in either the XMI.add or XMI.delete element. XMI also specifies that differences must be applied in the order defined, which is also a requirement of the algorithms in this chapter.

However, the semantics of XMI.difference imply unnecessary duplication of information in some cases, and sometimes extra, unnecessary information must be included. Also, very few actual usage reports have been published. A paper by Annika Wagner notices that we do not wish to describe complete elements or model fragments inside XMI.difference, and that we would rather want to omit some information [200]. Jernej Kovse and Theo Härder did not find any problems in using XMI.difference, but the context of their work was specifying templatized model transformations [96].

A better approach, and arguably within the spirit of the modeling technical space, would be to describe the difference between two models as a model itself.

7.3.4 Calculating the Difference

Once we know how to represent the difference between two models, we can describe an algorithm to calculate it. The proposed difference algorithm has four steps, as discussed in [37]. The objective is first to create an unambiguous bijective mapping between the elements in M_{old} and in M_{new} and then calculate an exact sequence of operations that can transform each element in M_{old} to the corresponding one in M_{new}. The phases Map, Create and Delete create the bijective mapping, whereas the Change phase handles the transformations.

The bijective mapping is done with two functions $E_{\text{old}} : \mathcal{U} \nrightarrow E$ and $E_{\text{new}} : \mathcal{U} \nrightarrow E$ that map UUIDs to model elements in M_{old} and M_{new}, respectively.

Map

This phase creates a mapping between elements in M_{old} and M_{new}. In our case, the UUIDs of the elements serve as the map. From this, we create $E_{\text{old}}(u) = e$ such that u is the UUID of $e \in M_{\text{old}}$ and $E_{\text{new}}(u) = e$ such that u is the UUID of $e \in M_{\text{new}}$.

It is possible to create such a mapping without relying on the UUIDs, but it would be a lot more difficult and resource-intensive. However, it could yield a smaller Δ.

Create

The Δ should contain an operation to create each element in M_{new} that does not exist in M_{old}. Given our mapping between elements in M_{old} and M_{new}, we define the sequence δ_C of elements that need to be created as follows:

$$\delta_C := [\,\text{create}(u,t)\ \cdot\ (\forall u,t\ \cdot\ u \in \text{Dom}(E_{\text{new}}) \setminus \text{Dom}(E_{\text{old}}) \wedge t = \text{typeof}(E_{\text{new}}(u)))\,]$$

Delete

Similarly, the Δ should contain an operation to delete each element in M_{old} that does not exist in M_{new}. The sequence δ_D of elements that need to be deleted is easy to define:

$$\delta_D := [\,\text{delete}(u,t)\ \cdot\ (\forall u,t\ \cdot\ u \in \text{Dom}(E_{\text{old}}) \setminus \text{Dom}(E_{\text{new}}) \wedge t = \text{typeof}(E_{\text{old}}(u)))\,]$$

After defining the sequences δ_C and δ_D, it is necessary to update the map between M_{old} and M_{new} to be bijective. This is accomplished by updating E_{old} and E_{new} to have the same domain, by adding new elements into E_{old} which are in E_{new} but not in E_{old}, and vice versa. The created elements have default values for all their slots. Now there exists a set of tuples (e_o, e_n) such that for each element $e_o \in \text{Range}(E_{\text{old}})$ there exists an element $e_n \in \text{Range}(E_{\text{new}})$ such that $\text{uuid}(e_o) = \text{uuid}(e_n)$, and vice versa.

Note that the sets E_{old} and E_{new} now in fact contain the same number of elements, and that we can pair the elements with the same UUID. This is a special case, in that we have assumed the UUID of an element to be unique; this is only true within either E_{old} or E_{new}.

Change

In this phase we match the slots for each pair of elements (e_o, e_n), both with the UUID u. Since both elements have the same type, their slots must conform to the same set of properties. Thus we have a set of pairs (s_o, s_n) for each property in $\text{effectiveProperties}(\text{class}(e_o))$, such that s_o is a slot in e_o, s_n is a slot in e_n and $\text{property}(s_o) = \text{property}(s_n)$. The task is to create operations that modify the contents of s_o into the contents of s_n. We collect all changes from all slots of all elements into a sequence δ_F, based on the following:

- For an unordered slot, create the following operations:
  $[\text{insert}(e,p,e_t)\ \cdot\ (\forall e_t\ \cdot\ e_t \in e_n.p \setminus e_o.p))] \triangleleft [(\text{remove}(e,p,e_t)\ \cdot\ (\forall e_t\ \cdot\ e_t \in e_o.p \setminus e_n.p)]$.

- For an ordered slot, the order of the contents must be preserved. The smallest sequence of changes that transforms one sequence into another is equivalent with the *Longest Common Subsequence* problem, to which there exists efficient solutions, e.g., by Myers [120]. The result is a sequence of insertAt and removeAt operations which, when applied to s_o, transforms it into s_n at minimal size cost. The sequence has length $\#s_o + \#s_n - 2L$, where L is the length of the longest common subsequence of s_o and s_n. The operations in the sequence should be added to δ_F. We omit a solution to the longest common subsequence problem, as any one of several suitable ones can be used with some performance differences; the details are not relevant since the result is the same regardless of the algorithm used.

Additionally, we may need to modify the primitive value of an element. We do this by visiting all pairs of elements (e_o, e_n) in the bijection. These changes are also added to the δ_F sequence.

- If e_o has a primitive value which is different in e_n, we must create a command to change the primitive value:
  $[\,\text{set}(e_o, \text{primitivevalue}(e_o), \text{primitivevalue}(e_n)) \cdot e_o \in \text{Dom}(\text{primitivevalue}) \wedge \text{primitivevalue}(e_o) \neq \text{primitivevalue}(e_n)\,]$.

A complete Δ between two models is then specified by a sequence of all operations as created by the above algorithm. This is a sequence of element creations, set commands and slot modifications, and element deletions, in that order, and so $\Delta = \delta_C \triangleleft \delta_F \triangleleft \delta_D$. Such a Δ should guarantee two properties:

- For each delete operation, there is a sequence of operations before the delete operation that resets the slots of the elements to be deleted to their default value.

- The complete sequence of operations maintains the bidirectional relations and subsetting in a consistent state. The individual operations only update one slot, and do not try to maintain subsets. However, we should ensure that for each operation that updates a slot, there is a corresponding operation that updates the opposite slot, and that no subset constraints are violated.

The algorithm proposed in this section satisfies these properties.

7.3.5 Applying a Difference

To merge a difference to a model is to apply the transformations contained in a Δ to a model.

We assume we are given a $\Delta = \delta_C \triangleleft \delta_F \triangleleft \delta_D$ and a model $M = (E, \text{type}, \text{slots}, S, \text{property}, \text{elements})$. We will use slightly modified model operations from Chapter 4 such that any subset/superset slot is not affected by any insert^u, insert^o or remove operation; these are marked with an asterisk.

The merge algorithm can be described by the following three steps:

1. $\forall \text{create}(u,t) \in \Delta$: Create an element of type t with the UUID u.

$$\begin{aligned} M', e &:= \text{create}(M,t) \\ \text{uuid}' &:= \text{uuid}[e \to u] \end{aligned}$$

2. $\forall f \in \delta_F$: Make the change f. It is crucial that subsetting and bidirectionality is ignored at this level, since there will be separate commands in δ_F that handle both of these concepts. Given that our model is $M = (E, \text{type}, \text{slots}, S, \text{property}, \text{elements})$, we have:

$$\begin{aligned}
\text{For set}(e,o,n) :& \\
\text{primitivevalue}' &:= \text{primitivevalue}[e \to n] \\
\text{For insert}(e,p,e_t) :& \\
\text{elements}' &:= \text{elements}[e.p \to \text{elements}(e.p) \cup \{e_t\}] \\
\text{For insertAt}(e,p,e_t,i) :& \\
\text{elements}' &:= \text{elements}[e.p \to \\
&\qquad \text{elements}(e.p)[0:i] \\
&\qquad \lhd [e_t] \\
&\qquad \lhd \text{elements}(e.p)[i: \ \#\text{elements}(e.p)\ ]] \\
\text{For remove}(e,p,e_t) :& \\
\text{elements}' &:= \text{elements}[e.p \to \text{elements}(e.p) \setminus \{e_t\}] \\
\text{For removeAt}(e,p,e_t,i) :& \\
\text{elements}' &:= \text{elements}[e.p \to \\
&\qquad \text{elements}(e.p)[0:i] \\
&\qquad \lhd \text{elements}(e.p)[i+1: \ \#\text{elements}(e.p)\ ]]
\end{aligned}$$

3. $\forall \text{delete}(u,t) \in \Delta$: Delete the element e of type t with the UUID u.

$$\begin{aligned} M' &:= \text{delete}(M,e) \quad (\text{Given uuid}(e) = u) \\ \text{uuid}' &:= \{e\} \blacktriangleleft \text{uuid} \end{aligned}$$

The actual implementation of these transformations depends on the action language used to transform the models. A requirement of a metamodel is that we have a reflection interface for determining and querying the properties of all classes, and a facility for modifying the slots of model elements.

7.3.6 Inverse of a Difference

Given two models M_{old} and M_{new} and a difference $\Delta = \delta_C \lhd \delta_F \lhd \delta_D$ such that $M_{\text{old}} + \Delta = M_{\text{new}}$, we can calculate the inverse of a difference $\tilde{\Delta}$ such that $M_{\text{new}} + \tilde{\Delta} = M_{\text{old}}$. It is useful, for example if we wish to (temporarily) back out a difference for testing purposes.

Operation O	Dual operation $\tilde{O}$
create(u,t)	delete(u,t)
delete(u,t)	create(u,t)
set(e,o,n)	set(e,n,o)
insert(e,p,e_t)	remove(e,p,e_t)
remove(e,p,e_t)	insert(e,p,e_t)
insertAt(e,p,e_t,i)	removeAt(e,p,e_t,i)
removeAt(e,p,e_t,i)	insertAt(e,p,e_t,i)

Table 7.1: The Map Between Operations and Dual Operations

The map between operations and their dual operations is given in Table 7.1, and we assume there is a function dual which maps operations from the left column to the corresponding dual operation in the right column. It is needed to calculate the inverse of a Δ. It also explains why e.g. delete(u,t) needs to have the type t as a parameter, even though that is not necessary for deleting an element. Without it, it would not be possible to deduce the dual operation create(u,t).

Calculating the inverse of a difference is a simple process that only requires us to reverse the sequence of operations in Δ and replace each operation with its dual:

$$\begin{aligned}
\tilde{\Delta} &= \tilde{\delta}_C \triangleleft \tilde{\delta}_F \triangleleft \tilde{\delta}_D \\
\tilde{\delta}_C &= [\, \mathrm{dual}(d) \cdot d \in \mathrm{reverse}(\delta_D) \,] \\
\tilde{\delta}_F &= [\, \mathrm{dual}(f) \cdot f \in \mathrm{reverse}(\delta_F) \,] \\
\tilde{\delta}_D &= [\, \mathrm{dual}(c) \cdot c \in \mathrm{reverse}(\delta_C) \,]
\end{aligned}$$

Once we have calculated the inverse of a Δ, it can be applied as described in Section 7.3.5.

7.4 Merging of Models

An interesting problem emerges when two differences, Δ_1 and Δ_2, should be applied onto the same model. This occurs frequently in a distributed development environment.

The objective is to apply Δ_1 first, then apply Δ_2. It should be noted that the result ought to be the same regardless of the order in which the differences are applied:

$$M_{\mathrm{final}} = M_{\mathrm{original}} + \Delta_1 + \Delta_2 = M_{\mathrm{original}} + \Delta_2 + \Delta_1$$

However, since the differences are calculated relative to M_{original}, applying one Δ first would create a model which is different from the base model, and the other Δ might not be applicable as such. The base case when both deltas can be applied

$$\begin{tikzcd}
& M_{\text{original}} \arrow[dl, "\Delta_1"'] \arrow[dr, "\Delta_2"] & \\
M_1 \arrow[dr, "\Delta'_2"'] & & M_2 \arrow[dl, "\Delta'_1"] \\
& M_{\text{final}} &
\end{tikzcd}$$

Figure 7.5: The Principle of Calculating the Final Model of Two Models and Their Original Model

one after the other is only possible when they do not modify the same slot of the same element; otherwise we say that a *conflict* has occurred.

7.4.1 Conflicts

Consider an ordered slot with elements $[B,C]$ and differences insertAt$(A,0)$ and insertAt$(D,2)$. Applying the differences would create either the correct $[A,B,C,D]$ or the incorrect $[A,B,D,C]$, depending on the order in which the differences were applied. In private communication with Ralph-Johan Back, the intuition on why the former is correct relies on our understanding that we do not wish to insert an element into a particular index, but rather into an index relative to other elements.

Another example is a set $\{A,B\}$ and differences insert(C) and insert(C). The second operation is spurious, since the first difference already accomplishes the task: adding C to the set. Removing the other operation shortens Δ. In this case the conflict is much less serious since adding an element already in a set does not change the set, i.e., we cannot modify the slot in an erroneous way.

Nevertheless, we need a reliable method to modify a Δ according to another Δ, and a shorter Δ is naturally preferred. The modification is necessary to avoid errors. We define the *difference minimization operator* $\otimes$:

$$\Delta'_2 = \otimes(\Delta_2,\Delta_1), \quad \Delta'_1 = \otimes(\Delta_1,\Delta_2)$$

Now the equation becomes:

$$M_{\text{final}} = M_{\text{original}} + \Delta_1 + \Delta'_2 = M_{\text{original}} + \Delta_2 + \Delta'_1$$

This principle is also illustrated in Figure 7.5. Without loss of generality, this chapter discusses only the calculation of $\Delta'_2 = \otimes(\delta_{C_2} \triangleleft \delta_{F_2} \triangleleft \delta_{D_2}, \delta_{C_1} \triangleleft \delta_{F_1} \triangleleft \delta_{D_1})$. We also provide a *conflict detection operator* $\times$ which returns true if operations conflict with each other.

7.4.2 Conflict Resolution

The calculation of $\delta'_{C_2} \triangleleft \delta'_{F_2} \triangleleft \delta'_{D_2} = \otimes(\delta_{C_2} \triangleleft \delta_{F_2} \triangleleft \delta_{D_2}, \delta_{C_1} \triangleleft \delta_{F_1} \triangleleft \delta_{D_1})$ has several different cases: element creation and deletion, changes to the primitive values, unordered and ordered slots. Given our method of modifying each slot of each element with a small sequence of operations, the calculation of δ'_{F_2} also happens one slot at a time. The difference minimization methods of the various kinds of operations, and which conflicts can occur, are presented in the following subsections. A common resolution is to ignore an operation $o \in \delta_{F_2}$ instead of adding it to δ'_{F_2}. This also implies that for bidirectional relations, the operation for the opposite slot must have the same resolution as o: properly ignored or added to δ'_{F_2}.

Element Creation and Deletion

The set of elements created in Δ_2 are unaffected by any operations in Δ_1. Due to the uniqueness of the UUID generator, the operations in Δ_1 cannot refer to the new elements in Δ_2. Therefore, all $\text{create}(u,t)$ operations in Δ_2 are valid.

Element deletion can be a source of conflicts. For each $\text{delete}(u,t)$ operation in Δ_2, if Δ_1 also has the same operation, we can remove it from Δ_2, since Δ_1 will already delete the element.

$$\begin{aligned} \delta'_{C_2} &= \delta_{C_2} \\ \delta'_{D_2} &= [o \cdot o \in \delta_{D_2} \wedge o \notin \delta_{D_1}] \end{aligned}$$

The worst case occurs when one Δ modifies e by adding elements to its slots, and the other Δ has an operation for deleting it. One solution would be to ignore the modifications and delete the element. However, it is questionable if this is the correct behavior and has the intended effect every time, so manual resolution might be the only viable choice. Thus the conflict detection is the following function:

$$\begin{aligned} \times(\delta_{D_1}, \delta_{D_2}) = \; & (\exists \text{delete}(\text{uuid}(e),t) \in \delta_{D_2} \\ & \wedge((\exists \text{set}(e,o,n) \in \delta_{D_1} \wedge n \text{ is not a zero}) \\ & \quad \vee (\exists \text{insert}(e,p,e_t) \in \delta_{D_1}) \\ & \quad \vee (\exists \text{insertAt}(e,p,e_t,i) \in \delta_{D_1}))) \\ & \vee \\ & (\exists \text{delete}(\text{uuid}(e),t) \in \delta_{D_1} \\ & \wedge((\exists \text{set}(e,o,n) \in \delta_{D_2} \wedge n \text{ is not a zero}) \\ & \quad \vee (\exists \text{insert}(e,p,e_t) \in \delta_{D_2}) \\ & \quad \vee (\exists \text{insertAt}(e,p,e_t,i) \in \delta_{D_2}))) \end{aligned}$$

Primitive Values

An element can represent a primitive value, depending on its type. Due to this, a conflict situation occurs whenever both δ_{F_1} and δ_{F_2} change the primitive value of the same element. Trivially the difference minimization retains primitive value changes in δ_{F_2} unless the primitive value of the same element is already changed in δ_{F_1}:

$$\otimes(\delta_{F_1},\delta_{F_2}) = (\,\mathrm{set}(e,o,n_2)\,\cdot\,\mathrm{set}(e,o,n_2)\in\delta_{F_2}\wedge\neg(\exists\mathrm{set}(e,o,n_1)\in\delta_{F_1}))$$

Conflict detection checks for two different changes in δ_{F_1} and δ_{F_2} to the primitive value of an element:

$$\times(\delta_{F_1},\delta_{F_2}) = (\exists\mathrm{set}(e,o,n_2)\in\delta_{F_2},\mathrm{set}(e,o,n_1)\in\delta_{F_1}\wedge n_1\neq n_2)$$

Unordered Slots

It is easy to see that double insertions into or double removals from an unordered set are harmless, but create an unnecessarily long δ'_{F_2}. Therefore, we can say that each remove(e,f,e_t) operation in δ_{F_2} can be added to δ'_{F_2} if it does not exist in δ_{F_1}. Each insert(e,f,e_t) operation in δ_{F_2} can be added to δ'_{F_2} if it does not exist in δ_{F_1}. For bidirectional relations, it is important to only add these operations to δ'_{F_2} if a similar operation can be added to δ'_{F_2} for the opposite slot, which can be unordered or ordered.

If there are no multiplicity constraints on the slot, no conflicts are possible. In particular, many slots have a multiplicity constraint of “at most one element”. If both differences add an element to such a slot, manual resolution must choose which one of the operations should prevail, but otherwise the following suffices:

$$\otimes(\delta_{F_1},\delta_{F_2}) = [\,o\,\cdot\,o\in\delta_{F_2}\wedge o\notin\delta_{F_1}\wedge(o\text{ is remove}\vee o\text{ is insert})\,]$$

There is no particular conflict detection function, since the above function suffices and removes all conflicts.

Ordered Slots

The idea of difference minimization for ordered slots is to interleave the insertAt and removeAt operations in Δ_2 with the ones in Δ_1. This is a similar operation that the Unix tools diff3 and merge already accomplish on text files [111] and will not be discussed further.

The most common conflict occurs when both differences insert different elements at the same index. Of importance is also to remember that the resulting sequence cannot contain the same element twice.

Figure 7.6 shows an example output for interleaving ordered features. Note how the insertion of B at index 1 pushes the insertion of D from index 2 to index 3.

$$\begin{tikzcd}
& {[\,A,C\,]} \arrow[dl, "{\delta_{F_1} = [\,\mathrm{insertAt}(B,1)\,]}"'] \arrow[dr, "{\delta_{F_2} = [\,\mathrm{insertAt}(D,2)\,]}"] & \\
{[\,A,B,C\,]} \arrow[dr, "{\delta'_{F_2} = [\,\mathrm{insertAt}(D,3)\,]}"'] & & {[\,A,C,D\,]} \arrow[dl, "{\delta'_{F_1} = [\,\mathrm{insertAt}(B,1)\,]}"] \\
& {[\,A,B,C,D\,]} &
\end{tikzcd}$$

Figure 7.6: Example of a Merge of an Ordered Sequence

7.4.3 A Complete Merging System

Metamodels have, in addition to the constraints expressible by MOF, a set of well-formed rules (WFR) which determine if a model is a valid instance of the metamodel. On a metamodel-independent level, the WFRs of a metamodel cannot be kept. Therefore, even a successful, conflict-free union can still be invalid by the rules of the metamodel. In these cases manual resolution may seem like the only choice, but metamodel-specific resolution may also automatically resolve some of the problems by analyzing the resulting nonwellformed model, and modifying it to a wellformed state.

One example of a metamodel-specific resolution mechanism presents itself with merging diagram information. The diagram elements themselves do not have any semantic meaning, so the information of the diagram elements is not nearly as correctness-critical as that of the underlying abstract model. For example, conflicting diagram element coordinates on the diagram canvas can more or less be ignored by removing or modifying the relevant operations from Δ_2. Clearly, there is a strong need for metamodel-specific resolvers, perhaps the prime candidate being a conflict resolver for Diagram Interchange diagrams.

The schema in Figure 7.7 summarizes a merging system for models, with the original model M_{original} being modified by two differences Δ_1 and Δ_2. The difference under modification, Δ_2, passes through several filters which modify it to better fit $M_{\text{original}} + \Delta_1$. Obviously, all possible mechanic resolution mechanisms should be tried before manual resolution is used. The algorithms described in this chapter work as the first, metamodel-independent filter.

