CONCRETE SYNTAX DEFINITION FOR MODELING LANGUAGES

THÈSE N^{O} 3927 (2007)

PRÉSENTÉE LE 2 NOVEMBRE 2007

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

Institut des systèmes informatiques et multimédias

PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Frédéric FONDEMENT

ingénieur diplômé de l'école supérieure des sciences appliquées pour l'ingénieur de l'Université de Mulhouse, France
et de nationalité française

acceptée sur proposition du jury:

Prof. E. Telatar, président du jury
Dr T. Baar, directeur de thèse
Dr P.-A. Muller, rapporteur
Prof. B. Rumpe, rapporteur
Prof. A. Wegmann, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2007

Abstract

Model Driven Engineering (MDE) promotes the use of models as primary artefacts of a software development process, as an attempt to handle complexity through abstraction, e.g. to cope with the evolution of execution platforms. MDE follows a stepwise approach, by prescribing to develop abstract models further improved to integrate little by little details relative to the final deployment platforms. Thus, the application of an MDE process results in various models residing at various levels of abstraction.

Each one of these models is expressed in a modeling language, in which one may find appropriate concepts for the abstraction level considered. Many advocate to use the right (modeling) language for the right purpose. This means that it is sometimes better approach to use small languages specific to the considered domain and abstraction level, than to use general purpose languages (e.g. UML) when they do not perfectly fit the (modeling) needs. As a matter of fact, an MDE development process, which involves many different domains and abstraction levels, should also involve a large variety of modeling languages. Project managers who want to apply an MDE process need to deal with this language proliferation to such an extent that, in the long run, one may infer that language engineers can become major actors of software development teams.

We believe that comprehensive modeling language management facilities may considerably alleviate that MDE drawback. Such facilities may include modeling language definition, extension, adaptation, or composition. To define a (modeling) language, one need to define its abstract syntax, its semantics, and one or more concrete syntaxes. This thesis focuses on concrete syntax definition for modeling languages, when the abstract syntax is given in the form of a metamodel. We will provide solutions both for textual and graphical concrete syntaxes.

Some of our experiences in building textual languages (as MTL, a model transformation language), and graphical languages (as Netsilon, a web-application modeler) has shown that a lot of work was spent in implementing interface using traditional techniques, be it a text processor generated from a compiler compiler specification, or a modeler making use of modern 2D graphical libraries. Indeed, abstract and concrete syntax were implemented in a disconnected way, and it was then necessary to assemble them, which became rapidly clumsy while abstract syntax evolved.

We built our solution to concrete syntax definition as companions of the abstract syntax. The definition of concrete syntax we propose here made it possible to build automatic tools able to analyze or synthesize models from/to text, and to create graphical modelers. We will present a metamodel for textual concrete syntax definition to construct constructive reversible grammars. We will also propose a technique for graphical concrete syntax definition following a two-step process: specification and realization. Specification is a restrictive approach in which a metamodel defines a graphical concrete syntax. Both relations with

abstract syntax and spatial relationships are expressed by means of constraints. The realization step proposes a way to provide the concrete syntax tree a meaning, by attributing it a graphical appearance, and by expressing possible user interactions.

The structure of the document is the following. After introducing in deeper details the problem and the general structure of the solution we propose, we will take a tour of MDE, text and graph grammars. Then, we will present Netsilon as an example of an MDE tool to MDE development, which required both the definition of a graphical and a textual modeling language. The two following sections will present the solutions we propose for textual and graphical concrete syntax definition, respectively. Final remarks and possible improvements, especially regarding reusability in general of MDE meta-artifacts (like metamodels or model transformations), and of concrete syntax in particular, will conclude the document.

Keywords

Model Driven Engineering, Metamodeling, Language Engineering, Concrete Syntax, Textual Concrete Syntax, Graphical Concrete Syntax, Scalable Vector Graphics.

Version Abrégée

L'ingénierie dirigée par les modèles (IDM) promeut les modèles comme composants principaux d'un processus de développement logiciel. Cette approche tente d'organiser la complexité par l'abstraction, par exemple afin de faire face à l'évolution des plate-formes d'exécution. L'IDM suit une approche par étapes dans laquelle sont développés des modèles abstraits, peu à peu améliorés pour y intégrer les détails dont a besoin la plate-forme de déploiement finale. Ainsi l'application d'un processus IDM produit une multitude de modèles à différent niveaux d'abstraction.

Chacun de ces modèles est exprimé dans un langage supposé apporter les concepts adéquats pour le niveau d'abstraction considéré. De nombreux auteurs préconisent d'utiliser le langage (de modélisation) le plus adapté possible au but poursuivi. En d'autre termes, il est souvent plus ingénieux d'utiliser de petits langages très spécialisés pour un domaine et un niveau d'abstraction donné, que d'utiliser des langages génériques (tels UML) quand ils ne satisfont pas pleinement au besoins de la modélisation. C'est pourquoi un processus de développement IDM, qui implique de nombreux domaines et niveaux d'abstraction, doit souvent impliquer de nombreux langages. Les chefs de projets désireux d'adopter une approche IDM se voient donc dans l'obligation de jongler avec une véritable prolifération de langages, ceci à tel point qu'on est en droit de se demander si les ingénieurs du langage ne seront pas les partenaires indispensables des équipes de développement logiciel du futur.

L'idée développée dans ce document est qu'un mécanisme générique de gestion de ces nombreux langages simplifierait sans doute cette situation spécifique à l'IDM. De tels mécanismes seraient par exemple capables de définir complètement des langages de modélisation, de les étendre ou les adapter, voire de les composer. Pour définir un langage, qu'il s'agisse d'un langage de modélisation ou non, il est nécessaire de spécifier sa syntaxe abstraite, au moins une syntaxe concrète et sa sémantique. Cette thèse s'intéresse à la définition de syntaxes concrètes, une fois la syntaxe abstraite établie sous forme de métamodèle. Ces syntaxes concrètes peuvent être soit textuelles, soit graphiques.

Certaines de nos expériences dans la construction de langages textuels (comme MTL, un langage de transformation de modèles) ou graphiques (tel Netsilon, un modeleur d'application internet) ont montré que beaucoup d'énergie est dépensée dans l'implémentation des interfaces utilisateur, qu'il s'agisse de processeurs de texte générés à partir de spécifications pour compilateur de compilateurs, ou de modeleurs créés à partir de bibliothèques pour la représentation graphique en deux dimensions. En effet, dans ces projets, les syntaxes abstraites et concrètes sont développées de manière indépendante, ce qui pose rapidement des problèmes en terme de cohérence lors des évolutions de la syntaxe abstraite.