The merging algorithm as given works with two differences created in parallel, but can be extended further. Given a base model M_{original} and n differences $\Delta_1, \Delta_2, \ldots, \Delta_n$, we notice that the number of differences can be reduced by taking the union $M = M_{\text{original}} + \Delta_1 + \Delta'_2$, and calculating a difference $\Delta_{1'} = M - M_{\text{original}}$. Now we have the same base model M_{original} and $n-1$ differences $\Delta_{1'}, \Delta_3, \Delta_4, \ldots, \Delta_n$. Iterating through this algorithm we have the final model M_{final}. This is important in a version control system for models, where several developers base their work on some common base model, and later commit it back to the repository, merging their changes with the work of others.

$$\begin{tikzcd}
\Delta_2 \arrow[d] \\
\text{Metamodel-independent resolver} \arrow[d] \\
\Delta_2' \arrow[d] \\
\text{Metamodel-dependent resolver} \arrow[d] \\
\Delta_2'' \arrow[d] \\
\text{Manual resolution} \arrow[d] \\
\Delta_2''' \arrow[d] \\
\text{Calculate } M_{\text{original}} + \Delta_1 + \Delta_2'''
\end{tikzcd}$$

Figure 7.7: A Merging System with Three Distinct Resolution Steps

The above mechanism also emphasizes the scalability of the approach, as an arbitrary number of parallel differences can be merged. It remains to be investigated if it is feasible in a practical context for models, or if a merging algorithm is required which would consider more differences in parallel [115, 147].

7.5 A Model Repository

The task of a model repository is to store successive versions of a model and retain old versions. A simple model repository can store each version of a model as a different file containing the model as an XMI document. In such a system, the file name can be used to identify each version of a model in the version control system. Access control to the repository is managed by the access control mechanism of the filesystem.

This simple repository is too coarse-grained for most practical uses. It also lacks many important features. We may want to use the repository to keep a history of the evolution of a model through the whole development cycle. In this case, it is important that we are able to identify, version and retrieve each individual element in a model.

7.5.1 Model Storage

While a filesystem still can be used for all this, it is not efficient and we can quickly run into problems with respect to atomicity and concurrency. An alternative is to store models into a database. A database can set arbitrary rules for access, modification and retrieval and transactional properties such as atomicity, consistency, isolation and durability (ACID) are guaranteed by the database engine. Additionally, it is easy to store additional metadata of the models.

The upside in using relational databases is that they have been researched very thoroughly and industry has greatly invested in creating highly scalable and efficient products. The downside is that model information is inherently object-oriented and it does not map naturally into the relational model. This is also known as the impedance mismatch between the object-oriented and the relational paradigm [113, 99].

The advent of XML has spurred research in databases particularly suited for storing large XML documents [80]. XML repositories are very similar to object-oriented databases (OODBs), and share their benefits and ills. Among the benefits are much more flexible arrangements of data, ways to manipulate that data and more complicated queries. However, current technology does not scale as well as relational databases. Especially query optimization is not as well-known as in the relational database field. Using an XML database itself could be a great advantage, but until technology catches up, it cannot be deployed for large-scale projects. Also XML and object-orientation exhibit an impedance mismatch [114, 100].

Next, we will have a closer look at a relational database schema required for a model repository. It serves merely as an example skeleton of how to map hierarchical information to relational space, and to show that despite the impedance mismatch we can still use relational databases as the storage mechanism for versioning models.

Relational Database Schema

Formally, a relational database consists of relations, tuples and attributes. Each relation is defined by its name and its named attributes. A tuple is a record of the database, i.e., the actual data we are storing. The data can be cross-linked between several relations. Retrieval consists of fetching the tuples matching a certain query, often speeded up by matching attributes which are *primary keys*, i.e., unique values in the relation. The problem of storing models in a relational database is fundamentally about mapping a graph with different kinds of nodes and edges to a relational model.

The relational database schema consists of a static set of relations independent of the metamodels used, and a set of relations for each class in every metamodel used. The static set consists of database tables that maintain the version history of the models and enables arbitrary elements to connect to other elements, whereas the rest are purely containers of primitive model data such as strings, integers or enumeration values.

There are two strategies to store the different versions of a model element. The first one is to store each individual version as a different element including all its attributes. The second alternative is based on the previous discussion on differences between models. We can store only the difference between two versions of a model instead of the complete model elements.

If we store complete model elements, the database will require more space. If we store only the differences between revisions of an element, the size of the database will be smaller, but the queries will be more complex and require more processor time. To simplify our exposition, we present only a database schema where model elements are stored completely. We also do not discuss tables required for a full repository implementation requiring for example transaction histories, branching and access control, and omit various string encoding rules.

To clarify, the database tables that we are about to discuss in this section have been collected below. Where possible, the database has received hints of which columns can be indexed, made primary keys, or reference other tables. Indexes group together a set of rows with the same column data, primary keys define uniqueness of rows in a table and references provide integrity between the various database tables. These common database layout enhancements speed up queries considerably, and provides some referential integrity in the database. They are marked with the keywords **indexed**, **primary** and **references** `tablename`:

- Model = (model_revision **indexed**, element_revision)
- ElementType = (UUID **primary**, type **references** `ClassNames`)
- RevisionUuid = (revision **primary**, UUID)
- Connections = (revision **indexed**, name **references** `ConnectionNames`, target, index)
- ConnectionNames = (name **primary**)
- ClassNames = (name **primary**)
- Additionally, one table for each metamodel class or enumeration is required.

In the database schema, the table Model consists of a map between model revision to element revisions. The ElementType table contains a row for each model element in a model. It has two columns, the UUID of the model element and the type of the model element. We assume that an element cannot change its type. Each revision of an element receives a revision number which is unique for the repository. To know what UUID a revision has, we must keep a RevisionUuid table which maps revisions to UUIDs.

A revision of an element consists of data of primitive type, and of links to other elements. These are collected into two different tables, of which one is specific to the metamodel type, and the other is generic. The name of the specific table is created based on the full name of the modeling language and the name of the class.

revision	**name**	**visibility**	**isAbstract**	. . .
5	Class1	public	false	. . .
9	Class2	protected	true	. . .
13	Class3	private	true	. . .

Table 7.2: An Extract from an Elements_UML14_Class Table

For example, the table Elements_UML14_SimpleState is used to store SimpleState elements from the UML 1.4 language, and Elements_UML14_ is just a prefix to avoid name collisions with other tables unrelated to the UML 1.4 language. Each row in this table will represent a specific revision of a model element of that given class. The primary key of the table is the revision number of the model element. The other attributes are the properties of primitive type of the element. Another example of a metamodel-specific table can be seen in Table 7.2.

The generic table is called Connections and maps the connections between elements. Since a slot of an element contains a set or sequence of references to elements, it is important to be able to remember the position of elements for ordered slots. Thus, Connections is defined as (revision, name, target, index). Here, revision is the repository-unique version identifier of our element. Name represents the name of the property, e.g., ownedElement for a Package owning several classes. Curiously, target must reference the UUID of the target element, not its repository version identifier. This is inherent in the bidirectionality of slots in most modeling languages. If we had two interconnected elements A and B and repository revisions were used exclusively, and A were to change, its revision identifier changes, and thus the database tuples of B would change, resulting in a change of its revision identifier as well! This would cascade through the whole model and create new revisions of every element in the model. This is clearly not desired, and thus links must be from a revision to a UUID. Then RevisionUuid and Model can be used to retrieve the actual revision to use. The final column, index, simply keeps track of which element should be in which position in the slot. This is required since few (if any) relational databases keep their records in order. An example of a Connections table can be seen in Table 7.3.

Naturally, there are several more ways to encode the same information, and several optimizations that could be made, especially space-wise. For example, since the set of properties is known, we can also have separate Connections tables for each, in this case for example Connections_UML14_Package_ownedElement, thus making some of the information implicit in the table name. Also, a trivial optimization is to split the Connections table in two, one for all unordered properties (thereby making the fourth attribute index unnecessary), and one for all ordered properties. The main drawback of these optimizations is that the number of database commands that must be used increases, and thus there is a risk for an effective slowdown of the repository. This, however, can only be determined by

revision	name	target	index
1	parameter	E2	0
1	parameter	E3	2
1	parameter	E4	1
2	behavioralFeature	E1	-1
3	behavioralFeature	E1	-1
4	behavioralFeature	E1	-1

Table 7.3: An Extract from a Connections Table

performing empirical tests since the actual performance varies from one database to another.

In order to aid in debugging the database, several string constants are kept in some tables, and rows of the other tables reference these strings. Each enumeration is kept in separate table, e.g., Enum_UML14_VisibilityKind, and the enumeration strings for attributes in the element tables keep references to those strings. Thus, the user debugging sees the actual enumeration strings instead of obscure enumeration numbers, while still keeping the memory requirements low. All element names and all property names are also kept in separate tables, ConnectionNames and ClassNames, for the same reason.

7.5.2 Access Control

Access control defines the mechanisms by which read or write access to parts of the model are defined and modified. Access control can be implemented at different levels of granularity. The access control level that is easiest to understand and implement is access control at the model level. In such a mechanism the repository may grant read-only or read and write rights to a whole model for a specific set of developers.

However, in some projects, limiting access for developers to only some parts of a model may be important, or even mandatory. As an example of limiting read access, security-related information is to be disclosed only to a specific set of developers. A more common scenario is limiting write access, such that a group of developers may work on a part of a model, and another group on another part. In such cases, it might feel intuitive to set the granularity of access at the element level, whereby read or write access is determined based on the elements that a developer wants to read or change. However, this may be impossible due to bidirectionality. Modifying a slot implies the modification of the opposite slot.

For example, consider a class which has write restrictions. In UML, this class is represented as different properties, including its name, attributes, superclasses and subclasses. It is then impossible to create a new class as a subclass of this write-restricted class, since that requires modification of the specialization property of that class—which the developer is not allowed to modify! However, most developers would consider these changes harmless to the original class. This is because while some properties carry semantic meaning for an element, other properties only act as a navigational aid.

Clearly, the level of detail in access control must be based on the properties of elements, not on the elements themselves. In some cases, the developer ought to be able to use a class, subclass it but not add new operations or change existing attributes. The distinction cannot be made by allowing or disallowing write access to the class element itself, but to the properties of the class.

7.5.3 Client-Server Communication

Most repositories are usually centralized systems that allow different developers to work simultaneously on the same project. In this case, the repository resides on (at least) one server and the client computers read and update parts of the repository as needed. We would like the communication between the repository and the client to be based on open standards so we can seamlessly use tools from different vendors. This includes two different standards: a standard to define how and when information is transferred and a standard to define the format of the information that is being transferred. XMI as such does not define a protocol for transferring models over a network, only the encoding of a model. Special interest groups, separate from OMG, are advancing the state of the art of distributed authoring, and are creating official Internet standards to fill this void. Good examples are the IETF WebDAV and Delta-V working groups, which have defined "HTTP Extensions for Distributed Authoring—WEBDAV" (RFC 2518) [62] and "Versioning Extensions to WebDAV" (RFC 3253) [44] to ease communication in a distributed development environment. These standards can be used for a protocol between a model repository and the client tools, such as a UML editor. It is beneficial to examine in more detail what operations especially related to models these relatively new protocols should provide in order to become de facto standards of model transportation.

Many times a client will not be interested in all the elements in a model but only in a subset of them. The problem is that a client might not know the name or the UUID of a certain model element in which it is interested. The reason the client should not download the whole model is due to the comparatively low bandwidth between the client and the server. There are two main solutions to this problem: one is to let a client seek elements in the model and the other is to implement a query language.

In the first solution the server should provide a simple interface with two services: one service, named *getRoot*, returns the UUID of the root element in a model, while the second service, *getElement*, accepts a UUID as a parameter and returns the model element associated to it. As an example, we can assume our repository contains a simple UML model with two packages and one class as shown in Figure 7.8. For simplicity, we use integers to denote the UUIDs.

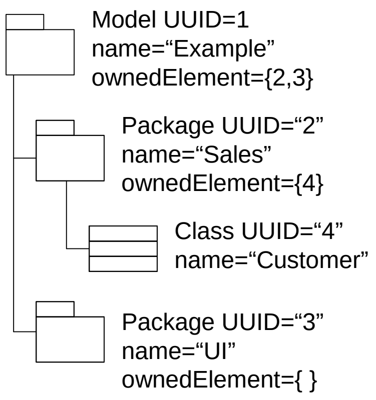


Figure 7.8: Example Repository Client Access

In the example repository, if a client invokes the service *getRoot* the repository will return the value "1", i.e., the UUID of the main element in the repository. If the client invokes *getElement("1")*, the repository will return all the properties of the element with UUID "1", including the property named ownedElement. In UML, this property describes the contents of a model or a package. In the repository the model contains the packages Sales and UI, therefore the ownedElement property will be the set $\{ 2, 3 \}$, i.e., the UUIDs of the previous packages. The client can use these UUIDs to continue traversing the models. This interface can be naturally extended to include revision numbers, for example *getRoot(5)* will return the root element in the 5th revision of the model.

An alternative solution is to use a query language, something akin to the SQL in the world of relational databases. In this case, the client will send a query as a text string to the repository that will evaluate the query against all elements in the model and return those which satisfy it. We can use different alternatives as a query language. OCL [203] is used in the UML metamodel to define additional constraints over valid UML model elements, but it can also be used as a query language. As an example, if a client sends the query *self.oclIsKindOf(Class) and self.name="Customer"* to our example repository, the repository will return the set $\{ 4 \}$. The result is a set since there can be more than one class with the name Customer.

We would have to extend the current OCL standard with queries to retrieve version information so we can perform queries against the version history of the repository such as the following:

```
self.name = "Customer" and self.lastEdited < "24 Dec 2006"
```

which returns a set of all those elements with a name of "Customer" which were edited before the specified date.

An alternative to OCL is to use some other query language based on XML, such as XQuery [199]. However, this approach neither solves the need to know how the model information is arranged in the UML metamodel in order to create a complex query, nor does it provide a direct solution to querying version information.

7.6 Validation of Research

Considering only technical issues, the most basic problems in version control systems are in calculating differences between models, and merging one or several parallel differences together into a final model. As a proof of concept, the algorithms in this paper have been used to create a simple centralized SQL model repository server [5] using XML Remote Procedure Calls (RPC) [179] over HTTP [56].

The server supported describing model updates to clients using the difference algorithm, and the clients used the merge algorithm to include the update into the client code, while keeping client changes intact. The commits to the server were done similarly.

Primitive automatic conflict resolution in the form of a metamodel-independent resolver was included. There was no metamodel-dependent resolver part; diagrams unfortunately were not described as models so they could never be stored into the repository. The metamodel-dependent step can also be considered as a quality-of-implementation issue, since a manual resolution mechanism should still catch any remaining conflicts. In the implementation, even a crude graphical interface for the manual resolver was never finished; usability was thus a major concern.

Nevertheless, all the algorithms presented here were part of the System Modeling Workbench by Ivan Porres et al. [11, 152]. Currently, the difference algorithm has also been implemented in the Coral modeling framework, with a metamodel specifically for describing model differences.

7.7 Related Work

As described by Yuehua Lin, Jing Zhang and Jeff Gray in [105], the topic of versioning models is very important. During the recent years there has been some research in the area, mostly by combining algorithms from graph (tree) and word/text theory, as we also have done. Furthermore we can split the research into different areas, depending whether or not order of elements are taken into account, and whether the elements need persistent identifiers.

Ignoring the order criteria leads to very fast change detection for hierarchical information [202], and this has also been researched by [212]. While this ignorance is convenient it is too lax, as the order of element references in slots is not kept.

Since XMI is an XML application, one could assume that ordinary XML merging tools could be used. However, they are too strict since all nodes in XML have to be kept in the correct order. A good tool must take into account ordering only when necessary, in order to minimize manual resolution.

Rather than using persistent identifiers for the mapping phase as described in Section 7.3.4, we could use some other mechanism. Other possibilities include matching by element name or some more complicated similarity function [210, 87]. However, such results are heuristics and might thus produce erroneous mappings, but they might be necessary since UUIDs might not always be available.

Cicchetti et al. have described a model difference metamodel [40] and a way to compose several difference models [41]. However, the details of how this model weaving is accomplished are not discussed.

Prawee Sriplakich et al. [165] also describe algorithms for calculating the difference and merging of several differences. Interestingly, insertions into ordered slots do not use integer indices. Rather, an identifier to the element after which the insertion should be done is given. This avoids having to modify the indices when merging two modifications to the same ordered slot. They also correctly note that for a bidirectional relation with an unordered slot, the difference calculation of that slot can be derived from the difference calculation of the opposite slot. They also have a difference metamodel, but conflicts do not have one. They do not discuss difference inversion.

An interesting way to display differences is coloring. Here, a special "union" is calculated such that a difference is shown on top of a base version [125, 124]. Usually the common parts are shown in black, whereas addition and removals of elements are displayed in two different colors. In [125], the difference calculation also has a separate operation for a *position shift*, which is emulated by our algorithm with an insertAt/removeAt pair of operations. It is not clear if either choice has any significant impact.

An interesting use case for model differences is given by Lin, Zhang and Gray in [106]. They integrate model difference and difference visualization routines into a model testing framework, where model differences are used to compare test results with the expected output model. Dimitrios Kolovos, Richard Paige and Fiona Polack also use model differences for model compositions and model transformation testing in [94].

An important break-through in practice can be expected from Eclipse, in the form of the new EMF Compare project [27]. It is a work in progress component with the goal to provide model comparison and merge support for EMF. It also has a better separation of concerns in its architecture than the algorithms in this chapter. It especially considers *matching* a separate activity from calculating a difference. It can thus perform matching using different algorithms, which is a clear improvement.

The model repository with version information using a SQL backend described in this chapter is merely a proof of concept tool. As already stated in the previous chapter, the relatively new Teneo library [54] supports creating an object-relational mapping from EMF models to a relational database.

It is noteworthy that most successful textual repositories work on a line-by-line basis. However, Coven [39] emphasizes fine-grained management of textual code. Model repositories provide a similar fine-grained versioning of elements and their interconnections.

7.8 Conclusions

This chapter has presented several metamodel-independent algorithms regarding difference calculation between models. We have described a difference calculation algorithm between two models and a merging algorithm for applying the difference to the original model to produce the target model. Additionally, we have shown how to perform the operation in reverse, producing the original model starting from the target model and the original difference. These algorithms describe a difference as a sequence of operations. The set of operations is complete and each operation has a dual.

Although calculating the inverse of a difference is a rather simple operation and not really discussed by other related work, we consider it to be important nevertheless. At least its textbased equivalent (i.e., reverse patch) is frequently used and useful.

Our work is heavily dependent on persistence of element identifiers, but we do not see that as an issue. For example in the context of a development project within a company it can be assumed everybody uses adequate tools and version control supporting element identifiers.

Furthermore, the difference calculation is used to form a union algorithm, where two separate modifications are made to a base model, and the union algorithm combines both differences into one model by properly interleaving, where possible, the operations in the latter difference with the former difference. At all times, distinguishing unordered slots from ordered ones is important, since conflict resolution is greatly simplified with unordered slots.

There are several cases where merge conflicts are a fact and manual resolution is required, as is also the case with textual programming languages. Modifying the same ordered slot easily creates such situations. For bidirectional relations, both the slot and its opposite slot must be kept in synchronization. The extreme case of deleting an element even though another difference merely modifies it slightly leads to a complicated question; which difference should be prioritized? It remains to be seen whether a union algorithm can work by merging only two parallel differences, or if more differences have to be considered simultaneously.

Additionally, we have shown how to use these algorithms to create a version control system. However, these basic algorithms should be extended to support metamodel-specific resolution mechanisms, and usable manual conflict resolution from the user's point of view is a major hindrance to the adoption of collaborative development of models. Conflicts also need more rigorous handling, perhaps in the form of a separate conflict metamodel. Further work in the general area of difference calculation, merging and version control is required as automatic conflict resolution is important to enable models to become the primary software artifact. The technology transfer into vendor tools is also lacking, as usability is a big problem.

Chapter 8

Diagram Handling

8.1 Introduction

To fully realize the Model Driven Engineering [89] vision we need to define modeling languages and model transformation languages rigorously and we need to provide software development tools supporting them. To ensure interoperability, long term availability and support, these languages and tools should support accepted standards. Software modeling languages are often based on visual notations since this brings important benefits to software development [70]. As a consequence, model transformation languages and model transformation tools need to support visual notations in one way or another.

The Object Management Group (OMG) maintains a series of modeling standards that are used by the industry and studied by the academia. One of the main characteristics of the technical space [19] defined by the OMG modeling standards is that the abstract syntax and concrete syntax of a model are two different artifacts that are defined and maintained independently. The abstract syntax of a language can be defined using the Meta Object Facility (MOF) [139] and the UML 2.0 Infrastructure [142]. To represent the concrete syntax or diagrams of a model, the OMG provides a standard called Diagram Interchange (DI) [138] to interchange two-dimensional diagrams. DI is a language that has been defined following the same metamodeling approach as MOF and the UML. However, DI is not specific to UML. It can be used to represent UML diagrams as well as diagrams for domain-specific modeling languages.

We must emphasize our understanding of the separation into abstract and concrete models. An abstract model contains the modeled data, whereas a concrete model does not contain any information relevant to the modeled data. Thus, a concrete model is only necessary for displaying to a human being. For example, using colors in state machines to denote any information of relevance is not a proper use of diagrams, unless that color information is based directly on the information in the abstract model.

DI is a key standard to allow the interchange of models between tools that need to represent, create or transform diagrams. Examples of these tools range from simple diagram viewers to full-featured interactive model editors or model transformation tools.

Many researchers have studied the definition of new model transformation languages and tools that support in one way or another these OMG standards. The OMG also proposes a standard for a model transformation language called Query-View-Transform (QVT) [133], but there are many other interesting transformation languages and tools, such as [46, 146, 86, 21, 181]. However, most of the existing transformation languages and tools are based on the abstract syntax of a modeling language. Therefore, they do not deal with diagrams as such.

A transformation definition could include rules to update the diagrams associated to a model, since, in fact, diagrams are just organized as another model. We consider that this approach is not satisfactory due to the fact that diagram transformations are rather complex but independent of the semantic transformations and, therefore, they can be reused from one model transformation to another. That is, it is possible to define rules to create and update diagrams that are independent of the actual model transformations.

In this chapter, we describe a way to create and update diagrams after executing a model transformation. We exemplify this idea by creating a domain-specific mapping language called DIML, and show how we have used it to provide a mapping from abstract models to DI diagrams. We assume a tool setting as shown in Figure 8.1: model transformations are executed by a generic transformation component that updates the semantic information in a model based on a transformation definition. Once the transformation is completed, a generic diagram reconciliation component updates the diagrammatic representation of a model based on a diagram definition. A key requirement for these tool components is to follow the existing OMG standards and to be able to interoperate with existing modeling tools.

This chapter is based on Publications II and III. We proceed as follows: first, we briefly discuss how models and diagrams are organized according to the OMG standards in Section 8.2. In Section 8.3, we describe a language to define mappings between models and DI diagrams. In the following two sections, we describe how these mappings can be used to construct a diagram creation and reconciliation component for new and existing diagrams. We discuss our experiences in implementing this approach in a tool in Section 8.6. Finally, we conclude with a discussion on related work in Section 8.7 and final remarks in Section 8.8.

8.2 Models and Diagrams

A complete discussion of modeling languages is out of the scope of this chapter but has been discussed in Chapters 2 and 3, where related work has also been mentioned more extensively. However, from the point of view of this chapter, it

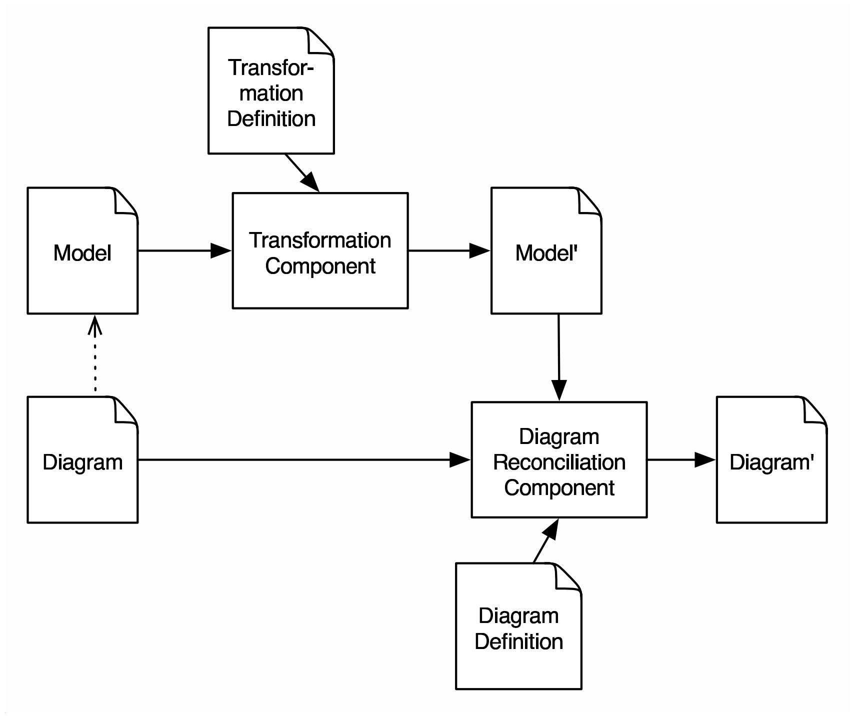


Figure 8.1: Diagram Reconciliation in a Model Transformation Toolchain

is merely necessary to require a modeling language to exhibit these two important features:

- Classes and instances: The type of a model element is defined as a class that can inherit the properties of other classes. Each element has one single type that defines all its possible slots. Each slot always belongs to one element.

- Separation of abstract and concrete syntax: Models as such only contain semantic information but not how to represent it diagrammatically. A different language is used to define the visual appearance of a model. Therefore, a model and its diagram(s) are different artifacts maintained independently.

We consider that the work described in this chapter can be used in any modeling framework that exhibits these two features. In the rest of this section we briefly discuss the language proposed by the OMG to define the concrete syntax of a model.

An example fragment of the UML metamodel describing state machines is shown in Figure 8.2. It must be noted that we have simplified the metamodel for the purposes of this chapter. We will use this metamodel later in this chapter.

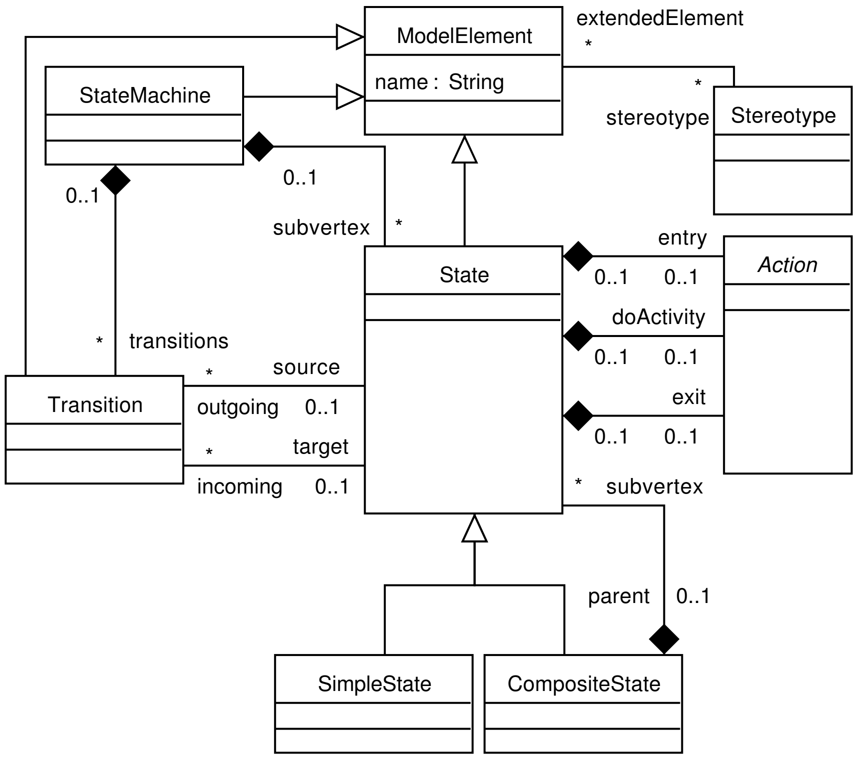


Figure 8.2: A Simplified Fragment of the UML Metamodel for State Machines

8.2.1 The Diagram Interchange Metamodel

The purpose of the OMG Diagram Interchange is to allow the diagrammatic representation of concepts in a model. DI is a rather small language with only 22 classes; a relevant subset of them is shown in Figure 8.3.

There are basically four main concepts in DI: GraphNode, GraphEdge, GraphConnector and SemanticModelBridge. A GraphNode represents a rectangular or elliptical shape in a diagram, such as a UML Class or an Actor, while a GraphEdge represents an edge between two other elements such as two nodes in a UML Association or a node and another edge such as in a UML AssociationClass. A GraphConnector is used as an anchor point for an edge. Nodes, edges and connectors have different properties to define the position and dimensions in a two-dimensional space. It is also possible to define other features such as colors or fonts to be used to render these elements.

Finally, a SemanticModelBridge is used to establish a link between the semantic or abstract model and the diagrammatic model. For example, a GraphNode representing a UML Class is connected to that class using a SemanticModelBridge. There are two types of bridges. A Uml1SemanticModelBridge uses a directed

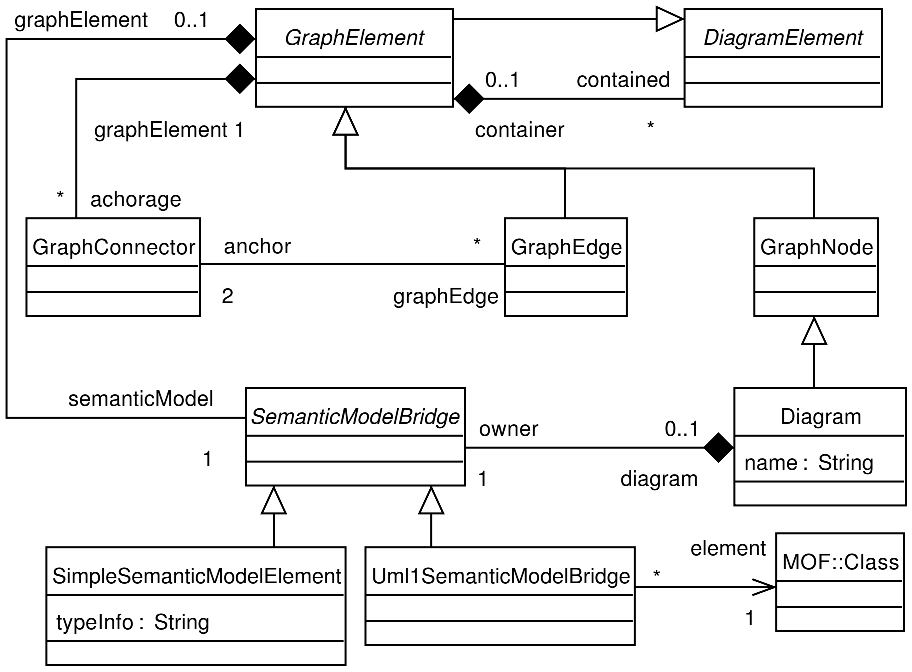


Figure 8.3: A Subset of the Diagram Interchange Metamodel

link to an element, while a SimpleSemanticModelElement contains a string property called typeInfo. These concepts are explained in more detail in the DI standard [138].

Figure 8.4 shows an example of a fragment of a UML model. The left part of the figure is an object diagram showing a simple state and its diagrammatic representation in DI. On the right hand side of the figure the same model fragment is shown, rendered as an image. The dashed arrows in the figure represent how each individual node in the DI model is rendered. This image was created by a tool based on the information contained in the UML model, such as the name of the states, the DI model, such as the layout of the states, and built-in knowledge about the UML notation for state machines, such as the fact that a state is represented as a rectangle with rounded corners. In this case the image was rendered by the tool as Encapsulated Postscript [3]. However, it is also possible to render the image in other graphical formats, such as SVG [190].

From the object diagram we can see that this DI model contains elements necessary for displaying and layouting information retrieved from the UML model. To simplify the figure, we have omitted some UML and DI elements. We especially do not show the Uml1SemanticModelBridge elements but merely a directed link between DI graph elements and the UML elements. We should also note that we show the links that correspond to composition associations using a black diamond.

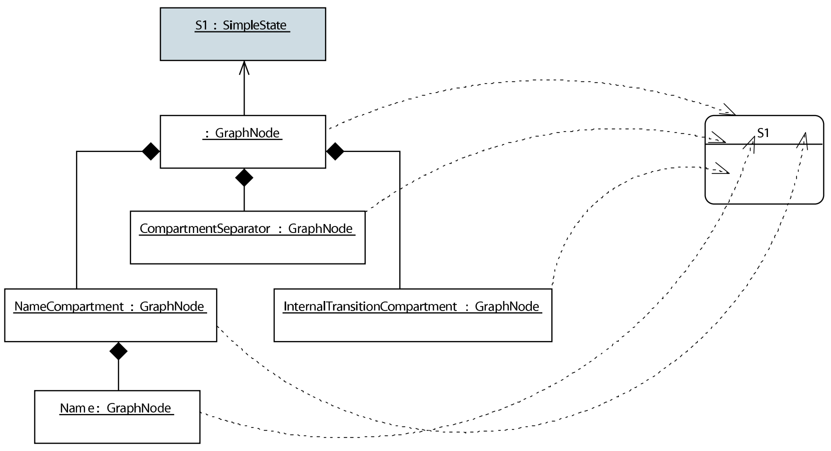

S1 : SimpleState
: GraphNode
CompartmentSeparator : GraphNode
NameCompartment : GraphNode
InternalTransitionCompartment : GraphNode
Name: GraphNode
S1

Figure 8.4: Rendering of a DI Model to an Image

Although the notation of this object diagram is not defined in the UML standard, it is useful for the purposes of this chapter.

In Figure 8.5 a larger example of a fragment of a UML model is shown. The top part of the figure is a simple UML statemachine model with two states connected with one transition, presented as a UML object diagram. The bottom part of the figure shows the same DI model rendered as an image. This example shows the states represented as nodes and the transition represented as an edge containing nodes in the same DI Diagram. From this figure we can also see that DI uses GraphConnectors for connecting the endpoints of an edge to other DI elements. Although the GraphConnectors are not visible in the rendered diagram, they are important for layouting the edges.

8.2.2 The Diagram Reconciliation Component

The diagram reconciliation component can create new diagrams or update existing ones. New diagrams may be created if there were no previous diagrams in the model, for example the transformation component is actually a reverse engineering tool that has created a model from code, or when the transformation maps a model from one language to another. On the other hand, a transformation component may sometimes perform a partial change in a model, where only a subset of the existing elements are updated, added or deleted. In this case, the diagram reconciliation component should try to preserve as much information from existing diagrams such as layout, colors and fonts as possible. That is, the diagram reconciliation component should work incrementally, performing the minimum set of updates necessary to maintain consistency of existing diagrams with the abstract models.

In our approach we consider the transformation definition and the diagram definition to be two different artifacts that can be defined and maintained inde-

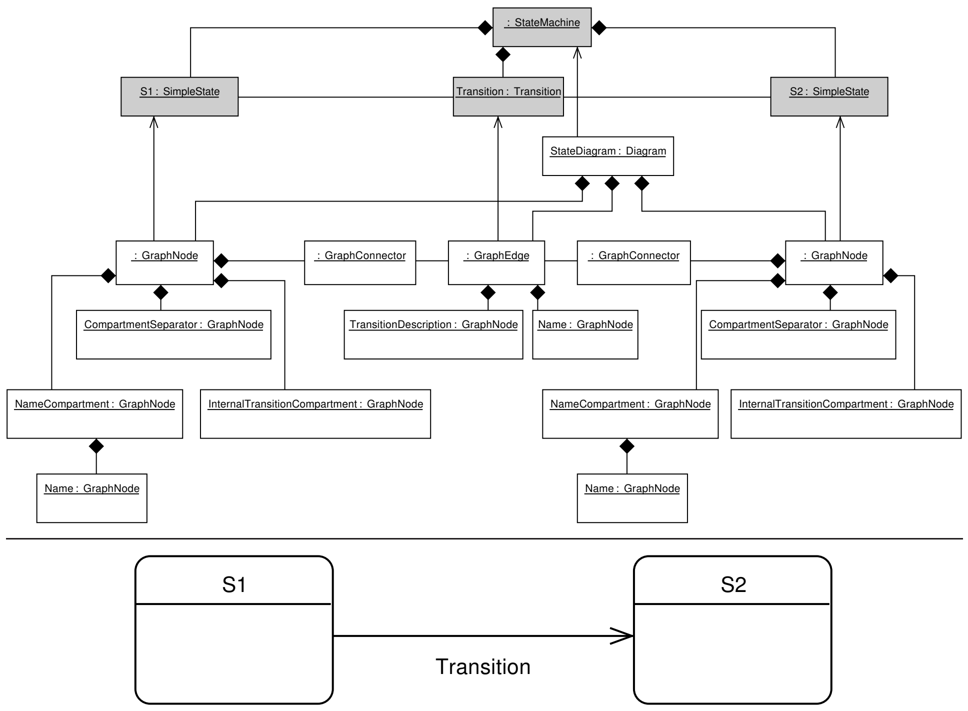

: StateMachine
S1 : SimpleState
Transition : Transition
S2 : SimpleState
StateDiagram : Diagram
: GraphNode
: GraphConnector
: GraphEdge
: GraphConnector
: GraphNode
CompartmentSeparator : GraphNode
TransitionDescription : GraphNode
Name : GraphNode
CompartmentSeparator : GraphNode
NameCompartment : GraphNode
InternalTransitionCompartment : GraphNode
NameCompartment : GraphNode
InternalTransitionCompartment : GraphNode
Name : GraphNode
Name : GraphNode
S1
S2
Transition

Figure 8.5: (Top) UML Model in Gray with Two SimpleStates and a Transition and its Diagram Representation in DI (Bottom) DI Diagram Rendered Using the UML Concrete Syntax

pendently. The same applies to the transformation and the diagram reconciliation components: they can run in the context of an integrated modeling environment or they could be completely different tools.

We consider that there are several important benefits in this approach. First, the construction of new transformation tools and the definition of new transformations is simplified since they do not need to deal with diagrams. Secondly, it is possible to create different diagrammatic representations from the same semantic information. Also, it allows a market of independent tool components to transform and update models and diagrams. Finally, the tool component in charge of diagram handling can be optimized to its specific task and therefore it can be more efficient than a generic transformation engine.

8.3 From Models to Diagrams

We have seen in the previous examples that DI provides us with the basic classes, the instances of which can be combined to create diagrams. However, neither the UML nor the DI standard tell us which classes we should use to create a specific diagram to represent a specific model (Appendix C of [138] does not provide adequate information for either UML 1.4 or UML 2.0). As we have seen in the example, this task is not trivial since each UML model element is represented using many DI elements and the mapping between the model element and its diagram representation cannot immediately be derived. This in turn complicates the interchange of DI diagrams between modeling tools, as diagrams created by one tool may not be compatible with the diagrams the other tool creates. Full compatibility can be ensured only if the tools use the same definitions for creating the diagrams.

To be able to create a truly reusable and independent diagram reconciliation component, this component should support different modeling languages and be based on standards. The first requirement implies that the rules to create or update diagrams for a given modeling language should not be hardcoded into the transformation tool but defined in a tool-independent format that can be loaded by the reconciliation component at runtime. The second requirement implies that the diagrams created or updated should conform to a standard diagram interchange format such as DI. This would allow the user to view and edit transformed models in a DI-compliant model editor. Currently, we are only aware of one commercial tool, Gentleware's Poseidon [61], that supports DI. We consider this tool as a reference implementation of DI.

To address this issue, we have created a language called the Diagram Interchange Mapping Language (DIML). Its purpose is to define mappings between classes in MOF-based modeling languages, such as UML, and corresponding elements in the DI language. An overview of the DIML language with respect to other modeling languages can be seen in Figure 8.6. In this figure, a dashed arrow indicates conformance, and a plain arrow indicates usage. We assume that this mapping language is defined using the MOF standard. The actual mappings are described using a model in this mapping language. Each of these models maps an element in the modeling language to a set of elements in the DI language. This information can then be used by an application of this mapping language that interprets the mappings and applies them to actual models and diagrams. Thus, DIML is a domain-specific weaving metamodel.

We can see example DIML models for UML StateMachines, SimpleStates and Transitions shown in Figures 8.7, 8.8 and 8.9 respectively. These mappings conform to the simplified structure of StateMachines presented in Figure 8.2. In the figures, an abstract element on the left is mapped to a hierarchy of diagram elements as DIML Parts. Each part, shown as rectangles, maps to a GraphNode, GraphEdge or Diagram in DI. The directed arrow corresponds to the mapping concept, whereas the edges with black diamonds correspond to element ownership based on guard and selection statements, the former inside brackets. The hierarchy forms a parameterized skeleton which when transformed into DI elements in a specific context gives us the intended result.

An example of the application of these three mappings was seen in Figure 8.5. The topmost part of the figure (colored gray) shows a StateMachine with two Sim-

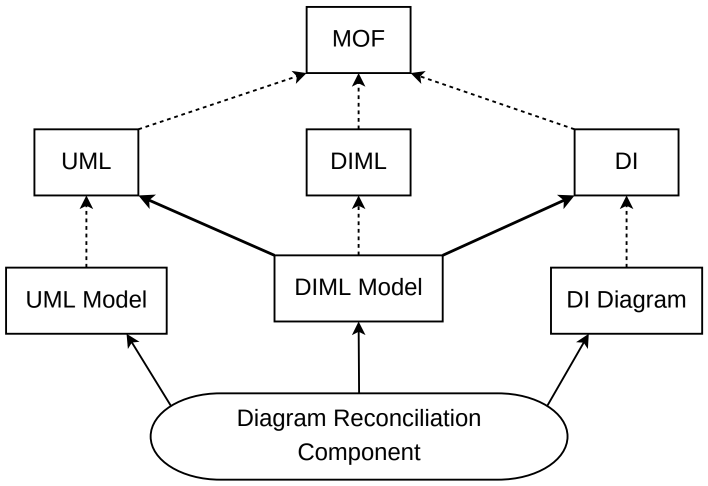


Figure 8.6: Overview of the Mapping Between Models and Diagrams

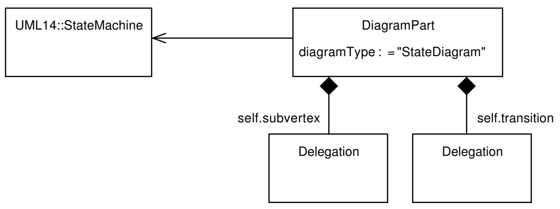


Figure 8.7: The DI Mapping Rule of StateMachines

pleStates and one Transition. When the mapping for UML StateMachines (Figure 8.7) is applied to the StateMachine, a DI Diagram will be created. When the mapping for UML SimpleStates (Figure 8.8) is applied to the SimpleStates and the mapping for UML Transitions (Figure 8.9) is applied to the Transition, DI elements will be created for these UML elements. Finally, these DI elements will be connected to the Diagram. As a result, the DI model shown in the middle of Figure 8.5 is obtained. By comparing the DIML models to the actual diagram, we see that not all DIML Parts are represented in the resulting diagram. For example, there is no StereotypeCompartment for the SimpleStates. This is an example of the parameterization; since the SimpleStates had no abstract Stereotype elements, the guard “self.stereotype–>notEmpty()” in the DIML model returned false and thus no StereotypeCompartment was created.