Les solutions proposées ici définissent les syntaxes concrètes comme dépendantes des syntaxes abstraites. Il a été possible de construire des implémentations prototypes basées sur les approches proposées. L'une d'elle est capable de produire du texte à partir d'un

modèle, ou inversement un modèle à partir d'un texte. Une autre est capable de fournir un modeleur graphique maintenant en cohérence un modèle et sa représentation graphique. La première contribution de cette thèse est la définition d'un métamodèle pour la spécification constructive de syntaxes textuelles réversibles. La seconde contribution est une méthode de spécification de syntaxes graphiques en deux étapes: l'étape de spécification et l'étape de réalisation. La spécification suit une approche restrictive dans laquelle un métamodèle définit la structure d'un graphe de syntaxe concrète. Des contraintes viennent en complément afin d'exprimer la cohérence entre les syntaxes concrète et abstraite, ainsi que les règles de composition spatiale. L'étape de réalisation consiste à préciser le mode de représentation du graphe de syntaxe concrète, ainsi que des interactions possibles avec un utilisateur souhaitant modéliser un système.

Après une brève introduction, ce document commencera par un aperçu de l'IDM ainsi que des techniques communément acceptées de spécification de syntaxes concrètes textuelles et graphiques. Nous verrons ensuite l'outil Netsilon, qui est un exemple d'outil de modélisation pour l'IDM: nous pourrons alors avoir un exemple de démarche IDM et de spécification de langages textuels et graphiques. Nous en déduirons de possibles améliorations dans la construction d'outils pour l'IDM basés sur les langages. Les deux chapitres suivants décriront les approches proposées pour la spécification de syntaxes textuelles et graphiques. Enfin, nous conclurons sur les améliorations à apporter aux approches proposées, notamment en ce qui concerne la réutilisabilité des artefacts des méthodes de type IDM en général (par exemple les métamodèles ou les transformations de modèles), et des syntaxes concrètes en particulier.

Mots Clefs

Ingénierie Dirigée par les Modèles, Métamodélisation, Ingénierie des Langages, Syntaxes Concrètes, Syntaxes Textuelles, Syntaxes Graphiques, Scalable Vector Graphics.

Acknowledgements

First of all, I need to apologize to the reader of this document for that I used the English language that I poorly master. So best thanks to all those readers who will manage to ignore all these «implementation» defects that appear all along the document.

Obviously, a thesis is never the matter of a single person. It is especially true regarding this one. So, though I appear as the "main" author of this thesis, I want to acknowledge the following people.

First, this thesis was possible to start. Prof. Alfred Strohmeier accepted my application for this doctoral assistant position that I found thanks to Google using a search criteria like `uml ocl "offre d'emploi"`. He accepted me regarding my work on the MTL language, which was possible thanks to Prof. Jean-Marc Jézéquel, who in turn accepted me regarding my work on Netsilon, which was possible thanks to Dr. Pierre-Alain Muller.

This thesis was also possible to continue when Prof. Strohmeier acceded to his new responsibilities of rector at the university of Neuchâtel. I am very grateful to Dr. Thomas Baar for receiving me as a doctoral student. I also want to thank him for freedom he let me in my work.

As an apprentice researcher, I needed advisors from which I could learn and who could introduce me to the scientific community. If Thomas helped me a lot, I also want to acknowledge Dr. Pierre-Alain Muller and Dr. Raul Silaghi.

I also would like to thank all those people with whom a valuable scientific collaboration was possible, but I'm afraid I will not be able to mention them all here. Nevertheless, I would like to cite Dr. Raul Silaghi, Dr. Pierre-Alain Muller, Dr. Thomas Baar, and Slavisa Markovic. Some other people helped me a lot in engineering and implementing ideas presented here: Prof. Michel Hassenforder, Rémi Schnekenburger, Fabrice Hong, Fabien Rohrer, François Helg, and Amit Raj.

This thesis was also financially supported: thanks to the Hasler foundation, INTERREG (i.e. European Union and Switzerland), EPFL, PUBLICA, and CFC. Thanks also to the ENSISA, which provided me with an office where I could write this document.

Finally, this thesis could be defended, so thanks a lot to the reviewers who accepted to dive into the details of the present document: Dr. Thomas Baar, Prof. Bernhard Rumpe, Prof. Alain Wegmann, and Prof. Emre Telatar. I am also very grateful to Dr. Alban Rasse and Slavisa Markovic for reading over that document and who helped in improving it. Again, special thanks to Dr. Pierre-Alain Muller for providing abundant advice all along (and before) the thesis, including regarding the dissertation.

But, most important, my deepest gratitude goes to my mother and Stéphanie. Their support in those dark times that occurred during this thesis period was the most precious ally one could dream; their literal devotion saved me from troubles I do not dare to imagine. Without them, that document would not have existed. In these times, I also received a lot of support from Dr. Anne Stucki, and Aurélia. Again, thanks to Dr. Pierre-Alain Muller for indefatigably motivating me to write that document.

Table of Contents

Abstract i

Version Abrégée iii

Acknowledgements v

Table of Contents vii

List of Figures xi

1 Introduction 1

1.1 Promoting Abstraction through Languages 1
1.2 Motivations 4
1.3 Contributions 8
1.4 Plan 8

2 Language Driven Engineering 11

2.1 Language Driven Engineering (LDE) 11
2.1.1 A Method for Methodology 11
2.1.2 Published Techniques 14
2.1.2.1 Model Integrated Computing 15
2.1.2.2 Model Driven Architecture 15
2.1.2.3 Software Factories 16
2.1.2.4 Domain Specific Languages (DSL) 16
2.1.2.5 UML 17
2.1.3 Key Technologies 18
2.1.3.1 Metamodeling 18
2.1.3.2 Model Transformation 22
2.1.3.3 Code Generators 23
2.1.3.4 Graphical Concrete Syntax 24
2.2 Traditional Concrete Syntax Engineering 25
2.2.1 Traditional Textual Language Engineering 26
2.2.2 Graphical Language Engineering 27
2.2.2.1 Graph Grammars 29
2.2.3 Scalable Vector Graphics 29
2.3 Conclusions 30