Thereby the three mappings for StateMachine, SimpleState and Transition from Figures 8.7, 8.8 and 8.9 have been used to create several DI tree fragments as outlined by triangles in Figure 8.10, yielding the final DI Diagram in Figure 8.5. The GraphConnectors are used to connect GraphEdges together with GraphElements.

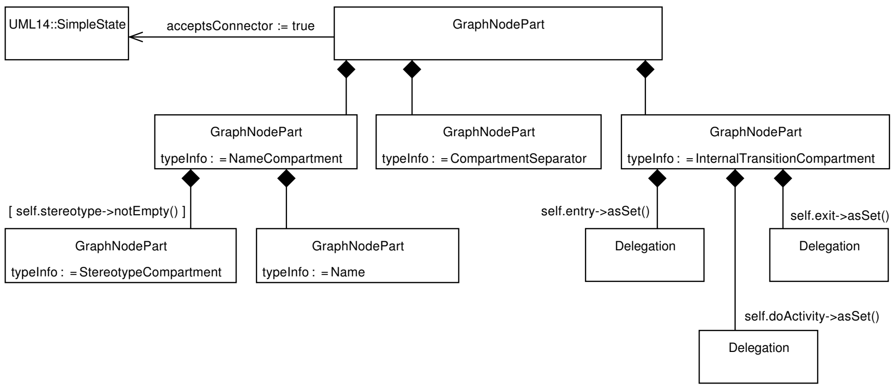


Figure 8.8: The DI Mapping Rule of SimpleStates

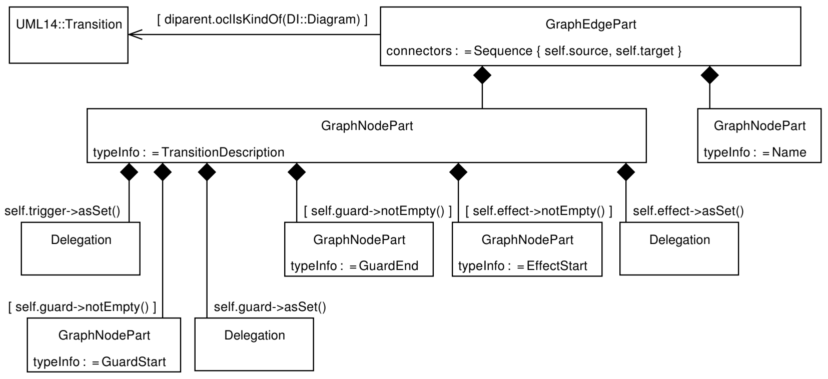


Figure 8.9: The DI Mapping Rule of Transitions

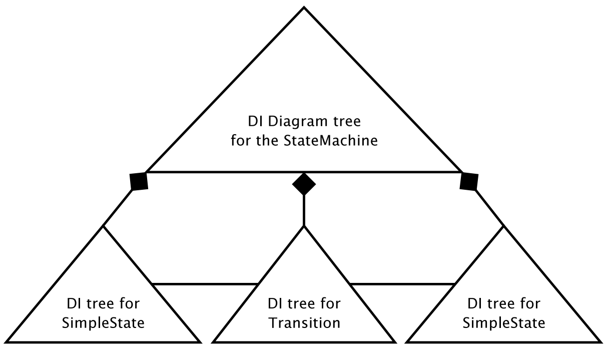


Figure 8.10: DI Fragments Created by the DIML Mappings are Combined into the Final DI Diagram

8.3.1 DIML Metamodel and Semantics

This section discusses the concepts we have used in creating DIML and the semantics of the classes. It is important to notice the separation between the DIML language itself and the various applications of the DIML language. While the main use of DIML is to define diagrams using the OMG standards, DIML does not define or enforce any particular method for applying these mappings on model data. Assuming that a DIML mapping is correct, any tool is still allowed to maintain the abstract model and concrete models in any way it wants as long as the end result is correct, i.e., as if it had used DIML.

This *as if* rule is well-known from for example C compiler technology and gives implementations the greatest leeway while still retaining compatibility between implementations. This separation enables us to concentrate on acquiring a usable mapping language and its semantics, while leaving the actual applications of DIML as a separate concern for modeling tools. In our opinion this separation works favorably for both standardization as well as enabling competing implementations.

The metamodel for DIML is shown in Figure 8.11. In the figure, MOF::Class represents the type of any class. The OCL::OclExpression refers to any OCL expression. OCL [141] is a language for creating arbitrary queries on models. It can be used to return a collection of element references from models or to assert that certain properties hold in a model. The MappingModel is a container to collect the mappings of a given modeling language or profile. Every DIML model must have one MappingModel as its root element. An ElementToDIMapping element *m* is then a mapping of one abstract element of type *m*.element to its diagrammatic representation as DI elements.

In Figures 8.7, 8.8 and 8.9, the ElementToDIMapping elements are denoted by directed arrows and the Contained elements are the composition links. There can be two different text strings next to those links; a text in brackets is a Contained.guard expression, and a text without brackets is a Contained.selection expression.

The slot *m*.root points to a *DIML tree*.

8.3.2 DIML Tree

Each mapping rule is basically a tree of parts. Such a DIML tree consists of an InitialPart as its root, and a hierarchy of Contained and GraphElementPart elements. Leaves in the tree are either of type Delegation or have no children Contained elements. The purpose of a DIML tree is to describe a parameterized skeleton which can be used to compute a resulting DI tree. Parameterization is accomplished by the Contained elements, and means here that the occurrence and recurrence of child GraphElementParts is determined by the slot values in Contained.guard and Contained.selection. These expressions allow us to create a mapping to DI context-dependent on the abstract model element and all the other abstract model elements

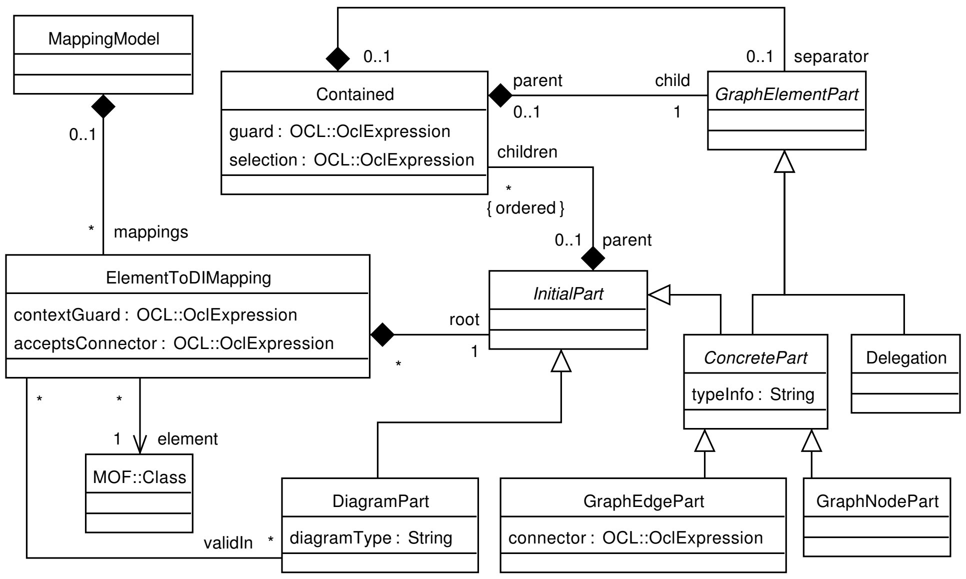


Figure 8.11: The DIML Metamodel

as well as the chain of parents in the DI model. These expressions, together with instances of ConcretePart and Delegation are the primary means to represent a collection of similar DI fragments (modulo the parameterization) as one DIML tree.

In a DIML tree, an instance *p* of GraphEdgePart or GraphNodePart corresponds to an arbitrary number of instances of the DI elements GraphEdge or GraphNode *d*, respectively. A DiagramPart always corresponds to exactly one DI Diagram. There is one DI element for each element found when executing the Contained.selection expression. A Delegate corresponds to a change of DIML mapping rule. It does not correspond to any DI element. It can be seen as a placeholder for the next DIML tree.

According to the DI standard, a Diagram has a SimpleSemanticModelElement *s* in its semanticModel slot such that *p*.diagramType = *s*.typeInfo, and a Uml1SemanticModelBridge in its Diagram.owner slot which points to the abstract element for which the diagram was created for. A GraphEdge or GraphNode has either a Uml1SemanticModelBridge or a SimpleSemanticModelElement. If *p*.typeInfo is empty, *d* must have a Uml1SemanticModelBridge which points to the abstract element. Otherwise, *d* must have a child element *s* of type SimpleSemanticModelElement such that *p*.typeInfo = *s*.typeInfo.

8.3.3 Support for DI Elements

Figure 8.7 showed an example mapping for a diagram. Such a mapping *m* has a DiagramPart element *r* in its *m*.root slot, with *r*.diagramType denoting what diagram type is being considered (e.g., “ClassDiagram” or “StateDiagram”). The *m*.contextGuard is evaluated and must return true. It is an OCL expression which receives the abstract element and *diparent* as its parameters. *Diparent* is the parent element in the DI model. It is guaranteed to exist for any GraphNode or GraphEdge except for Diagram, which has no DI parent. Thus, for diagrams, *diparent* is always a null pointer/reference. Using *diparent*, we can query the chain of parents in the DI model. The contextGuard can be used to limit whether or not it is allowed to create a diagram for the given abstract element.

The slots *m*.validIn and *m*.acceptsConnector are unused and must be empty. The element *m*.root is the root element of a DIML tree.

Figures 8.8 and 8.9 show example mappings for states and transitions. Such a mapping *m* is otherwise similar to a mapping for a Diagram, but with some small differences. The element *m*.root must either be a GraphEdgePart or a GraphNodePart, with *m*.root.typeInfo being the empty string.

The *m*.contextGuard must still hold, but the *diparent* will now be a valid DI element in the diagram. An example of this can be seen in Figure 8.9, where the mapping can only be used if the expression *diparent*.oclIsKindOf(DI::Diagram) holds, i.e., when the parent DI element is the root Diagram element.

The set *m*.validIn.diagramType denotes the set of valid diagram types, for example { “ClassDiagram”, “SequenceDiagram” }. This is the set of types of diagrams in which the mapping can be applied. Although technically the validIn information could be embedded in the contextGuard, it is more convenient to have a set of diagrams where a mapping can be applied because it avoids unnecessarily long OCL expressions in the contextGuard, and the information about suitable diagrams is easier to extract from a slot made for that purpose rather than extract it by parsing an OCL expression. Again, starting at *m*.root, the DIML tree can be described.

8.3.4 Connecting Edges to GraphConnectors

The interpretation of a DIML mapping so far enables us to describe a tree of DI elements. However, a diagram in DI is not merely a tree, but a graph where GraphElements are connected together via GraphEdges. The problem is how to describe which connections are allowed, and which are not, specially when we consider that the same abstract element can appear several times in a diagram. For example, a UML Class can be shown both as a rectangle but also as the type of an attribute or a parameter of an operation. However, only when a class is represented as an independent rectangle can it be used to connect Generalization or Association edges. Connecting an Association to the type of an Attribute can be considered valid from a semantic point of view, but it is against the presentation rules of UML class diagrams.

Our solution to this problem requires two properties. A GraphEdgePart *p* has a *p*.connector expression. It is evaluated in the context of the corresponding abstract element and receives the GraphEdge *g* as an additional parameter. The evaluation results in a sequence of abstract elements. For each element *e* in the sequence, a GraphConnector is created (or must already exist) and anchored to *g*. The owner of the GraphConnector must then be found in the set of all GraphElements in the same diagram whose corresponding abstract element is *e*. This GraphElement must correspond to a root ConcretePart in an ElementToDIMapping *m* mapping such that *m*.acceptsConnector is satisfied. The acceptsConnector expression does not receive any parameters, and is thus usually the expression *true* or *false*.

This scheme is required since not all GraphElements may be connected to and the only distinguishing information is the context. In our work, this context can be exploited by having several ElementToDIMappings for the same abstract class. One of the mappings is chosen based on the contextGuard slot value.

8.3.5 Known Limitations

We have used DIML quite extensively in managing the diagrams in our modeling framework. From this experience we have found that the metamodel and semantics of DIML imply some limitations in what kinds of Diagrams DIML can describe.

The first limitation is that the source of a mapping can only be a class, not a property or a relation between classes. Even though properties can appear in the mapping rules, specially in the Contained.selection expressions, they cannot map to the root InitialPart of a DIML tree. This limitation appears quite often in OMG standards. For example, the model interchange format XMI [129] can serialize an element and its slots, but it cannot represent individual slots. The consequence is that a relationship between elements should be represented as a class. For example, in UML, the Generalization relationship is a class, instead of two combined properties (such as superclass and subclass).

The second limitation is that the target of a mapping is a single DIML tree instead of multiple DIML trees. Although this multiplicity would be easy to fix in the metamodel, the elements in different trees must also be able to reference each other (as per one of our use cases related to UML AssociationClasses). This would require more thorough changes.

Finally, that our diagrams can be built top-down, i.e., starting from the DI Diagram element, child elements can be transitively connected to form a complete diagram without any changes required in their parents during diagram construction. This means that a parent DI element does not depend on what child DI elements exist underneath it. This is emphasized by the decoupling provided by the Delegation elements in the DIML models.

At the moment, we believe the first limitation to be more important than the second or third one for practical modeling. Improvements to DIML or similar mapping languages should address at least this first limitation, but preferably also the second. However, we would still like to keep DIML a domain-specific transformation language instead of creating or using a general-purpose one.

8.4 Generation of New Diagrams

As mentioned earlier, a use of DIML is the automatic creation of a specific diagram from the abstract model. In this section we assume that a new model that does not contain any diagram has been created and we wish to create a specific diagram to represent the model graphically.

This task can be described by a depth-first algorithm, of which an outline is seen in pseudocode in Figure 8.12. The starting point is the function *create_diagram*, which takes the abstract model element *e* and a diagram type string *diagramType* as its formal parameters. Since a diagram is a tree of DI elements with respect to element ownership and has a DI Diagram element as the root, we first need to find a valid ElementToDIMapping element *e* where *e*.element points to the corresponding abstract class (MOF::Class), *e*.root points to a DiagramPart where *e*.root.diagramType tells what kind of a diagram the mapping describes. After that, the hierarchy of DI elements is created by recursing in the *create_di* function.

To summarize the algorithm, we can consider that a diagram is created as follows. In *create_di*, the generator follows the mappings given in the DIML model. Here, *e* is an abstract element, *diparent* is either the immediate parent DI GraphElement or the null pointer, and *part* is a DiagramPart, GraphNodePart, GraphEdgeParts or Delegation.

If *part* is a Delegation, we need to find a new mapping for the abstract element. The function *in_suitable_diagram* returns true when the mapping is valid in a specific diagram: for example, a mapping for a UML Class is valid in Class diagrams, not in State diagrams. The actual definition of the function is simple and not described further.

Otherwise, we create a corresponding DI element on lines 24–36 and set the SemanticModelBridge: either a Uml1SemanticModelBridge or a SimpleSemanticModelElement. After that, the loop on line 37 is responsible for creating DI elements on one level of the hierarchy, with the recursion occurring on lines 41 and 45. The guard evaluation on line 38 and the selection evaluation on line 44 give the developer of DIML models the flexibility to create a parameterized DI Diagram from abstract model data.

On lines 38–49 we do the following for every Contained element *c* in the children slot of the *part*:

- Evaluate *c*.guard in the context of *e* and with *diparent* as its parameter. If it does not hold, we must proceed to the next Contained element.
- Evaluate *c*.selection in the context of *e* and with *diparent* as its parameter. The expression must return an OCL collection of abstract elements. If the expression string is empty, it defaults to returning a set consisting of the current element *e*; this is mainly used for children GraphElementParts with a typeInfo string. For each element *s* in the collection, the *c*.child GraphElementPart is accepted in the context of *s* as the abstract element, and *g* as

```
function create_diagram(e, diagramType):
    find an ElementToDIMapping m where e.oclIsKindOf(m.element)
        and m.root.oclIsKindOf(DiagramPart)
        and m.root.diagramType = diagramType
    return create_di(e, null, m.root)
```

```
function attach_connectors(diagram):
    for each GraphEdge g transitively contained in diagram:
        find a GraphEdgePart part that corresponds to g
        e := g.semanticModel.element
        for c in part.connector(e):
            find an ElementToDIMapping n where n is a valid mapping of c
                and n.acceptsConnectors() = true
                find a GraphElement r where in_same_diagram(g, r)
                    and n.root corresponds to r
                create GraphConnector gc
                gc.graphElement := r
                gc.graphEdge.add(g)
```

```
function create_di(e, diparent, part):
    if part is a Delegation:
        find an ElementToDIMapping m where e.oclIsKindOf(m.element)
            and m.contextGuard(e, diparent) = true
            and in_suitable_diagram(m.validIn, diparent)
        return create_di(e, diparent, m.root)
    create g, corresponding to part:
        A GraphNode for a GraphNodePart
        A GraphEdge for a GraphEdgePart
        A Diagram for a DiagramPart.
    g.container := diparent
    if part.typeInfo is nonempty:
        create a SimpleSemanticModelElement s
        s.typeInfo := part.typeInfo
        g.semanticModel := s
    else:
        create a Uml1SemanticModelBridge s
        s.element := e
        g.semanticModel := s
    for c in part.children:
        if c.guard(e, diparent) = false:
            continue
        if c.selection is empty:
            x := create_di(e, g, c.child)
            g.contained.append(x)
        else:
            for s in c.selection(e, diparent):
                x := create_di(s, g, c.child)
                g.contained.append(x)
                if this is not the last s and c.separator exists:
                    y := create_di(s, g, c.separator)
                    g.contained.append(y)
    return g
```

Figure 8.12: Outline of an algorithm for Generating a DI Diagram from an Abstract Model

its *diparent*. This means that same computation must be performed on the new child DIML element by recursing into *create_di*.

- If *c*.separator is nonempty, it denotes a DIML subtree with corresponding DI elements that must be placed between each accepted element. This enables us to easily model the very common occurrence of having a simple separator between values, such as a comma sign between the parameters in an operation in a UML class diagram.

Delegation elements are used to decouple the representation and computation of individual DIML trees. When searching for a new mapping, only one mapping is allowed to be valid. No nondeterminism is allowed. Once a valid mapping is found, DI tree creation can begin again in the context of a new current abstract element and *diparent*.

Furthermore, the GraphEdge elements should be connected to other GraphNode or GraphEdge elements using GraphConnectors. Since the GraphConnectors are owned by the GraphNode or GraphEdge the new GraphEdge connects to, the creation of new GraphConnectors must occur after all other elements in the diagram have been created. This occurs in the operation *attach_connectors*, which takes a *diagram* as its only parameter. In *attach_connectors*, for each GraphEdge transitively contained in the diagram, the corresponding GraphEdgePart *part* from DIML is acquired (preferably retained from create_di). Then, *part*.connector in the context of the abstract element *e* mapped to the edge, returns a sequence of abstract elements. For each of these elements, a corresponding ElementToDIMapping *n* is located and *n*.acceptsConnectors is evaluated. After *n* is found, the GraphElement where the GraphEdge should connect is located. The function *in_same_diagram* tests that both elements are in the same diagram. Finally, a GraphConnector is created to link these DI elements together.

Once all the DI GraphNodes, GraphEdges and GraphConnectors are created they should be arranged using a layout algorithm that is appropriate for the particular type of diagram. Examples of a layout algorithm for class diagrams can be found in [53] and algorithms for statechart diagrams can be found in [33].

8.5 Reconciliation of an Existing Diagram

This section briefly discusses the principal idea of how diagram reconciliation works and why it is useful. We also give a short example on modifying the abstract elements in a statemachine and see how diagram reconciliation can update the diagram. Again, we emphasize that this outline of how diagram reconciliation could work is a technical detail, in that any solution that maintains a correct DI structure must be considered valid.

8.5.1 Principle Behind Diagram Reconciliation

As we have discussed in the introduction, there are situations where we want to preserve as much information from an existing diagram as possible after executing a model transformation. That is, the presence of elements in a diagram, their layout, text fonts and color should not change except when this is motivated by the execution of a transformation. In this case, it is possible to define a diagram reconciliation mechanism that can update existing diagrams while using the same DIML mapping language as before.

In principle, diagram reconciliation is an optimization of generating a new diagram. Technically, we could create a new diagram as described in Section 8.4 to replace the old diagram, but that would be too slow and some visual details would be lost. Diagram reconciliation should work at an acceptable speed and as if a new diagram had been created with the visual details intact. Since DIML mappings are declarative constructs instead of programs, they do not demand a certain algorithm for performing diagram generation or reconciliation. Instead implementations are free to use any algorithm that can provide a fast and correct solution.

Diagram reconciliation assumes that it is possible to discover what has changed in a model during a transformation. We assumed in the introduction that the reconciliation component had access to the current abstract model and the obsolete diagrams. We now require that it also has access to either the old abstract model or to a change description. This change description can be a list of atomic changes done to the slots of the elements, or a special model reflecting the difference between the old and the new abstract model. We have discussed a model difference algorithm in Chapter 7.

Based on this information, the reconciliation component can inspect which abstract elements have changed. Using the Contained.guard and Contained.selection expressions in the DIML mappings, it can then calculate which changes have invalidated which DI elements [30] and then apply the mappings again.