3 Netsilon: LDE Example and Embryonic Solution 33

3.1 Web Applications 33
3.2 LDE for Web Application Engineering 34

3.2.1 Business Model ... 35
3.2.2 Hypertext Model ... 36
3.2.3 Presentation Model ... 37
3.2.4 Xion ... 39
3.3 Implementation Insights ... 42
3.3.1 Metamodel ... 42
3.3.2 Improvements and Code Generation ... 44
3.4 Experiencing SVG Model Representation ... 46
3.4.1 Simplified UML Class Diagrams ... 47
3.4.2 Representation Using Netsilon ... 48
3.4.2.1 Representation Framework ... 48
3.4.2.2 Business Model: Concrete Representations ... 50
3.4.2.3 Hypertext and Presentation: Concrete Representations ... 54
3.4.2.4 Generating Representation ... 57
3.5 Conclusion ... 59

4 Textual Concrete Syntax ... 63

4.1 Introduction ... 63
4.2 Motivations ... 64
4.2.1 Abstract Syntax versus Concrete Syntax ... 64
4.2.2 Model-Driven Compilers ... 65
4.2.3 Related Works ... 66
4.2.4 Requirement for Bidirectional Mapping ... 67
4.2.5 Towards a Specification ... 68
4.2.5.1 Kinds of Data ... 69
4.2.5.2 Multiplicity of the Data ... 70
4.2.5.3 Shared and parameterized feature-mappers ... 70
4.3 Modeling Concrete Syntax ... 70
4.3.1 Overview of our Proposal ... 71
4.3.1.1 `Template` Rule ... 73
4.3.1.2 `Terminal` Rule ... 73
4.3.1.3 `Sequence` Rule ... 73
4.3.1.4 `Iteration` rule ... 73
4.3.1.5 `Alternative` rule ... 74
4.3.1.6 `PrimitiveValue` rule ... 74
4.3.1.7 `ObjectReference` rule ... 75
4.3.1.8 `RuleRef` rule ... 75
4.3.1.9 `Action` side effect ... 75
4.4 Examples ... 75
4.4.1 A very simple example of concrete syntax specification ... 76

4.4.2 Statechart Textual Concrete Syntax Example 78
4.5 Prototype Implementations 81
4.6 Conclusion 82

5 Graphical Concrete Syntax 83
5.1 Introduction 83
5.2 Language examples 86
5.2.1 Statecharts 86
5.2.2 Chessboard 87
5.3 Concrete Syntax Specification 89
5.3.1 Scheme-based Definition of Concrete Syntax 90
5.3.2 Statechart Concrete Syntax 91
5.3.3 Icon-definition within a Scheme 93
5.3.4 Constraint definition within a Scheme 96
5.3.5 Chessboard Concrete Syntax 97
5.4 Display Classes Implementation 101
5.4.1 A Template-Based Approach 102
5.4.2 Predefined DOM Components to Specify User Interactions 104
5.4.2.1 Architecture 104
5.4.2.2 Predefined DOM Interfaces 106
5.4.3 Relation With the Model 110
5.4.3.1 Access to Repository 111
5.4.3.2 Maintaining High-Level Variables 112
5.4.3.3 Reactions to Events 113
5.4.3.4 Reacting to Model Changes 116
5.4.4 Prototype Implementation Overview 117
5.5 Comparison with Other Approaches 121
5.6 Conclusion 123

6 Conclusion 127
6.1 Summary 127
6.2 «A language that is used will be changed» [Leh80] 129

Appendix A: Applying and Customizing LDE 131
A.1 Technologies 131
A.1.1 UML Profiles 131
A.1.2 MTL 132
A.2 Integrating Distribution Concern in an UML Model 134
A.2.1 From Object-Oriented Designs to Distributed Systems 135
A.2.2 Enterprise Fondue and the Distribution Concern 136
A.2.3 UML Profiles to Address the Distribution Concern 137

A.2.3.1 UML Distribution Profile 138
A.2.3.2 UML Abstract Distribution Realization Profile 141
A.2.3.3 UML CORBA Distribution Realization Profile 142
A.2.4 MTL Model Transformations for Applying the UML-D Profiles .. 143
A.2.4.1 Refining Along the Distribution Concern-Dimension 143
A.2.4.2 Refining Distribution Along the CORBA Technology-Dimension 145
A.2.5 Conclusions 148
A.3 AspectMTL: Customizing Improvements 150
A.3.1 Motivations 151
A.3.2 The MTL Weaver 153
A.3.3 MTL-Based Syntax for Describing the Weaving Behavior 155
A.3.4 Running Example 159
A.4 Conclusion 163

7 Bibliography 165

Curriculum Vitae 179

List of Figures

Chapter 1: Introduction

Figure 1.1: Concrete Syntax for Metamodel-Based Languages 5
Figure 1.2: Concrete Syntax for Metamodel-Based Languages Revisited 7

Chapter 2: Language Driven Engineering

Figure 2.1: LDE Concept of Successive Improvements 12
Figure 2.2: The V Model 13
Figure 2.3: M1: A Relational Database Language Sentence 18
Figure 2.4: M2: Relational Database Language Abstract Syntax 19
Figure 2.5: M1 conforming to M2: Figure 2.3 as an ASG (Excerpt) 19
Figure 2.6: M2 conforming to M3: Figure 2.4 as a MOF ASG (Excerpt)......... 20
Figure 2.7: M3: MOF 1.4 (© OMG)... 21

Chapter 3: Netsilon: LDE Example and Embryonic Solution

Figure 3.1: Web Application Modeling with Netsilon 34
Figure 3.2: Netsilon Set of Models 35
Figure 3.3: An Example of Business Model.. 36
Figure 3.4: Hypertext Model for Listing People.. 38
Figure 3.5: Genealogy Web Application at Work 39
Figure 3.6: Hypertext Model for `personInList` 39
Figure 3.7: Implementing `Person::marry` in Xion 41
Figure 3.8: Sisters found by a Xion Expression 42
Figure 3.9: Hypertext Metamodel (Excerpt).. 43
Figure 3.10: Deployment Metamodel (Excerpt).. 44
Figure 3.11: Code Generation Process 45
Figure 3.12: Simplified UML Class Diagrams Metamodel................................. 47
Figure 3.13: Diagramming Framework 49
Figure 3.14: `Attribute` Representation Element 51
Figure 3.15: `Classifier` Representation Element 52
Figure 3.16: `Generalization` Representation Element 53
Figure 3.17: `getSpecializationConnector` Xion Implementation 54
Figure 3.18: `Attribute` Representation Presentation Model 54
Figure 3.19: `Classifier` Representation Hypertext Model (Excerpt) 55
Figure 3.20: `Classifier` Main Representation Presentation Model 55
Figure 3.21: `Generalization` Representation Presentation Model 56
Figure 3.22: An Example Model for Figure 3.3 57
Figure 3.23: An Example Representation Model for Figure 3.22 58

Figure 3.24: Graphical Representation for Figure 3.23 59
Figure 3.25: The Netsilon LDE Process 60

Chapter 4: Textual Concrete Syntax

Figure 4.1: Model-Based Compiler Architecture 66
Figure 4.2: Abstract Syntax of a Simple Language 68
Figure 4.3: Example of Model and a Expected Textual Representation 69
Figure 4.4: Automatic Bidirectional Model Representation 71
Figure 4.5: Overview of the Metamodel for Textual Concrete Syntax 72
Figure 4.7: Variation with Top-Level Reusable Templates 76
Figure 4.6: Straightforward Textual Concrete Syntax Model 77
Figure 4.8: The Simplified Statechart Metamodel 78
Figure 4.9: A Statechart Sentence and a Textual Representation 79
Figure 4.10: Textual Concrete Syntax Specification for the Statechart Language 80

Chapter 5: Graphical Concrete Syntax

Figure 5.1: General Architecture 85
Figure 5.2: Symbols for the Statechart Concepts 86
Figure 5.3: A Statechart Sentence and its Graphical Representation 86
Figure 5.4: The Chessboard Metamodel 87
Figure 5.5: An Incorrect Chessboard Model 88
Figure 5.6: A Chessboard Sentence 89
Figure 5.7: Scheme Definition Architecture 91
Figure 5.8: Statechart Schemes 92
Figure 5.9: The Composite State and Transition Icons 94
Figure 5.10: The Chessboard Schemes 98
Figure 5.11: The Board and Square Icons 99
Figure 5.12: SVG Templates for Statecharts 103
Figure 5.13: `SimpleState` SVG Template:
CSVG Constraint to Handle Text Growth 104
Figure 5.14: Conceptual Architecture of DOM Interfaces 106
Figure 5.15: `SimpleState` SVG Template: Declaring DOM Components 109
Figure 5.16: `Transition` SVG Template: Declaring DOM Components 109
Figure 5.17: JMI Initialization Script 112
Figure 5.18: `SimpleState` SVG Template: Creation Reaction 112
Figure 5.19: `SimpleState` SVG Template: `onStick` Reaction 115
Figure 5.20: `SimpleState` SVG Template: Updater 117
Figure 5.21: Initial View of the Prototye Implementation 119
Figure 5.22: Simple State Template Instantiated 119
Figure 5.23: SVG File for a Single Simple State Instance 120
Figure 5.24: XMI File for a Single Simple State Instance 121

Figure 5.25: The Door State Machine Described in the Tool 122

Chapter 6: Conclusion

Appendix A: Applying and Customizing LDE

Figure A.1: The MTL Compilation Process 133
Figure A.2: Refinement Steps to Integrate a Distribution Concern with UML.. 134
Figure A.3: The Bank Example 135
Figure A.4: Refinement Process in Enterprise Fondue 138
Figure A.5: MDA-Oriented Hierarchy of UML-D Profiles 139
Figure A.6: The `MTL1-D` Transformation: Exploring Operations Part 144
Figure A.7: The `MTL1-D` Outcome for the Bank Example 145
Figure A.8: The `MTL2-D` Transformation: Creating Exposition Part 147
Figure A.9: The `MTL2-D` Outcome for the Bank Example 148
Figure A.10: Customizing an LDE Process 149
Figure A.11: Refining along the Distribution, RMI-Technology, and Java-Language Concern-Dimensions 153
Figure A.12: The MTL Weaving Process 154
Figure A.13: Snippets of the `MTL1-D-Aspect` 156
Figure A.14: MTL Weaver Snippets for Class Merge (`mergeClass`) 158
Figure A.15: Predefined MTL-Aspect Tags 159
Figure A.16: Snippets of the `Copy` Input Library 160
Figure A.17: Snippets of the `Distribution` Output Library 162

Chapter 1:
Introduction

1.1 Promoting Abstraction through Languages

The continuous evolution of hardware towards more and more execution speed and memory capacity paved the way to more and more complex software. Software engineering was born almost 40 years ago, when it became clear that software development could not be conducted anymore without engineering guidance and specific techniques, such as software development *methods*, execution *platforms* and *abstraction* mechanisms.

Methods intend to state clearly the steps to be taken to develop a software system. They identify the various stakeholders (e.g. business analyst, software architect, software developer) and their respective roles and duties. Methods help organizing tasks of team members for both producing and maintaining software systems. Methods range from very precise techniques designed for developing well-defined systems with an emphasis on reliability (e.g. B [Abr96]), to collections of informal best practices intended for rapid application development of systems with highly evolving requirements (e.g. eXtreme Programming [BA04]).

Platforms are layered collections of reusable software or hardware components intended to support further layers of software. They offer well defined services that can be accessed by connected software through well-agreed upon collections of interfaces. Platforms can be of very different natures: examples for kinds of platforms are processors, which offer sets of instructions, operating systems, which offer a common mean to access to categories of hardware (including processors) and organize process flows (e.g. POSIX), virtual machines, which offer a common behavior on different operating systems (e.g. JRE as implemented for various operating systems), and object request brokers, which facilitate the collaboration of distributed and heterogeneous software components (e.g. CORBA [ABB+04]).

Abstraction, in a general sense, is a mental selection mechanism intended to discard information that is not relevant at a given point in time. As such, abstraction is a cornerstone of software engineering techniques and methods. As a matter of fact, abstraction allows engineering systems by concentrating first on core business functionalities while deferring secondary concerns like details of final execution platform. The details discarded earlier are introduced later on in the system by "lowering the level of abstraction", either by evolution or by refinement. Platforms are examples for supporting abstraction in software engineering, in that offered interfaces actually hide implementation details to the client software system. Introducing abstraction in software system engineering can be achieved either

explicitly via interfaces of platforms, or *implicitly* by embedding specific concepts in languages.

In the *explicit* case, interfaces are collected in libraries that can be used in program code. An evolution of explicit use of platform is Component Oriented Programming [Szy02], which fragments computer systems into a framework of different software components interrelated through their interfaces. Platforms are here considered as already developed ("off-the-shelf") components, following the example of the CORBA platform which offers a predefined set of interfaces (e.g. in [BDII02] and in [ABH+00]). However, the explicit use of platform becomes tedious as soon as platforms become more complex [CBR03]. For instance, «*popular middleware platforms, such as J2EE, .NET, and CORBA, contain thousands of classes and methods with many intricate dependencies and subtle side effects that require considerable effort to program and tune properly. Moreover, since these platforms often evolve rapidly —and new platforms appear regularly— developers expend considerable effort manually porting application code to different platforms or newer versions of the same platform*» [Sch06].