Where order of elements is not important, i.e., where the selection expressions are unordered OCL collections, set operations can be used to calculate which elements should be removed, and which should be added. For ordered collections, there are several algorithms (for example [120]) for calculating the *edit distance* for transforming the old contents into the new contents.

The optimal reconciliation is one which reuses as much as possible from the obsolete diagram to bring it up to date. The level of reuse may depend on how sophisticated the reconciliation system is. For example, consider a UML Generalization connecting two classes, this being represented by a specially colored GraphEdge with a triangular endpoint. If a transformation changes which superclass the generalization points to, it can be argued that a valid reconciliation is to delete the edge and create a new edge in its place connecting to the new superclass. However, information such as color and the exact waypoints are lost. A better reconciliation approach is to reuse the existing GraphEdge.

8.5.2 Reconciliation Example

As an example, let us assume we introduce a new state into the model in Figure 8.5, by inserting a new SimpleState (S3) into the StateMachine.subvertex slot. This invalidates the previously valid diagram, since the new state is missing from it.

Using a change description or by calculating a model difference, the reconciliation component determines that there has been a change in the *subvertex* slot of the StateMachine, and finds the corresponding ElementToDIMapping for it, that is shown in Figure 8.7. Then it calculates the difference of the old *self*.subvertex and the new *self*.subvertex. The ElementToDIMapping for StateMachines states that new SimpleStates are inserted in the diagram owned by the StateMachine. The reconciliation component determines that the ElementToDIMapping shown in Figure 8.8 should be used to create a new DI representation of the SimpleState. A new representation of the SimpleState is created and inserted accordingly in the diagram. The resulting UML and DI model along with its visual representation is shown in Figure 8.13.

Next, assume the StateMachine in Figure 8.13 is modified by changing the target of the Transition from state S2 to S3. This is achieved by assigning *self.target := S3* for the Transition. Again, the reconciliation component determines that there has been a change in the *target* slot of the Transition, and finds the corresponding ElementToDIMapping for the Transition, shown in Figure 8.9. The ElementToDIMapping states that Transitions are represented as GraphEdges in diagrams and connect to States such that the first endpoint is represented by *self.source* and the second endpoint by *self.target*.

As a result, the reconciliation component connects the Transition in the diagram to the GraphNode representing the SimpleState S3. The corresponding UML and DI model along with its visual representation is shown in Figure 8.14.

8.6 Validation of Research

In this section we discuss some aspects related to the practical implementation and validation of a diagram reconciliation component. It should be noted that the actual implementation of DIML has not been done by the author of this thesis.

8.6.1 Optimization of Query Expressions

A great deal of the flexibility of DIML comes from the use of OCL expressions. Using these expressions, DIML models can navigate through the abstract model and select the relevant subset of model elements that will be presented in a diagram. However, this flexibility also leads to complex expressions, which further leads to a cost with respect to reconciliation: it is not apparent which changes in the abstract model will trigger changes in the diagram.

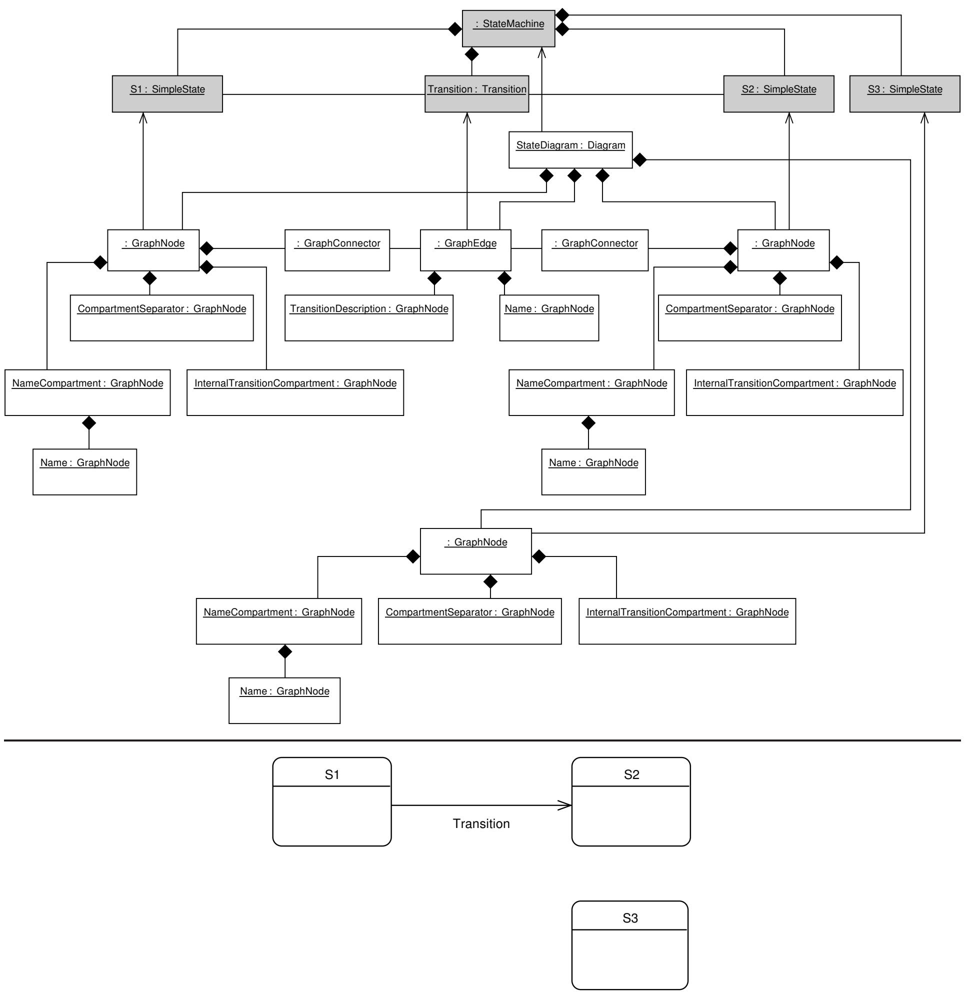

: StateMachine
S1 : SimpleState
Transition : Transition
S2 : SimpleState
S3 : SimpleState
StateDiagram : Diagram
: GraphNode
: GraphConnector
: GraphEdge
: GraphConnector
: GraphNode
CompartmentSeparator : GraphNode
TransitionDescription : GraphNode
Name : GraphNode
CompartmentSeparator : GraphNode
NameCompartment : GraphNode
InternalTransitionCompartment : GraphNode
NameCompartment : GraphNode
InternalTransitionCompartment : GraphNode
Name : GraphNode
Name : GraphNode
: GraphNode
NameCompartment : GraphNode
CompartmentSeparator : GraphNode
InternalTransitionCompartment : GraphNode
Name : GraphNode
S1
S2
Transition
S3

Figure 8.13: (Top) UML Model in Gray After Adding a Third SimpleState and its Diagram Representation in DI (Bottom) DI Diagram Rendered Using the UML Concrete Syntax

Thus, if we were to use the OCL expressions of DIML, we need to be able to quickly tell which values of OCL expressions have changed when the abstract model is modified; this is not easy. Furthermore, OCL allows expressions of computationally arbitrary complexity.

The DIML metamodel uses OCL in five different properties. Since we wish to create an efficient reconciliation system, expressions that can take an unknown amount of computational time are not desired. Our solution has been to use a restricted subset of OCL in some of the DIML properties. This OCL subset should be expressive enough to define the DIML mappings for modeling languages as

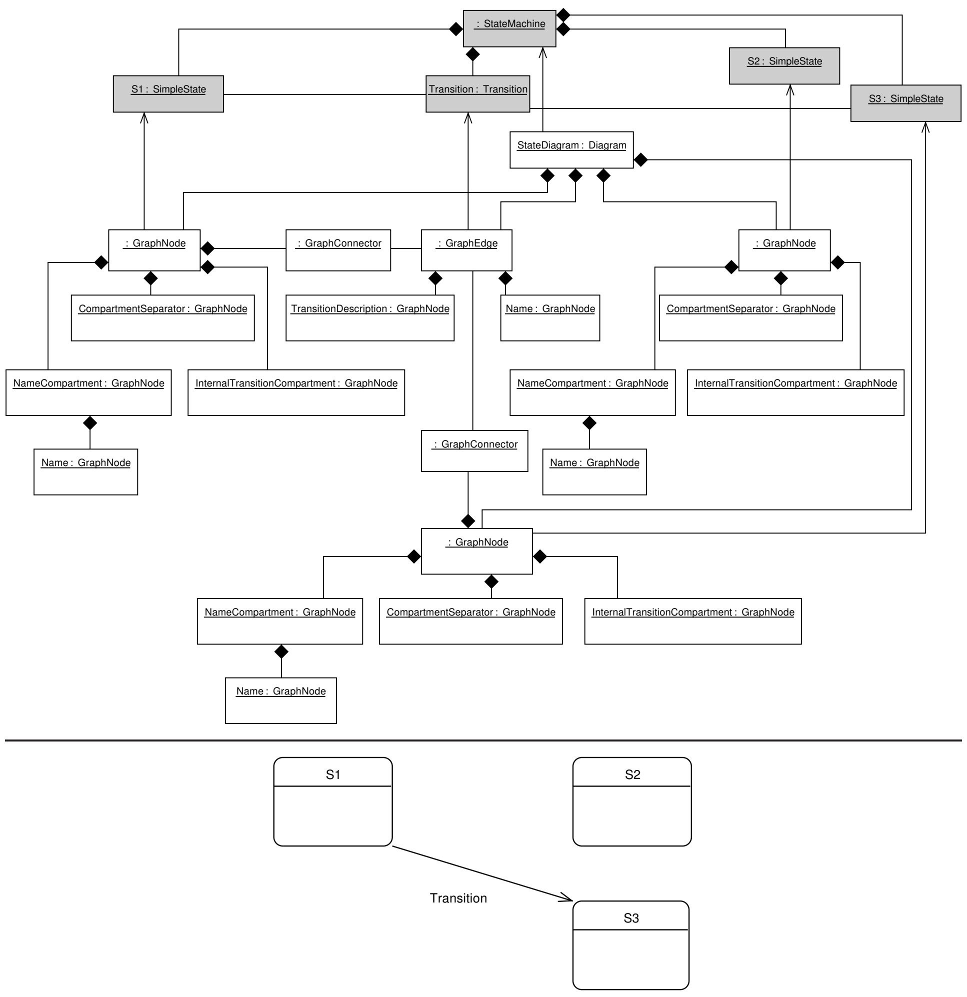

: StateMachine
S1 : SimpleState
Transition : Transition
S2 : SimpleState
S3 : SimpleState
StateDiagram : Diagram
: GraphNode
: GraphConnector
: GraphEdge
: GraphNode
CompartmentSeparator : GraphNode
TransitionDescription : GraphNode
Name : GraphNode
CompartmentSeparator : GraphNode
NameCompartment : GraphNode
InternalTransitionCompartment : GraphNode
NameCompartment : GraphNode
InternalTransitionCompartment : GraphNode
Name : GraphNode
: GraphConnector
Name : GraphNode
: GraphNode
NameCompartment : GraphNode
CompartmentSeparator : GraphNode
InternalTransitionCompartment : GraphNode
Name : GraphNode
S1
S2
Transition
S3

Figure 8.14: (Top) UML Model in Gray After Relinking the Transition and its Diagram Representation in DI (Bottom) DI Diagram Rendered Using the UML Concrete Syntax

complex as UML but it should also be possible to evaluate quickly. The restricted language in the Contained.selection property only accepts expressions based on these patterns:

- *self.x*
- *self.x*.select*(e* | *e*.oclIsKindOf*(z))*
- *self.x*.select*(e* | *e.v* = *w)*
- *self.x*.select*(e* | *e.v* $\neq$ *w)*

- *self*.*x*.select*(e* | *e*.oclIsKindOf*(z)* $\wedge$ *e*.*v* = *w)*
- *self*.*x*.select*(e* | *e*.oclIsKindOf*(z)* $\wedge$ *e*.*v* $\neq$ *w)*

Basically, this restricted language can navigate one slot away from the current element, and optionally filter out elements which are not of the correct type (or a subclass of that type). It can also be used to check primitive values for equality or inequality. This enables us to use the following expressions, for arbitrary property names *x*, *y* and *v*, class *z*, and value *w*.

The GraphEdgePart.connector property can handle expressions either one or two slots away. That is, expressions such as *self*.*x* and *self*.*x*.*y* can be described. For example, the DIML mapping for Transitions uses the expressions *[self*.source, *self*.target *]* whereas the DIML mapping for Associations uses:

$$[\,\mathit{self}.\text{connection}[0].\text{participant}, \mathit{self}.\text{connection}[1].\text{participant}\,]$$

At the moment, we do not restrict the Contained.guard expression language. In our implementation, these expressions are arbitrary programs returning *true* or *false*. This is a drawback in the implementation and makes it impossible to determine when a change in the abstract model triggers a change in the result of a guard evaluation. A more sophisticated implementation using traceability technology would do similar optimizations as we have done, but behind the scenes, and also support more complex OCL expressions. However, implementing an efficient OCL interpreter can be a daunting task since determining quickly which OCL expressions need to be revalidated is not easy. There have been advancements in this area by Jordi Cabot and Ernest Teniente [29, 30].

We consider that the simplified language serves the purposes we have outlined in Sections 8.1 and 8.2, but we acknowledge that the language proposed in this chapter is more general. Even with these restrictions, the optimized query language has shown to be adequate in expressions for UML diagrams. It makes it easier to determine which parts of the diagram need to be updated and therefore it has enabled us to perform reconciliation of models and diagrams using algorithms of low computational complexity, while still being able to support large and complex languages such as UML. This ensures that diagram reconciliation is not an expensive operation, and hence it is fast enough to be integrated with an interactive model editor.

We are still in the process of extracting a clean, easy-to-understand algorithm for reconciling diagrams. A very interesting idea for future work would be to try to accomplish the same as DIML currently does with more common transformation technology, such as ATL [21] or QVT [133].

8.6.2 Implementation in the Coral Tool

We have implemented DIML in Coral, which has a GUI for metamodel-independent model management and different graph transformation mechanisms.

We have implemented a component for Coral that reconciles models and diagrams after executing model transformations or performing editing operations. Our implementation uses DIML mappings with the restricted expression language described in Section 8.6.1. Also, our implementation uses the Python language to define restricted OCL expressions such as Contained.guard or ElementToDIMapping.contextGuard. This is due to the fact that Coral lacks a complete OCL interpreter.

The reconciler was in fact implemented twice, once so that it analyzed the DIML and generated code, and once as an interpreter of DIML models. The first implementation is presented in Torbjörn Lundkvist's Master's Thesis [110], whereas the second one was programmed by Andreas Söderlund and Matias Nyman at Åbo Akademi University.

Our implementation is based on a model manager that automatically collects model changes into a transaction as described in [152]. A transaction is then a list of individual atomic commands, such as "insert element *e* into slot *s* at position *i*" or "create new element of type *t*". The diagram reconciliation component is invoked by the model manager at the end of a transaction so it can update all the necessary diagrams.

Coral can edit and transform models using an interactive graphical editor but also using different model transformation engines. Actually, the diagram reconciliation component does not distinguish between these two cases and it performs the same task independently if a model has been edited interactively by the user or transformed by a program.

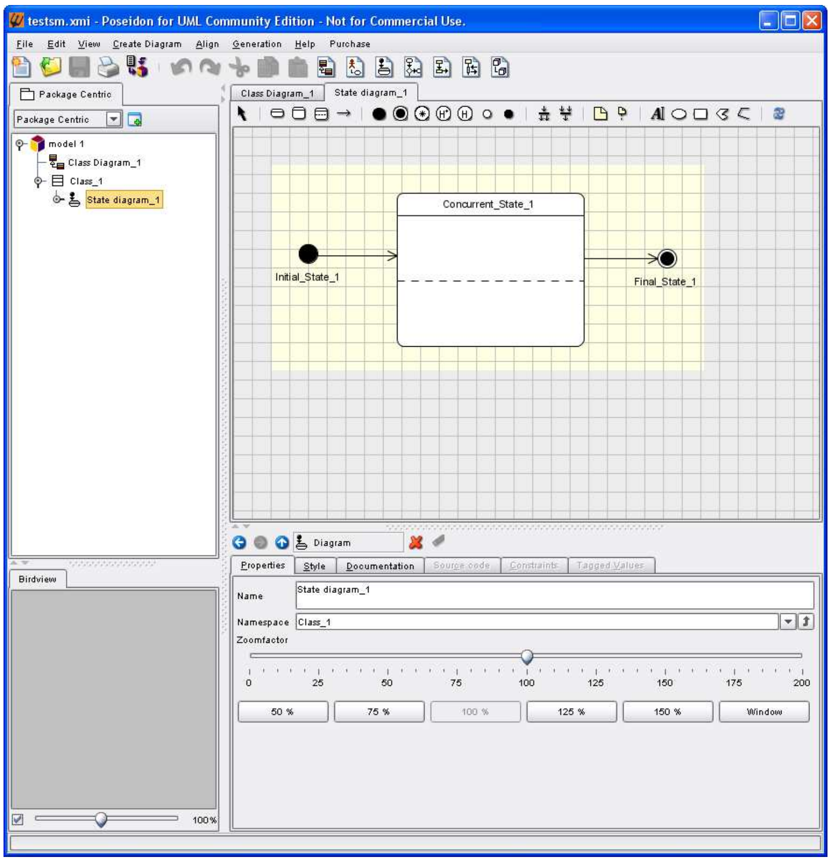
Although most of Coral is written in an interpreted language which does not include a just-in-time compiler or similar feature, the model editor is nevertheless fast enough for interactive editing of models using a low-end desktop computer. Most of the time is in fact spent by the various change propagation components, and the graphical updates; the diagram reconciliation is a very small part of the computational time required.

8.6.3 Validation

We have implemented mappings for the UML 1.4 class, statechart, object, use case and deployment diagrams and we are confident that the DIML language can be used to define mappings for other UML diagrams. The mappings we have used for UML 1.4 in the Coral tool are available in [110]. From these mappings we can see that by using Delegation elements and DIML tree parameterization extensively, we have been able to support all the UML diagrams mentioned above. However, some constructs have not been possible to describe due to the known limitations from Section 8.3.5: the links between Comments and other elements, and AssociationClasses in class diagrams. The former maps a relation from the abstract model to the diagram, and the latter requires a complicated multirooted tree of DI elements. We plan to improve on DIML to overcome these limitations.

The Coral tool supports other user-defined modeling languages and profiles besides UML. We have used DIML to define the concrete syntax of MICAS [102], a domain-specific modeling language to define peripherals for mobile phones, and the concrete syntax of SOCOS [12], a domain-specific modeling language to define refinement diagrams. These examples show that DIML is viable to define the concrete syntax of DSM languages that are different from UML, and that DIML does not require UML.

We have also assessed compatibility with Gentleware Poseidon version 3.0 from which the DIML mappings presented in this chapter are based. In Figure 8.15 we see a model created in Poseidon.