In the *implicit* case, services offered by platforms are conceptualized and made available via constructs of a specific *artificial* language (as the opposite of *natural* language), be it a modeling language or a programming language. In the case of executable languages, language statements (i.e. language *sentences*) are either interpreted by a system that refers to the platform, or compiled into another language that makes less assumptions about the peculiarities of the platform. In both cases, additional information about the platform is required.

The hierarchy of programming languages is a good example of abstraction through languages. Basically, hardware processors interpret instructions written in a language with only two lexemes: true and false. For computer systems to be easier to program by humans, patterns of binary instructions were abstracted in assembly languages (one per kind of processor), which define concepts such as register, instruction, or datum. Assembly code can be compiled back into a stream of true/false lexemes so that the processor (platform) can execute the specification. The pattern applies again for 3GL languages (e.g. Fortran, C) that are compiled into assembly languages, and that embed concepts such as function or structure. An interesting result is code portability across platforms: a C specification may be compiled for various kinds of processors that offer similar behavior yet offering different instruction sets. Note that a 3GL language may be defined on top of another 3GL language (e.g. C++ on top of C). 4GL languages apply the pattern once more and leave the programmatic technological space to offer concepts closer to the problem domain. The idea was further emphasized in Domain Specific Languages (DSLs) [KMB+96].

Another such illustration of practical use of abstraction is modeling. A model is a simplification of a system valid in certain circumstances, and that is easier to reason on, i.e. easier to engineer, manipulate, communicate, etc. Again, a model is expressed in a given (modeling) language that captures abstraction through specific concepts. Model Driven

Engineering (MDE - [Ken02]) and related techniques ([GSCK04], [MCF03], etc.) promote the systematic use of models in the software lifecycle. A model resides at a given level of abstraction regarding platform dependency. Platforms for computer systems (web application servers, database servers, ORB frameworks, operating systems, etc.) appear and evolve very rapidly. While a given category of platforms embrace a given number of common concepts, each platform has its specificities that computer systems need to deal with. By the systematic use of models, computer system developers can first concentrate on an abstract model to describe the core business. Further activities *concretize* the model (as the opposite of abstract) either by improving the model or by building new models to integrate details about how the final system may integrate the platform(s). An important advantage of MDE-like technique is that they provide means to *automate* the concretization mechanism, following the example of compilers for 3GL languages. Note that the decision of the final platform is not taken yet at the time the abstract model is developed, and that the abstract model may be reused to target different platforms. As we will see in section 2.1.1 on page 11, one may even see MDE-like technologies as a mean to formalize at least part of a method, by stating, using formal specifications, the flow of models to develop and improvements to bring them. Moreover, especially if one introduces the notion of abstract platform [ADvSP04], one may develop a system specification taking the form of a model that can serve to target different platforms. As a summary, this promising way makes it possible to develop software systems taking advantage of *abstraction* as offered by modeling languages, handle introduction of *platforms*, and even provide support for *methodology*.

MDE-like techniques propose abstraction through modeling languages. To do so, two philosophies are possible. General purpose modeling languages, like UML [AAB+07], offer a common mean to model systems of very different natures. Advantage is that those (usually broad) languages can be used to model different systems once those languages are mastered by actors of development project. Moreover models are easier to understand when it comes to maintenance. General purpose languages are often well supported by tools. A problem is that the concepts they propose fit only partially the demands of the system domain(s). The second philosophy, as proposed in [CESW05] or in [KSLB03], is to apply the experience gained in the domain of specific languages to modeling, by defining specific modeling languages dedicated to the domain of the system under study. Developing system using such domain specific languages is called Domain Specific Modeling (DSM, e.g. in [Poh03]). In the software development lifecycle, which implies various models of various domains at various levels of abstraction, making use of a DSM approach implies the use of a large number of DSLs. The problem is that the target audience of a domain specific modeling language is rather small, making it hardly affordable to develop dedicated tools to support DSLs. Examples for such tools are modeling environments (graphical Computer-Aided Software Development tools, CASE tools in short, or textual Integrated Development Environments - IDEs in short), interpreters or transformers to executable languages.

To apply a DSM development process, a project architect needs to define all those DSLs that have to be used, how they relate to each other (e.g. transformation between two DSLs, or usage of a DSL within another), and under which circumstances they are applied. While MDE is designed to manage complexity of software and platforms, it tends however to make more complex the task of project management. The problem is even more complex if some of these languages need to be customized.

In the past, such proliferation of languages, with lack of interconnected tool support, put a break on the development of DSL usage [Iiv96]. The novelty brought by MDE-like technologies is to ease development of such languages and tools. Two key technologies are at the core of the MDE-like technologies: metamodeling and model transformation. Metamodeling makes it possible to define the abstract syntax of languages. A metamodel is the placeholder for modeling *static* concepts of a language (i.e. its vocabulary) and theirs relations (i.e. the taxonomy). A sentence of a language is embodied by a model that conforms to the metamodel of this language, i.e. structure of the model graph follows directives of the metamodel. By its graph nature, one may call the model the abstract syntax graph. Model transformations may express *behavior* of the language by stating how models may be manipulated. Model transformations, for their definition, rely on the metamodel of the transformed models. Model transformations may be used for various purposes: transforming a model of a given language into a model of another language, adding information into a model, mixing models of a same language, making a model evolve according to an event, etc.

A language is defined by its *abstract syntax*, its *semantics*, and as much *concrete syntaxes* as necessary. In the context of MDE-like technologies, metamodeling addresses that issue of defining the abstract syntax. Different solutions to capture semantics have been proposed (e.g. using model transformations [KW03, MB06]), but semantics specification is still subject to discussions [HR04]. The problem of concrete syntax has recently gained interest in the MDE community.

In this thesis, we concentrate on the definition of concrete syntaxes for languages whose abstract syntax is already provided as a metamodel. We investigate specification techniques (for both *textual* and *graphical* languages) that must be precise enough for an automatic tool to provide a modeling environment (CASE or IDE). Note that specifying concrete syntax is not a new problem and has found some solutions in non metamodel-related communities. For instance, text structures may be formally defined using generative grammars [Cho56] and graphics by graph grammars [Jon90].

1.2 Motivations

Figure 1.1 provides an overview of a typical usage and definition of concrete syntaxes for metamodel-based languages. The abstract syntax is modeled by a metamodel. A language

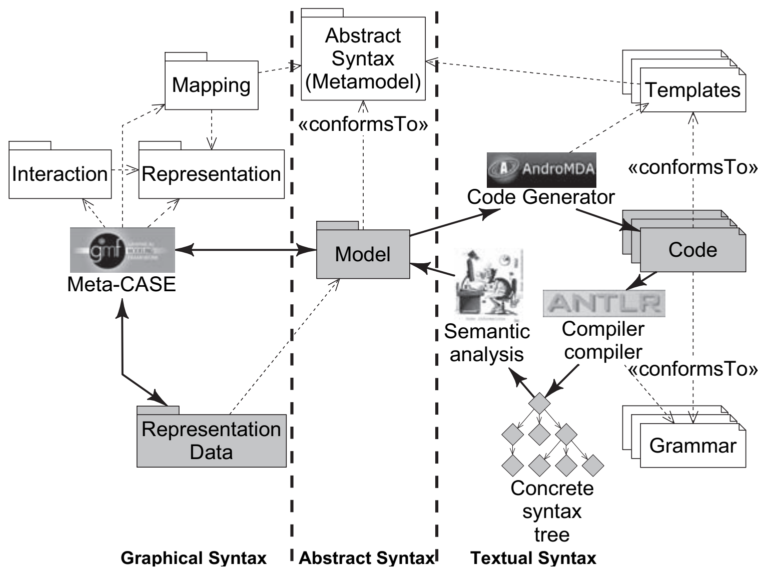


Figure 1.1: Concrete Syntax for Metamodel-Based Languages

sentence is a model, which is an instance of the concepts defined in the metamodel, as depicted by the «`conformsTo`» relationship. Dashed arrow represent dependencies while bold arrows represent data flow. Grayed items represent artifacts that are at the modeling level (e.g. a language sentence - M1 in the MDA terminology - see section 2.1.3.1 on page 18), and white items are data necessary to define a language (i.e. the specification for a language - M2 in the MDA terminology). The figure shows possible solutions to manage graphical concrete syntax (at the left) and textual concrete syntax (at the right).

At the left of model, in the figure, appears specification for *graphical concrete syntax* as promoted recently by meta-case tools such as the GMF tool [Ecl06]. A `Representation` model is defines the icons to be used. Such model is specified in a language which may declare concepts as circle, rectangle, or path, and which must have clear semantics regarding representation. `Mapping` is a model that maps a concept in the language to an icon described in the representation model. `Interaction` is another model which states how one may impact the model by interacting with the representation. As an example, this latter model may describe what a move is and how it impacts the representation. Note that this architecture is not adopted by all meta-case tools: some may ignore abstract syntax and

thus do not provide mapping; some others only provide default interactions and thus do not need any interaction model. Meta-case tools, according to those models, be it by interpretation or compilation, offer a "derived" CASE tool specialized in manipulating models of such defined graphical languages. Graphical representation for a model often requires information that may not be found in the model. As an example, if the location of an icon on a graphical diagram is left free, information about the actual location appears in the `Representation Data` model (following the example of Diagram Interchange [ADG+06]). Note that this approach to graphical concrete syntax definition for metamodel-based languages is relatively new. Before, one typically needed to design graphical languages in an ad-hoc way, using general purpose two-dimensional graphical libraries for programming languages (e.g. GEF [Ecl]).

At the right of the model, in the figure, appears a specification for *textual concrete syntax*. For historical reasons, this specification is separated in two different parts depending whether one consider producing textual representation from a model, or producing a model from textual representation. To produce textual representation, a code generator often instantiates text templates while visiting the complete model to produce text, following the example of the AndroMDA tool [Boh07]. Each text template states how a given concept should be rendered. When it comes to analyzing some text to produce a model, the usual way is to benefit from compiler technologies. A compiler compiler (e.g. the ANTLR tool [Par05]) creates a text processor from a structured text grammar specified with a hierarchy of rules. The result is a concrete syntax tree as a trace of triggered rules to recognize text. Further hand-coded specialized application needs to visit that concrete syntax tree to promote it into a model. One of the drawback of that architecture is that both ways (from model to text and from text to model) have to be coherent. Moreover, compiler technologies are not well suited for concrete syntax representation. Code generation technologies emerged when it came to produce code from models, regardless whether target code is actually a representation for another language. For instance, one could imagine, as it is the case in AndroMDA, to generate Java EJB code out of UML models.

We propose alternatives to both graphical and textual part of this architecture, as shown in figure 1.2.

Instead of asking language engineers to learn yet a new language dedicated to representation, we propose to let the *graphical concrete syntax* be defined using a metamodel of the same nature as the one that captures abstract syntax. Advantage is that the concrete syntax may be specified in a way that abstracts away representation technology. Concrete syntax is thus easier to understand when one needs to reuse it by rewriting the mapping to abstract syntax, or for maintenance purpose. Moreover, the task of developing abstract syntax and concrete syntax may be tackled by different language engineers. Using a metamodel to specify concrete syntax also allows better reuse and adaptation of concrete syntaxes by merely using metamodeling techniques (such as package merge or model transformation). Representation data are organized in a model with a metamodel with no need of a particular

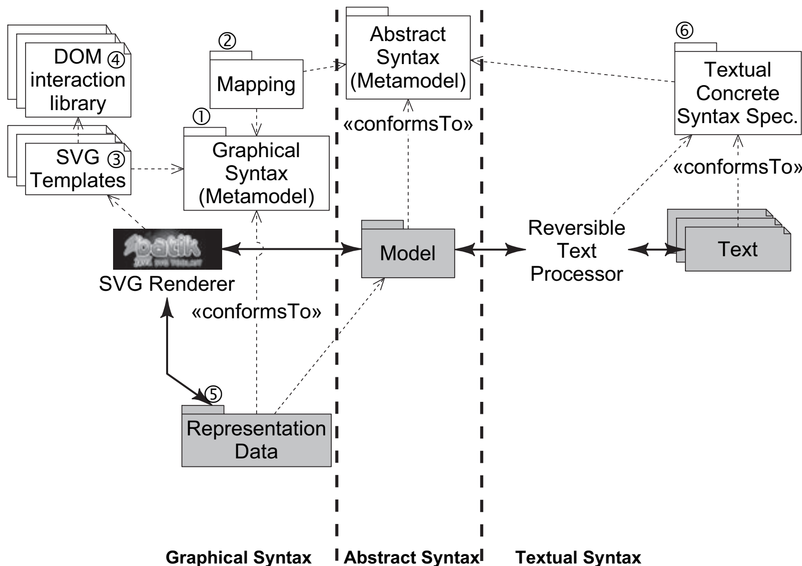


Figure 1.2: Concrete Syntax for Metamodel-Based Languages Revisited

standard agreement. Still, one needs to design actual representation. There is no agreement on a standard language for representation as defined by figure 1.1. That is why we propose to apply a template-based approach that would make use of the SVG well agreed standard for two-dimensional vector graphics. However, SVG needs to be extended so that it can communicate with the model. Finally, model may be viewed by a simple SVG renderer, such as Apache Batik. To make it possible to interact with the representation, we propose to use predefined interaction components that can be enabled in SVG templates.