Name: State diagram_1

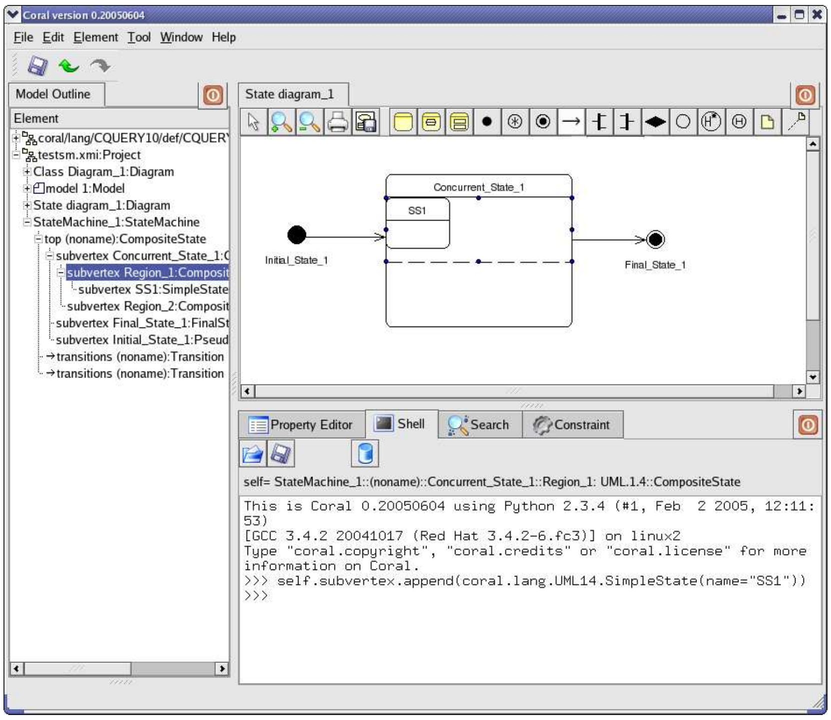
Namespace: Class_1

Figure 8.15: Screenshot of a Statemachine in Poseidon

In Figure 8.16, the exact same file has been loaded into Coral. The composite state is selected and the following command is executed in the Coral shell:

$$\mathit{self}.\mathrm{subvertex.append}(\mathrm{coral.lang.UML14.SimpleState}(\mathrm{name} = \text{“SS1''}))$$

Since the variable *self* refers to the current selection, this command creates a new state in the composite state. Then, the diagram reconciliation component notices the new state in the abstract model, and modifies the diagram accordingly.



```
coral/lang/CQUERY10/def/CQUERY
testsm.xmi:Project
  Class Diagram_1:Diagram
  model 1:Model
  State diagram_1:Diagram
  StateMachine_1:StateMachine
    top (noname):CompositeState
      subvertex Concurrent_State_1:C
        subvertex Region_1:Composit
          subvertex SS1:SimpleState
        subvertex Region_2:Composit
      subvertex Final_State_1:FinalSt
      subvertex Initial_State_1:Pseud
    transitions (noname):Transition
    transitions (noname):Transition
```

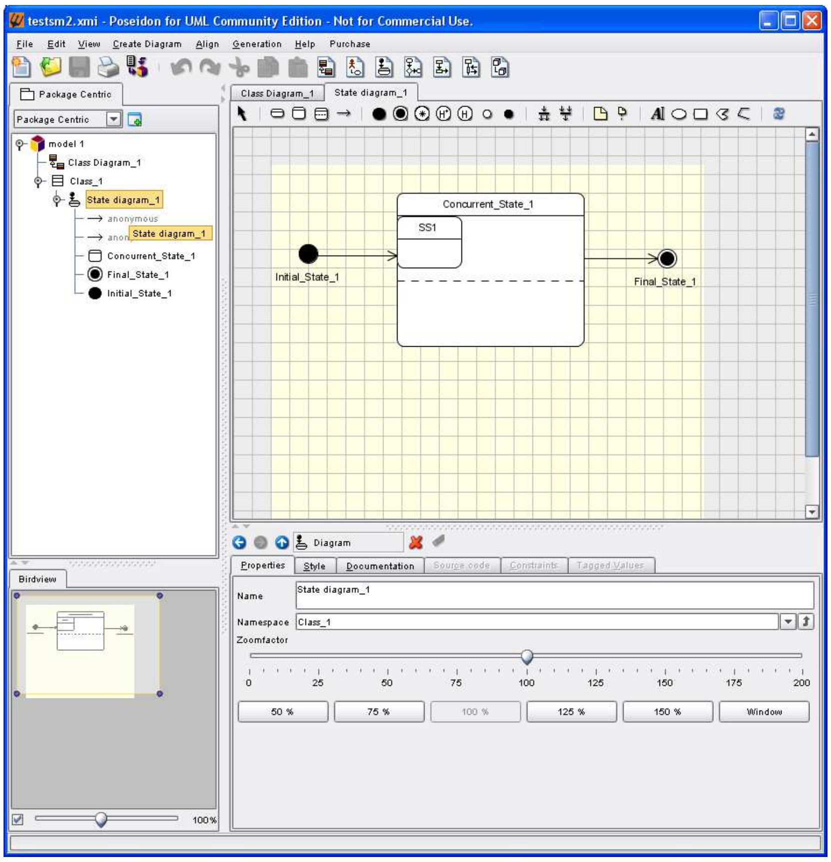
self= StateMachine_1::(noname)::Concurrent_State_1::Region_1: UML.1.4::CompositeState

```
This is Coral 0.20050604 using Python 2.3.4 (#1, Feb  2 2005, 12:11:
53)
[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)] on linux2
Type "coral.copyright", "coral.credits" or "coral.license" for more
information on Coral.
>>> self.subvertex.append(coral.lang.UML14.SimpleState(name="SS1"))
>>>
```

Figure 8.16: Screenshot of the Model from Figure 8.15 Loaded into Coral, with a New State Added

Finally, the model is saved again into XMI and loaded back into Poseidon. As can be seen from the result in Figure 8.17, it is possible to load the file from Coral into Poseidon, and the diagram information is compatible with the diagram information that Poseidon expects. The conclusion is that DIML mappings (for the UML 1.4 statecharts) as used by Coral work according to our DI reference implementation in Poseidon.

We should note that newer versions of Poseidon have changed their internal DI mappings. Therefore, the current version Coral is not compatible with diagrams generated by newer Poseidon versions. However, this does not affect the actual diagram reconciliation component. In order to support the new Poseidon diagrams we should only update the DIML mappings. Naturally, this is an unfortunate situation in itself. The best approach would be for OMG to standardize on some specific mapping rules between the abstract syntax of UML and the concrete syntax of DI.



Name: State diagram_1

Namespace: Class_1

Figure 8.17: Screenshot of the Model from Figure 8.16 Loaded into Poseidon

8.7 Related Work

The work presented in this chapter is related to general purpose model transformation languages, languages specific to diagram definition and diagram editors and development environments that support user-defined visual languages.

There exists full-featured metamodel-based editors that allow the user to create and edit models in user-defined languages. Examples of these approaches are AToM^3 [47], MetaEdit+ [88] and Pounamu [211]. These tools are complete environments while our work describes only one component that should work with any other editing and transformation component based on the OMG standards.

Many researchers have studied the definition of new model transformation languages and tools that support in one way or another the OMG modeling standards. Among the general-purpose transformation languages are the relational approaches by David Akehurst and Stuart Kent [46] and YATL [146] by Octavian Patrascoiu, both of which use OCL for the declarative expressions. The relational approach is further investigated by Hausmann and Kent in [71]. There is also a special graph grammar system in VIATRA by Dániel Varró [181], which relies on graph grammars instead of OCL and has operational semantics. Jean Bézivin proposes the Atlas Transformation Language [21], which has tool support in the Eclipse Modeling Framework [52]. The MOLA transformation language by Audris Kalnins, Janis Barzdins and Edgars Celms [86] has a graphical imperative programming language with pattern-based transformation rules. Perhaps the most important general-purpose transformation language is the QVT language [133] from OMG. Also, the work presented in this chapter can be seen as a specific instance of a model composition framework, as described in [20].

We have decided to use a special language to define the diagram mappings and a special tool to apply these mappings due to the need for specific algorithms to create and reconcile existing diagrams. The diagram mapping language and corresponding tool support should work preserving as much information as possible from existing diagrams. Also, we consider that DIML mappings are more succinct than the equivalent transformation in many general transformation languages, due to the use of OCL queries and expressions to describe many cases in a single rule.

An alternative approach to implement the idea presented in this chapter is to use a general model composition framework [20], for example the Atlas Model Weaver [55]. In this case, a composition framework would take one or two user models and the DIML mappings as input models and recreate the necessary diagrams by implementing the algorithms presented in this paper using transformation rules that are specific to the composition framework. It would be an interesting and worthwhile experiment to compare different model composition frameworks with DIML. There are several properties that should be analyzed, such as the size of the mapping definitions, understandability, maintainability, any expressivity restrictions and the need for traceability mechanisms and separate tracing models. Model composition and weaving frameworks are being researched and developed at a fast pace, for example [55, 118, 18, 93]. Similarly, there is active research in model traceability [24, 95].

There are other approaches and tools that support the reconciliation of abstract models and their concrete models. However, none of them seem to support DI. This makes comparison awkward, since one of our goals has been the usage of OMG standards. The diagram definition facility [35] by Edgars Celms, Audris Kalnins and Lelde Lace specifically targets mapping to diagrams and uses its own diagram metamodel, although one different from DI in that it requires subclassing for each diagram element, whereas DI is specified so that a language developer should not inherit from DI classes. The work by Frédéric Fondement and Thomas Baar [57] formalizes the relationship between abstract and concrete syntaxes with OCL expressions using their own concrete syntax. While the ideas presented are interesting, it does not yet have any tool support and although diagram reconciliation is recognized as a problem, the authors do not offer any solution. In fact, our work addresses some of their concerns on DI.

8.8 Conclusions

We have described an approach to create new diagrams from the abstract syntax of a model and bring existing diagrams up-to-date after the execution of a transformation that updates the abstract syntax of a model. We consider that these problems are important because they are necessary to allow the automatic transformation of visual diagrammatic languages.

Our approach is based on a mapping language called DIML that describes the relation between the abstract syntax and the concrete syntax of a model defined according to the OMG standards. Our solution enables the creators of model transformation languages and tools to ignore diagrams and focus on the semantic information stored in models. One of the most important characteristics of DIML is that it is independent of the modeling and transformation language. This allows us to define metamodel-independent algorithms to create and update diagrams. On the other hand, DIML is specific to DI, the OMG diagram interchange standard. Other alternative diagram interchange languages are discussed in [169].

There are several practical benefits to reconciliation and using DIML. We realize that we have made several engineering decisions in creating a domain-specific weaving metamodel, and that it has limitations. At this time we do not know for certain to what extent the decisions are a hindrance, and how much they are beneficial. For example, by only allowing OCL expressions to query the chain of parent DI elements we establish that DIML rules are hierarchical. This makes the algorithms relatively simple, and we do not have to, for example, calculate any graph isomorphisms.

There are also some limitations in our work as described in Sec. 8.3.5, the most important limitation being the fact that a DIML rule cannot map relations, only a single class.

We have validated our approach by implementing it in an open source modeling tool called Coral. Due to the common interchange format and adherence to OMG standards we can use both Poseidon and Coral to edit, transform and interchange models and diagrams. We believe that this is an indication of the viability of DI and the DIML language.

DIML and the diagram reconciliation component do not contain all the functionality necessary for a full interactive editor since they lack a user interface, a layout and a rendering engine. We assumed that these problems are solved by other independent components. In some visual languages, such as UML class diagrams, the layout is a question of aesthetics. However, in other languages, such as UML sequence diagrams, the layout also conveys semantic information since the passage of time is represented in the vertical axis of a diagram. Therefore, we would like to extend the DIML language to include diagram layout constraints so that the layouting step can be integrated with the diagram reconciliation step.

Chapter 9

Conclusions

In this final chapter we succinctly summarize the contents of the thesis and include possible ideas for future work. We also give a short overview of how well our vision of modeling is in line with the vision of OMG and its standards, and to what extent OMG standards fulfill the role they should and where they are lacking. We finally conclude with some general remarks about the field of software engineering itself.

When the work of this thesis started, the state of modeling was in some ways quite different from its current situation. Modeling tools were almost always incompatible with each other, and there was a lot of confusion about what modeling is. Nowadays, many commercial modeling tools by large vendors are based on the Eclipse Modeling Framework, which has won a lot of ground, with the DSL Tools by Microsoft being a considerable commercial opponent. Eclipse itself provides an enormous number of plugins and several modeling-related tools are being developed for it. This has the benefit that more research prototypes can be used together, but a problem is still that Eclipse is only a de facto standard.

In Chapter 2, we compared several modeling languages. We use this study and knowledge from other modeling and programming languages throughout the other chapters. There is no definite way to say a priori which feature in a modeling language is useful. Rather, a feature must be tried out in practice and must be shown to have an effect on the expressiveness and versatility in the modeling framework. Also, features at the modeling language level must be very clearly defined, as any inaccuracies spreads to the languages and then to the models.

It can be said that this work defines features using two different constructs: basic building blocks from metametamodels and additional constraint rules. This creates a partially unnecessary schism, where we need to decide whether a newly invented construct should be a basic building block or merely something that is tagged along using a constraint. The latter solution does not provide a first-class concept in the metametamodel in the same way the former does. A similar problem is found in textual programming languages, where the facilities provided by the language and the extensions of the user have very differing syntax. Only LISP and its variants have solved this by presenting everything consistently using *s-expressions* [112], thereby also not having any syntactic sugar.

An interesting proposition is given by Guy Steele [167], where he argues that language and program must evolve together to solve the problem at hand, and that user-defined additions to the language must be indistinguishable from the constructs provided by the language. The current modeling frameworks do not support this idea well in practice.

The metamodeling language presented in Chapters 3 and 4 was accomplished using a set-theoretic representation of the class and object layers. The most important contribution in those chapters is the understanding gained that combining several new features into a metamodeling language must take the new and the existing features into account as a whole, lest *feature interaction* makes the result nonfunctioning. However, the various metamodel and model constraints that have been defined have not been formally proven correct with respect to the model operations. Proving correctness would be an important step in ensuring that the SMD language is solid, and further proofs of other algorithms operating on models could then be feasible. Similar work has already been done by Iman Poernomo to prove a MOF-like metamodeling language without support for subsets [151]. There is also a considerable effort in creating a formal semantics for UML [26], and for example [81] shows a formalization of UML statecharts using concurrent regular expressions.

The most novel part of the chapters is the structural definition of subsets and unions and operations involving subset and union slots. However, the property redefinition concept is not formalized and must be considered an omission in light of its frequent usage in the definition of UML 2.0 and MOF 2.0. There exists work in type-safe covariant specialization [157] which could be incorporated into a modeling framework as well. We described an implementation of our metamodeling language called SMD in Chapter 5 that supports subsets and derived unions as formalized by us.

On the surface, the topic of Chapter 6 seems mundane. Serialization ought to be a rather easy topic. Yet experience from using several tools suggests that incompatibilities abound as there is no viable way to certify compatibility. One of the greatest benefits of the success of tools built using EMF is that at least serialization of models seems to follow the same peculiarities, since the tools use the same technology underneath. Although we remarked on several issues with XMI, a subset of the standard could be used.

Some parts of XMI are not useful and not necessary. In particular, the parts describing model differences are underspecified and even if they worked, they would not be a good solution. A better solution is a separate metamodel that can describe differences between models. Model difference calculation is useful for version control, which we discussed in Chapter 7. It is interesting that although we have a long history of version control both in theory and practice in the textual domain—

Walter Tichy developed RCS [176] in the beginning of the 1980's—version control of models is still not as straightforward. We note that conflict resolution is still an unsolved problem, as it also is in the textual domain. Furthermore, displaying a model conflict is harder than displaying a textual conflict. There are no widely used solutions for a model collaboration environment and model repository. As briefly discussed, the XMI serialization standard does not adequately support describing a difference between models.

Chapter 8 described a weaving metamodel called DIML, which aids in mapping abstract syntax to concrete syntax. It is an interesting and current topic considering the Model View to Diagram Request for Proposal [140] from OMG. It would be very fruitful to assess the usability of the idea of DIML with the various general-purpose weaving languages, such as the Atlas Model Weaver (AMW) [55], or a general-purpose transformation language such as QVT [133] or ATL [21]. This would be accomplished by using these languages to solve the same problem as DIML does.

This thesis has strived to cover a broad range of topics, perhaps sometimes sacrificing depth for breadth. The topic of any single chapter is important and complex enough for more research in the future. The aim of the work has been to analyze the OMG standards from both a theoretical and practical context. Our long-term research goal is furthering the prevalence of modeling as the primary way to build software engineering artifacts, and the ubiquity of powerful and usable modeling tools. The current OMG standards are severely lacking in both theoretic viability and practical usability, and the standards are not true to the vision of OMG.

The author's view is in some ways similar to the one provided by OMG. The information of a system is given by an abstract model or models whereas separate views, also called diagrams or concrete models, display the abstract model to the developer. There is a single serialization technology that is capable of representing any kind of model or models. However, a single metamodel in the form of UML or profiles for UML is limiting, and the experience in our research group suggests that domain-specific languages bring benefits that UML cannot. The separation of an MDA [130] application into a single Platform-Independent Model (PIM) which is transformed into a Platform-Specific Model (PSM) by combining it with a Platform Model (PM) sounds too limiting, and anecdotal experience suggests that this is also the opinion of the research community at large.

An engineering aspect of this thesis has been to closely follow improvements in the OMG standards without necessarily being limited by them. It is unfortunate that standards consortia nowadays seem overeager to standardize technologies which have yet to mature. This problem is seen in several of the OMG standards, such as MOF and DI.

The evolution of the definition of MOF is highly unsatisfying. Although problematic, the 1.x series of small, clumsy and badly-specified standards did not hinder anybody with an object-oriented background, and thus many problems with them were not problems for research. However, version 2.0 feels less useful because it contains novel constructs such as subsets, redefinitions and package merges without a precise underlying mathematical model. We believe that leaner metamodeling languages will become more popular since MOF fails to deliver value due to its incompleteness. Examples of such metamodeling languages are ECORE [28], KM3 [84] or Kermeta [90]. GXL provides even more flexibility than any of the metamodeling languages mentioned. Some of the novel constructs from MOF might be adopted once their meaning has been defined. A formalization of subsets and unions was given in Chapters 3 and 4, and we hope that it aids in a standardized formalization of the features.

Assuming a working metamodeling language, we could define the syntax of abstract models by creating a metamodel. Visualizing this should use the DI standard for interoperability reasons. However, DI has received extremely little support from vendors and all problems with using DI are thus still unclear. The problem of which DI constructs are valid for a model is clear: DI does not specify a mapping language or similar feature with which a metamodel developer could define the valid diagrams of a model. Our example solution for this problem is DIML. However, any solution could be used and DIML merely serves as a case study. An interoperable diagram standard feels like the greatest concern for practical modeling to go forward. Currently, vendor lock-in hinders wholehearted adoption of software modeling.

There is a lot to software modeling that this thesis does not even try to address. Especially metamodel semantics, model transformations and metamodel evolution come to mind. Considering the MDA effort itself and how many different transformation languages are available, model transformations could be seen as the heart of modeling. Also, since graph transformation approaches such as single-pushout or double-pushout [160] seem to offer more highlevel constructs to transform models (programs) than many transformation languages for textual streams, model transformations can be seen as one of the primary reasons why they could provide modeling as a viable alternative to traditional textual software engineering.

It is the author's hope that this thesis would spur both further research and solutions to software modeling and an interest from the modeling community to provide working tools along with breakthroughs or ideas in the theory of software modeling. Our current state of not having tools of sufficient interoperability for empirical validation is disillusioning and disheartening.

Software engineering is, like any other engineering discipline, subordinate to the constraints of many masters. Constraints from computer science and mathematics dictate what could be possible to accomplish and must form the core of software engineering. But to evolve software engineering we must be able to take several other aspects into account, such as budgetary constraints and the people who are our developers, managers and customers. This means software engineering solutions must be validated empirically, and in the case of modeling this validation requires tools. Realizing improved software engineering practices might be our biggest obstacle for years to come.

Appendix A: Mathematical Preliminaries

This appendix summarizes the mathematical notation used in this thesis, with some small examples.

Set Theory

We use naïve set theory throughout the thesis. Sets of primitive data values are denoted with calligraphic letters: $\mathbb{B}$ is the set of boolean values, $\mathbb{Z}^{0+}$ is the set of integers $0, 1, 2, \ldots$ and $\mathbb{Z}^{+}$ is set of the set of integers $1, 2, 3, \ldots$.

The empty set is denoted with $\emptyset$ or $\{\ \}$. A set with elements x, y and z is denoted with $\{x,y,z\}$. The expression $\#S$ is the number of elements in a finite set S.

We use the notation $A \subset B$ to denote that A is a strict subset of B. We use $A \subseteq B$ to denote the possibility that A is a subset of or equal to B.

The expression $\mathcal{P}(A)$ is the powerset of a set A, i.e., the set of all possible subsets. For example, the powerset of $\{1,2,3\}$ is $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\ \}$.

Set comprehensions are denoted with $\{g(x) \cdot f(x)\}$, which returns a set of values $g(x)$ where $f(x)$ is true (for all possible legal values of x). The notation $\bigcup\{g(x) \cdot f(x)\}$ denotes the set consisting of all elements in all sets $g(x)$ where $f(x)$ is true.

Sequences

A sequence $(A, \prec)$ is an ordered set or array of elements A. It is essentially like a set of elements, except that all elements have a unique position in the sequence, that position denoted with an integer $i \in \mathbb{Z}^{0+}$. We denote $a \prec_x b$ if element a precedes element b in a specific sequence x. We denote $a \preceq_x b$ if a precedes b or if $a = b$. Note that two elements can be in different orders in different sequences.

For sequences, $\triangleleft$ denotes sequence concatenation and $t[a:b]$ denotes the sequence of elements $t[a], \ldots, t[b-1]$.

The function reverse(s) returns a new sequence with the same elements as in s but in reverse order.

Sequence comprehensions are denoted with $[g(x) \cdot f(x)]$, which returns a sequence of values $g(x)$ where $f(x)$ is true (for all possible legal values of x). Where possible, any data structures in $f(x)$ are visited in order. For example, $[x^2 \cdot x \in [1,2,3,4]]$ returns the sequence $[1,4,9,16]$.

Binary Relations

A binary relation R is a set of pairs (a,b). A reflexive relation is such that $(\forall a \cdot (a,a) \in R)$. A transitive relation is defined by $(a,b) \in R \wedge (b,c) \in R \Rightarrow (a,c) \in R$.

If we have a set of pairs R whose values are of the same domain A, we can create the transitive closure R^+ by taking the smallest transitive relation over the domain of the values that still contains R. The reflexive closure of R is $R^= \overset{\text{def}}{=} R \cup \{(a,a) \cdot a \in A\}$. The reflexive transitive closure of R is $R^* \overset{\text{def}}{=} R^{+=}$.

Functions

A function $f : A \rightarrow B$ maps an element of the domain set A to an element of the range set B. A partial function $f : A \nrightarrow B$ might not be defined for all elements of A. An unnamed function which maps x to y is denoted with $\{x \rightarrow y\}$.

Function application is denoted with $f(x)$, i.e., the function f is called with the parameter x. The domain of a function is acquired with Dom(f). The range of a function is acquired with Range(f).

Function Update

$f[S]$ where S is a set of pairs $x \rightarrow y$ values returns a new function f' such that $(\forall z \cdot z \notin \text{Dom}(S) \Rightarrow f'(z) = f(z)) \wedge (\forall z \cdot z \in \text{Dom}(S) \Rightarrow f'(z) = y)$. $f[x \rightarrow y]$ behaves like $f[\{x \rightarrow y\}]$.

$G \lhd F$ is the domain restriction operator and returns a new function identical with F but restricted to the domain of a set G, i.e., $G \lhd F \overset{\text{def}}{=} \{x \rightarrow y \cdot x \in G \wedge x \rightarrow y \in F\}$. $G \blacktriangleleft F$ is the domain subtraction operator and returns a new function identical with F but restricted to the domain which is not in set G, i.e., $G \blacktriangleleft F \overset{\text{def}}{=} \{x \rightarrow y \cdot x \notin G \wedge x \rightarrow y \in F\}$.