Regarding *textual concrete syntax*, we propose to have a unique specification both for analyzing and synthesizing text. The goal here is limited to representing a model of a given language into a concrete syntax of that same language, so that we do not need as much flexibility as depicted in figure 1.1. Indeed, the goal is not to cross the boundaries of a language. Some technologies (as model transformations) are extensively studied, improved and developed regarding transforming a model of a given language into a model of another language (exomorphic transformations). We have the feeling that it is better approach to work at model level rather than at concrete syntax level to bridge languages, since the concern of concrete syntax is not vital in this context, and since one could benefit from dedicated technologies as promoted by MDE-like technologies. As an example, one should better trans-

form his/her UML model into a Java model, which conforms to the Java metamodel, further rendered in text using the Java concrete syntax, instead of directly generating Java code. If the target concrete syntax evolves (as it happens when new keywords appear), code generation has to be maintained while a model transformation that works at the abstract syntax level is still valid. To specify such a reversible textual concrete syntax specification, we propose a new (DSL) modeling language to put into relation a text structure and a metamodel. Led by a specification written in such language, automated tools may produce a model by analyzing a text, or represent in textual form a model. We define this language by stating its abstract syntax in the form of a metamodel. Of course, such new metamodel may be used to provide a concrete syntax to itself.

1.3 Contributions

The goal of this thesis is to study the relationships between abstract and concrete syntaxes, in the context of metamodeling. We investigate solutions for specifying both textual and graphical concrete syntaxes.

The main contributions of this thesis are the following:

- a metamodel for (reversible) textual concrete syntax specification language as companion of the metamodel for abstract syntax (as in figure 1.2 ⑥),
- a mapping specification technique to keep two models synchronized (as in figure 1.2 ②),
- an algorithm to graphically depict a model using SVG templates,
- a set of concepts and components to interact with models depicted in SVG (as depicted in figure 1.2 ④).

1.4 Plan

The structure of the document is as follows. In chapter 2, we will emphasize the importance of languages in system development. We will also take a tour of the state of the art of MDE-like technologies applied to language definition and concrete syntax specification. Language engineering and engineering through languages is exemplified by Netsilon that will be detailed in chapter 3. Netsilon can be compared to a model-driven text generator, and will be of inspiration to the solutions we will propose after. We will also see an example of SVG text generation using Netsilon that will indicate that SVG templates may be interesting to define graphical concrete syntaxes. We propose, in chapter 4, a metamodel for textual concrete syntax specifications that is inspired from concepts of Netsilon. In chapter 5, we propose an architecture for specifying graphical concrete syntaxes. All the propositions we make will be illustrated by examples. As a conclusion, chapter 6 will outline that further

work is still necessary for reaching agile language engineering. To this end, we discuss techniques regarding metamodeling and model transformation thanks to tag-based extension mechanisms in appendix A.

Chapter 2:
Language Driven Engineering

The context of this thesis is the definition of methodologies based on models. We named such paradigm Language Driven Engineering (LDE) even though different names may be found in the literature. We will first present in section 2.1 what is our view of LDE and what are its various flavours as published in literature or implanted in tools. Major contribution of this thesis is improvement of concrete syntax definition for modeling languages in LDE methodologies. Nevertheless, problem of concrete syntax definition is not new in computer science, and may find some solutions in other technological spaces [KBA02] as presented in section 2.2.

> *Section 2.1.1 was published in the WISME@UML04 workshop [FS04], and section 2.2.2 was part of a publication in the Model Driven Architecture - Foundations and Applications, First European Conference (ECMDA-FA) held in 2005 [FB05].*

2.1 Language Driven Engineering (LDE)

Actors as important as IBM, Microsoft, or the OMG propose a set of techniques and/or tools based on models to software development. If many publish a software development process pattern with its own name (e.g. MDD, MDE, MIC, MDA, as we will see in section 2.1.2 and section 2.1.3), all of them share a certain number of ideas, and offer common possibilities. We describe in section 2.1.1 what we believe to be most interesting properties and possible applications of those patterns, commonalities that we will further refer to as Language Driven Engineering (LDE) which emphasize importance of modeling languages. As prescribed by the Domain Specific Language community, LDE promotes the usage of various different "small" dedicated languages.

2.1.1 A Method for Methodology

The idea promoted by LDE is to use models at different levels of abstraction for developing systems. Thus, the main activity of LDE developers is to design models, just like they used to develop code, but led by a more precise methodology. The advantage of having an LDE methodology is that one should clearly define each step to be taken before the development activity has even started. As a consequence, developers are more or less forced to follow a methodology. The LDE methodology should specify the sequence of models to be developed, and how to derive a model from another one at the abstraction level immediately

above it. By providing developers with such a methodology, they are supposed to know at any moment during the development life cycle what is to be done next and how to achieve it.

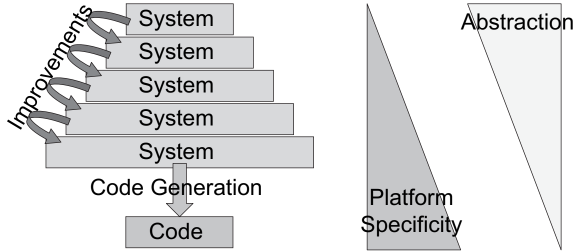


Figure 2.1: LDE Concept of Successive Improvements

Applying an LDE process is depicted in figure 2.1. The system under development is first described by a model at a very high level of abstraction, i.e., ignoring any kind of platform-related dependencies. This kind of model is intended to capture only the system requirements, without specifying how to achieve them; it is the description of the problem. Good candidates to play such role are use cases [Jac04] and feature-oriented diagrams [KCH+90]. A series of automatic yet interactive improvements may then be performed that have the responsibility to make the system description more platform-specific at each step. Improvements may be performed by refinement, generation, transformation, refactoring, etc. For instance, the system may be expressed once again, but more precisely this time, by class diagrams and state diagrams [Har87] to show business behavior. Further on, additional information can be added, e.g. to integrate the distribution concern, further complemented by directives for the system to work on the CORBA platform [Sil06] (see also appendix A).