Graphs

A graph $G = (V,E)$ is a set of vertices V and a set of edges E such that $V \cap E = \emptyset$. An edge is a tuple (v_1, v_2) where v_1 and v_2 are vertices and connects the two vertices together. A directed graph denotes that an edge has a direction.

Partial Orders

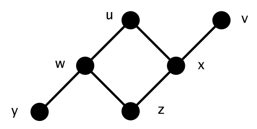
A partially ordered set or *poset* $(A, \subseteq_A)$ is a set A and a binary operator $\subseteq_A$. The operator determines the partial ordering of elements; given $a \in A$ and $b \in A$, the operation $a \subseteq_A b$ is true if a occurs before b in the ordering, otherwise false. The expression $a \,||_A\, b$ means that a and b are independent and cannot be compared in the partial order; it is equivalent to $\neg(a \subseteq_A b) \wedge \neg(b \subseteq_A a)$.

We denote by $a \prec_A b$ the fact that a is a "directly below" b in a partial order, i.e., $a \prec_A b \overset{\text{def}}{=} a \subset_A b \wedge \neg(\exists c \cdot c \neq a \wedge c \neq b \wedge a \subset_A c \subset_A b)$.

The partial order should not be confused with the subset operator $\subseteq$. The former is an arbitrary function that determines the partial order, whereas the latter has only one definition in set theory.

Hasse Diagrams

Hasse diagrams are graphical renderings of a poset $(A, \subseteq_A)$. Every element in A is drawn as a node such that a node representing element $x \in A$ is drawn visually lower than another element $y \in A$ if and only if $x \subseteq_A y$. Furthermore, a line is drawn from x to y if and only if $x \prec_A y$. An example of a Hasse Diagram of the set $A = \{u, v, w, x, y, z\}$ and the relation $\subseteq_A = \{(w,u), (x,u), (x,v), (y,w), (z,w), (z,x)\}$ is shown in the following figure:



Appendix B: The Simple Metamodel Description Language

This appendix summarizes the set-theoretical definition of SMD from Chapter 3.

Metamodel Formalization

We define the metamodeling language by:

$$\begin{aligned}
&ML_S \overset{\text{def}}{=} (C, \text{generalizations}, P, \text{owner}, \text{type}, \text{characteristics}_S)\\
&\text{effectiveProperties}_S(c) = \bigcup\{\text{properties}(d) \cdot c \subseteq_c d\}\\
&\text{effectiveType}_S(p) = \{c \in C \cdot c \subseteq_c \text{type}(p)\}
\end{aligned}$$

The characteristics of the properties, including subsets and strict unions, is defined by:

$$\begin{aligned}
\text{characteristics}_S \overset{\text{def}}{=} (&\text{lower}, \text{upper}, \text{ordered}, \text{composite}, \text{opposite}, \text{supersets},\\
&\text{strictUnion})
\end{aligned}$$

The following metamodel constraints are defined:

Metamodel Constraint 1 *Property Multiplicity:* $(\forall p \in P \cdot \text{lower}(p) \leq \text{upper}(p))$

Metamodel Constraint 2 *Opposite properties:* $(\forall p \in P \cdot \text{opposite}(p) \neq \Omega \Rightarrow p = \text{opposite}(\text{opposite}(p)) \wedge \text{opposite}(p) \neq p)$

Metamodel Constraint 3 *Both properties in a relation cannot be composite:* $(\forall p \in P \cdot \text{composite}(p) \wedge \text{opposite}(p) \neq \Omega \Rightarrow \neg\text{composite}(\text{opposite}(p)))$

Metamodel Constraint 4 *No infinite chain of compositions:* $(\forall c_1, \ldots, c_n, c_{n+1} \in C \cdot (\forall i \cdot 1 \leq i \leq n \Rightarrow (\exists p \in \text{effectiveProperties}(c_i) \cdot \text{composite}(p) \wedge \text{owner}(p) = c_i \wedge c_{i+1} = \text{type}(p) \wedge \text{lower}(p) \geq 1)) \Rightarrow c_1 \neq c_{n+1}), \quad \forall n \geq 1$

Metamodel Constraint 5 *Generalization is acyclic:* $\neg(\exists e \in C \cdot (e,e) \in \{(c,d) \cdot d \in \text{generalizations}(c)\}^+)$

Metamodel Constraint 6 *Upper multiplicity in subset properties:* $(\forall p \in P \cdot (\forall q \in \text{supersets}(p) \cdot \text{upper}(p) \leq \text{upper}(q)))$

Metamodel Constraint 7 *Subset only from owner or its superclasses* $(\forall p, q \in P \cdot p \subseteq_p q \Rightarrow \text{owner}(p) \subseteq_c \text{owner}(q))$.

Metamodel Constraint 8 *The property superset relation is acyclic:* $\neg(\exists e \in P \cdot (e,e) \in \{(p,q) \cdot q \in \text{supersets}(p)\}^+)$

Metamodel Constraint 9 *The opposite of a subset property must be a subset:* $(\forall p, q \in P \cdot p \subseteq_p q \wedge \text{opposite}(p) \neq \Omega \Rightarrow \text{opposite}(p) \subseteq_p \text{opposite}(q))$

Metamodel Constraint 10 *No circular transitive composition with subsets:* $(\forall p \in P \cdot \text{composite}(p) \Rightarrow \neg(\exists q \in P \cdot \text{opposite}(q) \neq \Omega \wedge p \subset_p q \wedge \text{composite}(\text{opposite}(q))))$

Metamodel Constraint 11 *Ordering characteristics are same in property poset:* $(\forall p \in P \cdot (\forall q \in \text{supersets}(p) \cdot \text{ordered}(q) = \text{ordered}(p))$. *This metamodel constraint is only necessary for the model operations.*

Model Formalization

The models are defined by:

$$M \stackrel{\text{def}}{=} (E, \text{class}, S, \text{property}, \text{slotowner}, \text{elements})\}$$

The following model constraints are defined:

Model Constraint 1 *Valid slots in element (1):* $(\forall e \in E \cdot (\forall s \in \text{slots}(e) \cdot (\text{property}(s)) \in \text{effectiveProperties}(\text{class}(e))))$

Model Constraint 2 *Valid slots in element (2):* $(\forall e \in E \cdot (\forall p \in \text{effectiveProperties}(\text{class}(e)) \cdot (\exists! s \in \text{slots}(e) \cdot \text{property}(s) = p)))$

Model Constraint 3 *Class of elements in a slot:* $(\forall s \in S \cdot (\forall e \in \text{elements}(s) \cdot \text{class}(e) \in \text{effectiveType}(\text{property}(s))))$

Model Constraint 4 *Valid number of elements in a slot:* $(\forall s \in S \cdot \text{lower}(\text{property}(s)) \leq \#s \leq \text{upper}(\text{property}(s)))$

Model Constraint 5 *Bidirectionality of slots:* $(\forall s \in S \cdot \text{opposite}(\text{property}(s)) \neq \Omega \Rightarrow (\forall e' \in \text{elements}(s) \cdot (\exists! s' \in S \cdot \text{slotowner}(s') = e' \wedge \text{opposite}(\text{property}(s')) = \text{property}(s) \wedge \text{slotowner}(s) \in \text{elements}(s'))))$

Model Constraint 6 *Overridden by model constraint 11.*

Model Constraint 7 *Composition is acyclic:* $(\forall e \in E \cdot e \notin \text{parentchain}(e))$

Model Constraint 8 *Unordered slots:* $(\forall r, s \in S \cdot r \subseteq_s s \wedge \neg\text{ordered}(\text{property}(s))$ $\Rightarrow \text{elements}(r) \subseteq \text{elements}(s))$

Model Constraint 9 *Ordered slots:* $(\forall x, y \in E, r, s \in S \cdot x \in \text{elements}(r) \wedge y \in$ $\text{elements}(r) \wedge x \preceq_r y \wedge r \subseteq_s s \wedge \text{ordered}(\text{property}(s)) \Rightarrow x \in \text{elements}(s) \wedge y \in$ $\text{elements}(s) \wedge x \preceq_s y)$

Model Constraint 10 *Strict union:* $(\forall s \in S \cdot \text{strictUnion}(\text{property}(s)) \Rightarrow$ $\text{elements}(s) = \bigcup\{\text{elements}(r) \cdot r \prec_s s\}$

Model Constraint 11 *(Subset) Only in one composite slot:* $(\forall e \in E \cdot \neg(\exists s_1, s_2 \cdot$ $\text{property}(s_1) \,||_p\, \text{property}(s_2) \wedge \text{composite}(\text{property}(s_1)) \wedge \text{composite}(\text{property}(s_2))$ $\wedge\, e \in \text{elements}(s_1) \wedge e \in \text{elements}(s_2))$

Bibliography

[1] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] Adaptive, Inc. Adaptive Repository. `http://www.adaptive.com/resources_papers/technology.html`, visited April 24th, 2007.

[3] Adobe Systems, Inc. *PostScript Language Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, third edition, 1999.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques and Tools*. Addison-Wesley, January 1986.

[5] M. Alanen. A Meta Object Facility-Based Model Repository With Version Capabilities, Optimistic Locking and Conflict Resolution. Master's Thesis in Computer Engineering, Department of Computer Science, Åbo Akademi University, Turku, Finland, November 2002.

[6] A. Albano, G. Ghelli, and R. Orsini. A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language. In *Proceedings of the 17th Conference on Very Large Databases*. Morgan Kaufman Publishers Inc., 1991.

[7] J. Álvarez, A. Evans, and P. Sammut. MML and the Metamodel Architecture. In J. Whittle, editor, *WTUML: Workshop on Transformation in UML 2001*, April 2001.

[8] C. Amelunxen, T. Rötschke, and A. Schürr. Graph Transformations with MOF 2.0. In H. Giese and A. Zündorf, editors, *Fujaba Days 2005*, September 2005.

[9] C. Atkinson and T. Kühne. Rearchitecting the UML infrastructure. *ACM Trans. Model. Comput. Simul.*, 12(4):290–321, 2002.

[10] T. Baar. Metamodels without Metacircularities. *L'Objet*, 9(4):95–114, 2003.

[11] R.-J. Back, D. Björklund, J. Lilius, L. Milovanov, and I. Porres. A Workbench to Experiment on New Model Engineering Applications. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language*, volume 2863 of *Lecture Notes in Computer Science*, October 2003.

[12] R.-J. Back, J. Eriksson, and M. Myreen. Verifying Invariant Based Programs in the SOCOS Environment. In *Teaching Formal Methods: Practice and Experience (BCS Electronic Workshops in Computing)*. BCS-FACS, December 2006.

[13] R.-J. Back, J. Eriksson, and M. Myreen. Testing and Verifying Invariant Based Programs in the SOCOS Environment. In *Proceedings of the International Conference on Tests And Proofs (TAP)*. Springer-Verlag, February 2007.

[14] R.-J. Back, J. Grundy, and J. von Wright. Structured calculational proof. Technical Report 65, Turku Center for Computer Science, November 1996.

[15] R.-J. Back and J. von Wright. *Refinement Calculus - A Systematic Introduction*. Springer-Verlag, 1998. ISBN 0387984178.

[16] F. Barbier, B. Henderson-Sellers, A. Le Parc, and J.-M. Bruel. Formalization of the Whole-Part Relationship in the Unified Modeling Language. *IEEE Trans. Software Eng.*, 29(5):459–470, 2003.

[17] L. Baresi and R. Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. Graph Transformation - First International Conf., ICGT 2002, Barcelona, Spain*, volume 2505 of *LNCS*. Springer, 2002.

[18] G. Beneken, F. Marschall, and A. Rausch. A Model Framework Weaving Approach - Position Paper. In *Proceedings of the First Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD at the 19th European Conference on Object-Oriented Programming (ECOOP 2005)*, July 2005.

[19] J. Bézivin. On the Unification Power of Models. *Springer Journal on Software and Systems Modeling*, 3(4), 2004.

[20] J. Bézivin, S. Bouzitouna, M. Didonet Del Fabro, M.-P. Gervais, F. Jouault, D. Kolovos, I. Kurtev, and R. Paige. A Canonical Scheme for Model Composition. In A. Rensink and J. Warmer, editors, *Model Driven Architecture—Foundations and Applications: Second European Conference, ECMDA-FA 2006, LNCS 4066*, pages 346–360. Springer-Verlag, 2006.

[21] J. Bézivin, E. Breton, G. Dupé, and P. Valduriez. The ATL Transformation-based Model Management Framework. Technical Report 03.08, University of Nantes, France, 2003.

[22] J. Bézivin, G. Hillairet, F. Jouault, I. Kurtev, and W. Piers. Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In R. Johnson and R. P. Gabriel, editors, *Proceedings of the International Workshop on Software Factories at OOPSLA 2005*, San Diego, California, USA, October 2005. ACM Press.

[23] G. Bierman and A. Wren. First-class relationships in an object-oriented language. In *Workshop on Foundations of Object-Oriented Languages (FOOL 2005)*, January 2005.

[24] L. Bondé, P. Boulet, and J.-L. Dekeyser. Traceability and interoperability at different levels of abstraction in model transformations. In *Forum on Specification and Design Languages, FDL'05*, September 2005.

[25] F. P. Brooks, Jr. No Silver Bullet. In H.-J. Kugler, editor, *Proceedings of the IFIP Tenth World Computing Conference*, pages 1069–1076. Elsevier Science B.V., 1986.

[26] M. Broy, M. L. Crane, J. Dingel, A. Hartman, B. Rumpe, and B. Selic. 2nd UML 2 Semantics Symposium: Formal Semantics for UML. In T. Kühne, editor, *MoDELS 2006 Workshops*, volume 4364 of *LNCS*, pages 318–323. Springer, 2006.

[27] C. Brun et al. The EMF Compare website. `http://www.eclipse.org/emft/projects/compare/`, visited April 24th, 2007.

[28] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, August 2003.

[29] J. Cabot and E. Teniente. Determining the Structural Events that May Violate an Integrity Constraint. In T. Baar, A. Strohmeier, A. Moreira, and S. J. Mellor, editors, *UML 2004 - The Unified Modeling Language. Model Languages and Applications. 7th International Conference, Lisbon, Portugal*, volume 3273 of *LNCS*, pages 320–334. Springer, October 2004.

[30] J. Cabot and E. Teniente. Computing the Relevant Instances that May Violate an OCL Constraint. In O. Pastor and J. Falcão e Cunha, editors, *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005, Porto, Portugal*, volume 3520 of *LNCS*, pages 48–62. Springer, June 2005.

[31] CAE Specification. DCE 1.1: Remote Procedure Call, 1997. Available at `http://www.opengroup.org/onlinepubs/9629399/toc.htm`, visited April 24th, 2007.

[32] G. Castagna. Covariance and Contravariance: Conflict without a Cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[33] R. Castello, R. Mili, and I. G. Tollis. Automatic layout of statecharts. *Software: Practice and Experience*, 32(1):25–55, 2001.

[34] P. Cederqvist et al. Version Management with CVS, 1992. Available at `http://www.cvshome.org/`, visited April 24th, 2007.

[35] E. Celms, A. Kalnins, and L. Lace. Diagram Definition Facilities Based on Metamodel Mappings, October 2003. Invited talk at the Third OOPSLA Workshop on Domain-Specific Modeling.

[36] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, volume 4111 of *LNCS*, pages 342–363. SV, 2006.

[37] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, 1996.

[38] N. Chomsky. Three models for the description of language. *Information Theory, IEEE Transactions on*, 2(3):113–124, 1956.

[39] M. C. Chu-Carroll and S. Sprenkle. Coven: Brewing Better Collaboration through Software Configuration Management. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering: twenty-first century applications*, November 2000.

[40] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. A Domain-Specific Modeling Language for Model Differences. Technical Report TR 005/2006, Dipartimento di Informatica, Università di L'Aquila, Italy, 2006.

[41] A. Cicchetti, D. Di Ruscio, and A. Pierantonio. Composition of Model Differences. In A. Kleppe, editor, *Proceedings of the First European Workshop on Composition of Model Transformations (CMT 2006)*, pages 9–14, July 2006.

[42] T. Clark, A. Evans, and S. Kent. The Metamodelling Language Calculus: Foundation Semantics for UML. In *Proceedings of the Fundamental Aspects of Software Engineering (FASE)*, pages 17–31, 2001.

[43] T. Clark, A. Evans, P. Sammut, and J. Willans. *Applied Metamodelling: A Foundation for Language-Driven Development*. 2005. Available at `http://www.xactium.com/`, visited April 24th, 2007.

[44] G. Clemm, J. Amsden, T. Ellison, C. Kaler, and J. Whitehead. Versioning Extensions to WebDAV, RFC 3253, March 2002. Available at `http://www.ietf.org/rfc/rfc3253.txt`.

[45] C. Connell. Why Software Engineering Is Not B.S. *Dr. Dobb's Journal*, July 2001.

[46] D. H. Akehurst and S. Kent and O. Patrascoiu. A relational approach to defining and implementing transformations between metamodels. *Software and System Modeling*, 2(4):215–239, 2003.

[47] J. de Lara Jaramillo, H. Vangheluwe, and M. A. Moreno. Using Meta-Modelling and Graph Grammars to Create Modelling Environments. *Electronic Notes in Theoretical Computer Science*, 72(3), 2003.

[48] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: an Overview. In *European conference on object-oriented programming on ECOOP '87*, pages 151–170, London, UK, 1987. Springer-Verlag.

[49] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A Theorem Prover for Program Checking. *ACM*, 52(3):365–473, 2005.

[50] R. Ducournau, M. Habib, M. Huchard, and M.-L. Mugnier. Proposal for a Monotonic Multiple Inheritance Linearization. In *Proceedings of the Ninth Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 164–175, 1994.

[51] J. Ebert, B. Kullbach, and A. Winter. Grax: Graph exchange format. In *Workshop on Standard Exchange Formats (WoSEF) at (ICSE'00)*, 2000.

[52] The Eclipse Modeling Framework website. `http://www.eclipse.org/emf`, visited April 24th, 2007.

[53] M. Eiglsperger, M. Kaufmann, and M. Siebenhaller. A topology-shape-metrics approach for the automatic layout of UML class diagrams. In *Proceedings of the 2003 ACM symposium on Software visualization*. ACM Press, 2003.

[54] Elver Project. Teneo. `http://www.elver.org/`, visited April 24th, 2007.

[55] M. Didonet Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. AMW: A Generic Model Weaver. In *Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM05)*, 2005.

[56] R. Fielding, J. Gettys, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1, RFC 2616, June 1999. Available at `http://www.ietf.org/rfc/rfc2616.txt`.

[57] F. Fondement and T. Baar. Making Metamodels Aware of Concrete Syntax. In *European Conference on Model Driven Architecture (ECMDA)*, volume 3748 of *LNCS*, pages 190 – 204, 2005.

[58] R. France and B. Rumpe. Domain specific modeling, Editorial. *Springer International Journal on Software and Systems Modeling*, 4(1), 2005.

[59] Free Software Foundation, Inc. GNU General Public License (version 2), June 1991. Available at `http://www.gnu.org/licenses/gpl.html`.

[60] G. Génova, C. Ruiz del Castillo, and J. Lloréns. Mapping UML Associations into Java Code. *Journal of Object Technology*, 2(5):135–162, 2003.

[61] Gentleware. The Poseidon for UML product. `http://www.gentleware.com/`, visited April 24th, 2007.

[62] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. HTTP Extensions for Distributed Authoring—WEBDAV, RFC 2518, February 1999. Available at `http://www.ietf.org/rfc/rfc2518.txt`.

[63] C. Gonzalez-Perez and B. Henderson-Sellers. A Powertype-based Metamodelling Framework. *Software and Systems Modeling*, 5:72–90, April 2006. doi:10.1007/s10270-005-0099-9.

[64] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[65] The GraphViz website. `http://www.graphviz.org/`, visited April 24th, 2007.

[66] J. Greenfield, K. Short, S. Cook, S. Kent, and J. Crupi. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004.

[67] The GUPRO website. `http://www.uni-koblenz.de/~gupro/`, visited April 24th, 2007.

[68] The GXL Validator website. `http://www.uni-koblenz.de/FB4/Contrib/GUPRO/Site/Downloads/index\_html?project=gxl`, visited April 24th, 2007.

[69] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[70] D. Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, May 1988.

[71] J. H. Hausmann and S. Kent. Visualizing model mappings in UML. In *SoftVis '03: Proceedings of the 2003 ACM symposium on Software visualization*, pages 169–178, New York, NY, USA, 2003. ACM Press.

[72] S. Haustein, February 2004. Discussion on the mailing-list puml-list@cs.york.ac.uk.

[73] B. Henderson-Sellers and F. Barbier. Black and White Diamonds. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 550–565. Springer, 1999.

[74] The Hibernate website. `http://www.hibernate.org/`, visited April 24th, 2007.

[75] R. C. Holt, A. Schürr, S. E. Sim, and A Winter. GXL: A graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2):149–170, April 2006.

[76] S. Holzner. *Eclipse*. O'Reilly Media, first edition, May 2004. ISBN 0596006411.

[77] D. Ilic, E. Troubitsyna, L. Laibinis, and S. Leppänen. Formal Verification of Consistency in Model-Driven Development of Distributed Communicating Systems and Communication Protocols. Technical Report 777, TUCS, June 2006.

[78] Information Sciences Institute, University of Southern California. Transmission Control Protocol, Darpa Internet Program, Protocol Specification—RFC 793, September 1981. Available at `http://www.ietf.org/rfc/rfc793.txt`.

[79] S. Iyengar et al. Java Metadata Interface (JMI) Specification API 1.0. Available at `http://java.sun.com/`, June 2002.

[80] H. Jagadish, S. Al-Khalifa, A. Chapman, L. Lakshmanan, A. Nierman, S. Paparizos, J. Patel, D.Srivastava, N. Wiwatwattana, Y. Wu, and C. Yu. TIMBER: A native XML database. *The VLDB Journal - The International Journal on Very Large Data Bases*, 11:274–291, December 2002.

[81] S. Jansamak and A. Surarerks. Formalization of UML statechart models using Concurrent Regular Expressions. In *ACSC '04: Proceedings of the 27th Australasian conference on Computer science*, pages 83–88, Darlinghurst, Australia, 2004. Australian Computer Society, Inc.

[82] J. Jiang and T. Systä. Exploring Differences in Exchange Formats – Tool Support and Case Studies. In *Seventh European Conference on Software Manteinance and Reengineering*. IEEE Computer Society, March 2003.

[83] S. Josefsson. The Base16, Base32, and Base64 Data Encodings—RFC 4648, October 2006. Available at `http://www.ietf.org/rfc/rfc4648.txt`.

[84] F. Jouault and J. Bézivin. KM3: a DSL for Metamodel Specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, Bologna, Italy, 2006.

[85] T. Jussila, J. Dubrovin, T. Junttila, T. Latvala, and I. Porres. Model Checking Dynamic and Hierarchical UML State Machines. In *MODEVA 2006: Perspectives on Integrating MDA and V&V*, October 2006. Held in conjunction with MoDELS 2006.

[86] A. Kalnins, J. Barzdins, and E. Celms. Basics of Model Transformation Language MOLA. In *Workshop on Model Transformation and Execution in the Context of MDA (ECOOP 2004)*, June 2004.

[87] U. Keller, J. Wehren, and J. Niere. A Generic Difference Algorithm for UML Models. In *Software Engineering*, pages 105–116, 2005.

[88] S. Kelly. Comparison of Eclipse EMF/GEF and MetaEdit+ for DSM. In *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Workshop on Best Practices for Model Driven Software Development*, October 2004.

[89] S. Kent. Model Driven Engineering. In *Proc. of IFM International Formal Methods 2002*, volume 2335 of *LNCS*. Springer-Verlag, 2002.

[90] The Kermeta website. `http://www.kermeta.org/`, visited April 24th, 2007.

[91] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.

[92] A. Kleppe, April 2003. Discussion on the mailing-list puml-list@cs.york.ac.uk.

[93] D. S. Kolovos, R F. Paige, and F. Polack. Merging Models with the Epsilon Merging Language (EML). In *MoDELS*, pages 215–229, 2006.

[94] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In *GaMMa '06: Proceedings of the 2006 international workshop on Global integrated model management*, pages 13–20, New York, NY, USA, 2006. ACM Press.

[95] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. On-Demand Merging of Traceability Links with Models. In *Proceedings of the 2nd EC-MDA Workshop on Traceability*, 2006.

[96] J. Kovse and T. Härder. Generic XMI-Based UML Model Transformations. In *OOIS '02: Proceedings of the 8th International Conference on Object-Oriented. Information Systems*, pages 192–198, London, UK, 2002. Springer-Verlag.

[97] H. Kühn and M. Murzek. Interoperability Issues in Metamodelling Platforms. In *Proceedings of the First International Conference on Interoperability of Enterprise Software and Applications (INTEROP-ESA 2005)*, February 2005.

[98] L. Lambers. A New Version of GTXL: An Exchange Format for Graph Transformation Systems. In *Workshop on Graph-Based Tools (GraBaTs) 2004 at Second International Conference on Graph Transformation (ICGT 2004)*, October 2004.

[99] R. Lämmel and E. Meijer. Mappings make data processing go 'round—An inter-paradigmatic mapping tutorial. In *Post-proceedings of GTTSE 2005, Generative and Transformation Techniques in Software Engineering, 4–8 July, 2005, Braga, Portugal*, Lecture Notes in Computer Science. Springer-Verlag, 2006. Summer school tutorial, GTTSE 2005.

[100] R. Lämmel and E. Meijer. Revealing the X/O impedance mismatch, 2006. Draft, available at `http://homepages.cwi.nl/~ralf/xo-impedance-mismatch/`, visited October 23rd, 2006.

[101] J. R. Levine, T. Mason, and D. Brown. *lex & yacc*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, second edition, 1992.

[102] J. Lilius, T. Lillqvist, T. Lundkvist, I. Oliver, I. Porres, K. Sandström, G. Sveholm, and A. P. Zaka. An Architecture Exploration Environment for System on Chip Design. *Nordic Journal of Computing*, 12(4):361–378, 2005.

[103] J. Lilius, T. Lillqvist, T. Lundkvist, I. Oliver, I. Porres, K. Sandström, G. Sveholm, and A. P. Zaka. The MICAS Tool. In Kai Koskimies, Ludwik Kuzniarz, Jyrki Nummenmaa, and Zheying Zhang, editors, *Proceedings of the NWUML 2005: The 3rd Nordic Workshop on UML and Software Modeling*, number Report A-2005-3, pages 180–192, August 2005.

[104] T. Lillqvist. Subgraph Matching in Model Driven Engineering. Master's Thesis in Computer Science, Department of Information Technologies, Åbo Akademi University, Turku, Finland, March 2006.

[105] Y. Lin, J. Zhang, and J. Gray. Model Comparison: A Key Challenge for Transformation Testing and Version Control in Model Driven Software Development. In *OOPSLA Workshop on Best Practices for Model-Driven Software Development*, October 2004.

[106] Y. Lin, J. Zhang, and J. Gray. A Testing Framework for Model Transformations. In S. Beydeda, M. Book, and V. Gruhn, editors, *Model-Driven Software Development*, pages 219–236. Springer, July 2005.

[107] J. Lindqvist, T. Lundkvist, and I. Porres. A Query Language With the Star Operator. Technical Report 801, TUCS, January 2007.

[108] B. Liskov. Keynote address - Data Abstraction and Hierarchy. *SIGPLAN Not.*, 23(5):17–34, May 1988.

[109] B. Lundell, B. Lings, A. Persson, and A. Mattsson. UML model interchange in heterogeneous tool environments: an analysis of adoptions of XMI 2. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, volume 4199 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, October 2006.

[110] T. Lundkvist. Diagram Reconciliation and Interchange in a Modeling Tool. Master's Thesis in Computer Science, Department of Computer Science, Åbo Akademi University, Turku, Finland, November 2005.

[111] D. MacKenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files with GNU Diff and Patch.* Network Theory Ltd., 1997.

[112] J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM*, 3(4):184–195, 1960.

[113] E. Meijer and W. Schulte. Unifying Tables, Objects and Documents. In *Proceedings of Declarative Programming in the Context of OO Languages (DP-COOL)*, September 2003.

[114] E. Meijer, W. Schulte, and G. Bierman. Programming with Circles, Triangles, and Rectangles. In *Proceedings of the XML Conference and Exposition (XML 2003)*, December 2003.

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

[116] B. Meyer. Design by Contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.

[117] B. Meyer. *Eiffel : The Language*. Prentice Hall PTR, first edition, October 1991. ISBN 0132479257.

[118] M. Milewski and G. Roberts. The Model Weaving Description Language (MWDL)-Towards a formal Aspect Oriented Language for MDA model transformations. In *Proceedings of the 1st Workshop on Models and Aspects - Handling Crosscutting Concerns in MDSD, in conjunction with the 19th European Conference on Object-Oriented Programming*, 2005.

[119] M. Murata, S. St. Laurent, and D. Kohn. XML Media Types—RFC 3023, January 2001. Available at `http://www.ietf.org/rfc/rfc3023.txt`.

[120] E. W. Myers. An O(ND) Difference Algorithm and Its Variations. *Algorithmica*, 1(2):251–266, 1986.

[121] U. A. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 742–745. ACM Press, 2000.

[122] J. P. Nytun, A. Prinz, and A. Kunert. Representation of Levels and Instantiation in a Metamodelling Environment. In *Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language NWUML 2004*, pages 1–17, 2003.

[123] Object Management Group website. `http://www.omg.org/`, visited April 24th, 2007.

[124] D. Ohst, M. Welle, and U. Kelter. Difference Tools for Analysis and Design Documents. Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[125] D. Ohst, M. Welle, and U. Kelter. Differences Between Versions of UML Diagrams. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 227–236, New York, NY, USA, 2003. ACM Press.

[126] OMG. UML 2.0 Tools Certification (XMI). Available at `http://www.omg.org/xmitest/`. Press release at `http://www.omg.org/news/releases/pr2006/01-18-06.htm`. Visited December 11th, 2006.

[127] OMG. Object Constraint Language Specification, version 1.1, September 1997. Document ad/97-08-08, available at `http://www.omg.org/`.

[128] OMG. Meta Object Facility, version 1.4, April 2002. Document formal/2002-04-03, available at `http://www.omg.org/`.

[129] OMG. XML Metadata Interchange (XMI) Specification, version 1.2, January 2002. Available at `http://www.omg.org/`.

[130] OMG. MDA Guide Version 1.0.1, June 2003. OMG Document omg/03-06-01, available at `http://www.omg.org/`.

[131] OMG. UML 2.0 OCL Specification, October 2003. Document ptc/03-10-14, available at `http://www.omg.org/`.

[132] OMG. XML Metadata Interchange (XMI) Specification, version 2.0, May 2003. Document formal/03-05-02, available at `http://www.omg.org/`.

[133] OMG. MOF 2.0 Query / View / Transformation Final Adopted Specification, November 2005. OMG Document ptc/05-11-01, available at `http://www.omg.org/`.

[134] OMG. MOF 2.0/XMI Mapping Specification, v2.1, September 2005. Document formal/05-09-01, available at `http://www.omg.org/`.

[135] OMG. UML 2.0 Superstructure Specification, August 2005. Document formal/05-07-04, available at `http://www.omg.org/`.

[136] OMG. Unified Modeling Language: Diagram Interchange version 2.0, June 2005. OMG document ptc/05-06-04, available at `http://www.omg.org`.

[137] OMG. XML Metadata Interchange (XMI) Specification, version 2.1, September 2005. Document formal/05-09-01, available at `http://www.omg.org/`.

[138] OMG. Diagram Interchange version 1.0, April 2006. OMG document formal/06-04-04, available at `http://www.omg.org`.

[139] OMG. Meta Object Facility (MOF) Core Specification, version 2.0, January 2006. Document formal/06-01-01, available at `http://www.omg.org/`.

[140] OMG. Model View to Diagram Request for Proposal, November 2006. Document ad/06-11-07, available at `http://www.omg.org/`.

[141] OMG. Object Constraint Language, version 2.0, May 2006. Document formal/2006-05-10, available at `http://www.omg.org/`.

[142] OMG. UML 2.0 Infrastructure Specification, March 2006. Document formal/05-07-05, available at `http://www.omg.org/`.

[143] OMG Architecture Board. Model Driven Architecture—A Technical Perspective, 2001. Document ormsc/01-07-01, available at `http://www.omg.org/`.

[144] S. Owre, J. M. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[145] P. R. Panda. SystemC: a modeling platform supporting multiple design abstractions. In *ISSS '01: Proceedings of the 14th international symposium on Systems synthesis*, pages 75–80, New York, NY, USA, 2001. ACM Press.

[146] O. Patrascoiu. YATL:Yet Another Transformation Language. In *Proceedings of the 1st European MDA Workshop, MDA-IA*, pages 83–90. University of Twente, the Nederlands, January 2004.

[147] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel Changes in Large Scale Software Development: An Observational Case Study. In *Proceedings of the International Software Engineering Conference*, April 1998.

[148] A. Persson, H. Gustavsson, B. Lings, B. Lundell, A. Mattsson, and U. Ärlig. Adopting Open Source development tools in a commercial production environment—are we locked-in? In *Tenth International Workshop on Exploring Modeling Methods in Systems Analysis and Design (EMMSAD 2005)*, June 2005.

[149] A. Persson, H. Gustavsson, B. Lings, B. Lundell, A. Mattsson, and U. Ärlig. OSS tools in a heterogeneous environment for embedded systems modelling: an analysis of adoptions of XMI. In *Proceedings of the 5th Workshop on Open Source Software Engineering , held in conjunction with the 27th International Conference on Software Engineering (ICSE 2005)*, May 2005.

[150] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, June 2004. ISBN 0596004486.

[151] I. Poernomo. The Meta-Object Facility Typed. In Hisham M. Haddad et al., editors, *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 1845–1849, April 2006.

[152] I. Porres. A Toolkit for Model Manipulation. *Springer International Journal on Software and Systems Modeling*, 2(4), 2003.

[153] A. Prinz et al. The Semantic Metamodel-based Integrated Language Environment (SMILE) Project. Website at `http://osys.grm.hia.no/osys/projects/smile`, visited January 2nd, 2007.

[154] A. Prinz, J. P. Nytun, L. Chen, and S. Wei. Integration of MATER and EMF. In Andreas Prinz and Merete Skjelten Tveit, editors, *Proceedings of the 4th Nordic Workshop on the Unified Modeling Language NWUML 2006*, June 2006.

[155] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *LNCS*, pages 479–485. Springer-Verlag, 2004.

[156] V. Ribaud and P. Saliou. Roles Transformation within a Software Engineering Master by Immersion. September 2004.

[157] R. Rinat. Type-safe covariant specialization with generalized matching. *Inf. Comput.*, 177(1):90–120, 2002.

[158] M. Rose. On the Design of Application Protocols—RFC 3117, November 2001. Available at `http://www.ietf.org/rfc/rfc3117.txt`.

[159] D. Rosenthal and S. W. Marks. Inter-Client Communication Conventions Manual Version 2.0, December 1993. Available at `http://tronche.com/gui/x/icccm/`, visited April 24th, 2007.

[160] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[161] S. Demathieu and C. Griffin and S. Sendall. Model Transformation with the IBM Model Transformation Framework. Available at `http://www-128.ibm.com/developerworks/rational/library/05/503_sebas/`, visited April 24th, 2007.

[162] M. Scheidgen. On Implementing MOF 2.0—New Features for Modelling Language Abstractions. July 2005. Available at `http://www.informatik.hu-berlin.de/~scheidge/`, visited January 2nd, 2007.

[163] R. W. Scheifler and J. Gettys. *X Window System: Core and Extension Protocols*. Digital Equipment Corp., Acton, MA, USA, 1997.

[164] A. Schürr, A. J. Winter, and A. Zündorf. The PROGRES Approach: Language and Environment. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*, pages 487–550, 1999.

[165] P. Sriplakich, X. Blanc, and M.-P. Gervais. Supporting Collaborative Development in an Open MDA Environment. In *International Conference on Software Maintenance*, pages 244–253, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[166] J. Steel and J.-M. Jézéquel. Typing Relationships in MDA. In D. H. Akehurst, editor, *Proceedings of the Second European Workshop on Model Driven Architecture (EWMDA)*, number 17, Canterbury, Kent CT2 7NF, UK, September 2004. University of Kent.

[167] G. L. Steele, Jr. Growing a Language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[168] P. Stevens. Small-scale XMI programming: a revolution in UML tool use? *Automated Software Engineering*, 10(1):7–21, January 2003.

[169] H. Stoeckle, J. C. Grundy, and J. G. Hosking. Approaches to supporting software visual notation exchange. In *Human Centric Computing Languages and Environments*, pages 59–66. IEEE Computer Society, 2003.

[170] B. Stroustrup. *The C++ Programming Language*. Addison Wesley Longman, third edition, 1997. ISBN 0-201-88954-4.

[171] H. Sutter and J. Hyslop. Logically Shallow Views. *C/C++ Users Journal*, 23(5), May 2005.

[172] A. Sutton. Open Modeling Framework. Available at `http://www.sdml.info/projects/omf/`, visited January 2nd, 2007.

[173] SWIG Interface Generator from C/C++ to Scripting Languages. Available at `http://www.swig.org/`, visited April 24th, 2007.

[174] G. Taentzer. Towards Common Exchange Formats for Graphs and Graph Transformation Systems. *Electronic Notes in Theoretical Computer Science*, 44(4), 2001.

[175] ATLAS Team. Atlantic Metamodel Zoo. Available at `http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/`, visited April 24th, 2007., 2006.

[176] W. F. Tichy. RCS—A System for Version Control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[177] L. Tratt. The MT model transformation language. In *Proc. ACM Symposium on Applied Computing*, pages 1296–1303, April 2006.

[178] Unicode Consortium. *Unicode Standard, Version 5.0*. Addison-Wesley, 2006. ISBN 0-321-48091-0.

[179] UserLand Software, Inc. XML-RPC Specification, October 1999. Available at `http://www.xmlrpc.org/`, visited April 24th, 2007.

[180] G. van Rossum et al. The Python Programming Language. Available at `http://www.python.org/`, visited April 24th, 2007.

[181] D. Varró. Automatic Program Generation for and by Model Transformation Systems. In H.-J. Kreowski and P. Knirsch, editors, *Proc. AGT 2002: Workshop on Applied Graph Transformation*, pages 161–173, Grenoble, France, April 12–13 2002.

[182] D. Varró and A. Pataricza. Metamodeling Mathematics: A Precise and Visual Framework for Describing Semantics Domains of UML Models. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 18–33, London, UK, 2002. Springer-Verlag.

[183] D. Varró and A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, 2(3):187–210, October 2003.

[184] D. Varró, G. Varró, and A. Pataricza. Designing the Automatic Transformation of Visual Languages. *Science of Computer Programming*, 44(2):205–227, August 2002.

[185] W3C. Namespaces in XML, January 1999. Available at `http://www.w3.org/`.

[186] W3C. WAP Binary XML Content Format, June 1999. Available at `http://www.w3.org/TR/wbxml/`.

[187] W3C. XSL Transformations (XSLT) Version 2.0, November 1999. Available at `http://www.w3.org/TR/xslt20/`.

[188] W3C. Extensible Markup Language (XML) 1.0 (Second Edition), October 2000. Available at `http://www.w3.org/`.

[189] W3C. XML Linking Language (XLink) Version 1.0, June 2001. Available at `http://www.w3.org/TR/xlink/`.

[190] W3C. Scalable Vector Graphics (SVG) 1.1 Specification, January 2003. Available at `http://www.w3.org/TR/SVG11/`.

[191] W3C. The W3C Workshop on Binary Interchange of XML Information Item Sets, September 2003. Workshop in Santa Clara, California, USA. Report available at `http://www.w3.org/2003/08/binary-interchange-workshop/`.

[192] W3C. XPointer Framework, March 2003. Available at `http://www.w3.org/TR/xptr-framework/`.

[193] W3C. XML Schema Part 1: Structures Second Edition, October 2004. Available at `http://www.w3.org/TR/xmlschema-1/`.

[194] W3C. XML Schema Part 2: Datatypes Second Edition, October 2004. Available at `http://www.w3.org/TR/xmlschema-2/`.

[195] W3C. Binary Characterization—W3C Working Group Note, March 2005. Available at `http://www.w3.org/TR/xbc-characterization/`.

[196] W3C. XML-Binary Optimized Packaging (W3C Recommendation), January 2005. Available at `http://www.w3.org/TR/xop10/`.

[197] W3C. XML Path Language (XPath) Version 1.0, February 2005. Available at `http://www.w3.org/TR/xslt20/`.

[198] W3C. xml:id Version 1.0, September 2005. Available at `http://www.w3.org/TR/xml-id/`.

[199] W3C. XQuery 1.0: An XML Query Language (Candidate recommendation), June 2006. Available at `http://www.w3.org/TR/xquery/`.

[200] A. Wagner. A pragmatical approach to rule-based transformations within UML using XMI.difference. In *Proceedings of the Workshop on Integration and Transformation of UML models (WITUML 2002)*, July 2002.

[201] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, third edition, July 2000. ISBN 0596000278.

[202] Y. Wang, D. J. DeWitt, and J.-Y. Cai. X-Diff: An Effective Change Detection Algorithm for XML Documents. In *International Conference on Data Engineering (ICDE)*, pages 519–530, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[203] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998. ISBN 0201379406.

[204] B. Werther and D. Conway. A Modest Proposal: C++ Resyntaxed. *SIGPLAN Notices*, 31(11):74–82, 1996.

[205] E. D. Willink. UMLX: A graphical transformation language for MDA. In A. Rensink, editor, *CTIT Technical Report TR-CTIT-03-27*, pages 13–24, Enschede, The Netherlands, June 2003. University of Twente.

[206] A. Winter. Exchanging Graphs with GXL. Technical Report 9–2001, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2001.

[207] A. Winter et al. The Graph Exchange Language website. `http://www.gupro.de/GXL/`, visited April 24th, 2007.

[208] A. Winter, B. Kullbach, and V. Riediger. An Overview of the GXL Graph Exchange Language. In *Revised Lectures on Software Visualization, International Seminar*, pages 324–336, London, UK, 2002. Springer-Verlag.

[209] XIG—An XSLT-based XMI2GXL-Translator. Available at `http://www.gupro.de/mirror/xig/index.htm`, visited April 24th, 2007.

[210] Z. Xing and E. Stroulia. UMLDiff: An Algorithm for Object-Oriented Design Differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 54–65, New York, NY, USA, 2005. ACM Press.

[211] N. Zhu, J. Grundy, and J. Hosking. Pounamu: A Meta-Tool for Multi-View Visual Language Environment Construction. In *VLHCC '04: Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing (VLHCC'04)*, pages 254–256, Washington, DC, USA, 2004. IEEE Computer Society.

[212] A. Zündorf, J. P. Wadsack, and I. Rockel. Merging Graph-Like Object Structures. In *Proceedings of the Tenth International Workshop on Software Configuration Management*, 2001.

ISBN 978-952-12-1901-6
ISSN 1239-1883