At each step of an LDE process, information related to quality management could be integrated as well, such as verification, validation, and test case generation as promoted by the V-model [BD99] as shown by figure 2.2. A verification might be the action that checks whether a more concrete model does not break the specification promoted by its abstract specification model, or vice-versa in the case of reverse engineering. A validation step may allow system developers (or even clients) to instantiate prototypes out of intermediate models in order to test their features before the system is fully implemented. Automatic test case generation may produce as outcome scenarios, i.e., sets of messages that are supposed to be sent and received by the working system, allowing in this way to test its actual implementation (e.g. in [NFTJ06]).

One of the most important advantages of using an LDE process is its robustness to changes. When a change occurs, be it at the highest level of abstraction (e.g., a change in the

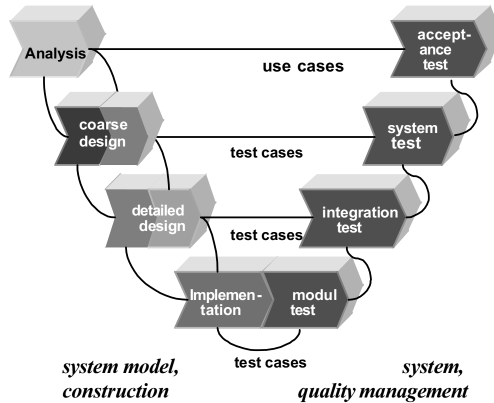


Figure 2.2: The V Model

requirements of the system) or at a lower level of abstraction (e.g., moving to another platform, such as moving from PostgreSQL to MySQL), its impact is well localized and the parts that are not touched by the change are immediately reusable. However, the improvements have to be performed once again in order to "update" the changing parts. It becomes more problematic when the modeling language changes because such re-improvements are not directly possible. Note that re-improvement requires a comprehensive tool support.

In order to apply LDE-inspired processes in large projects, which typically involve many developers and tools, several issues have to be addressed, such as model interchange, diagram interchange, model versioning, concurrent management, and so on, but these are more tool-related issues than methodology issues. Nevertheless, they remain problems that will have to be addressed sooner or later [ALPT03].

An LDE process should thus define:

1. how many levels of abstraction are there, and what platforms have to be integrated;
2. what are the modeling notations and the abstract syntax to be used at each level of abstraction;
3. how refinements are performed, and what platform and additional information they integrate into the lower level of abstraction;
4. how code is generated for the modeling language used at the lowest level of abstraction, and perhaps even how to deploy that code;
5. how can a model be verified against the upper level model, how can it be validated, and how can it generate test cases for the system under development.

The first important technique is *metamodeling* [AK02], which allows methodologists to define precisely a class of models. Metamodeling clearly defines a modeling language by

specifying its abstract syntax, eventually along with its semantics. We also believe it is of paramount importance to define its possible concrete syntaxes in order to allow conforming models to be viewed and modified by different human stakeholders using the different modeling notations that are available in a given view. If we have a closer look, one level of abstraction is already defined by the modeling languages to be used. Therefore, defining the corresponding metamodels solves already the first two duties of methodologists (points 1 and 2 presented above). Moreover, if the semantics is clearly defined, it is also possible to perform the validation part of point 5.

The second technique is *model transformation* [SK03]. This technique allows methodologists to clearly define relationships between models. Model transformations depend only on the metamodels of the related models. Methodologists may use this technique to clearly specify the refinements between models. An ideal model transformer should be able to perform both forward and reverse transformations, which will allow to propagate changes to models at lower, respectively upper, levels of abstraction, enabling the possibility of automatic synchronizations and re-improvements. Moreover, it gives the possibility to verify the more concrete model against the more abstract model, and vice-versa, at any moment during the development life cycle. As soon as such bidirectional tool will be available, point 3 and the verification part of point 5 will be solved as well. Note that some solutions are being studied regarding model refactoring using model transformation [MB05], all the same regarding semantics [CESW05, MB06].

For solving point 4, *code generators* are needed, which would map a model to some textual or binary files.

Referring to point 5, an important problem is the current lack of a tool-independent solution and of appropriate modeling notations for completely specifying how to generate test cases out of models, how to perform validation and deployment, how to depict such models, and so on. Moreover, one should be aware that the proposed list of artefacts to be delivered by methodologists is not at all complete, and new points will probably have to be added as LDE moves along.

Examples of MDE processes and applications are provided in chapter 3 and in appendix A. In the following of this section, we detail technologies introduced above.

2.1.2 Published Techniques

LDE is not a well-known technique but our analysis of many different software development process patterns promoting models as first class artefacts. In the literature, one may find many other TLAs (Three Letter Acronyms) to embrace any model-based software development recommendations and paradigms. Most famous terms include Model Driven Engineering (MDE) [Ken02], Model Driven Development^{TM} (MDD) [MCF03] as registered by the OMG, or Model Driven Software Development (MDSD) [VS06]. Even thought they defend the same principles as LDE, we preferred to introduce yet another such acro-

nym to emphasize importance of the language part. In section 2.1.2, we take a tour of the most highlighted techniques of the moment here, namely MIC, MDA, and Software Factories. In order to be applied, such paradigms need to be supported by tools and standards. We will introduce some of the most regarded ones (in section 2.1.3) after some general remarks on Domain Specific Languages and the UML generic purpose language.

2.1.2.1 Model Integrated Computing

As from mid-nineties, Model Integrated Computing (MIC) [SKB+95, SK97] promotes domain-specific models as primary artefacts to software development. First motivation for MIC was to supply complex embedded software engineering [KSLB03] with methodology and accompanying tools [LBM+01].

MIC proposes a methodology decomposed in two different phases. The first phase is achieved by software and system engineers and consists in analyzing the application domain. The goal is to find appropriate modeling paradigms together with a formal modeling language definition. An automatic tool can then use those artifacts to generate a domain-specific modeling environment. That environment is directly usable by engineers of the domain to achieve the second phase, that is modeling the desired application.

The GME (Generic Modeling Environment) toolset [Dav03] implements MIC ideas. In its last version (6.0), it is integrated in the Visual Studio .NET environment from Microsoft, and proposes languages and tools for model and language engineering. However, semantics for such developed modeling languages are not directly supported.

2.1.2.2 Model Driven Architecture

Model Driven Architecture® (MDA) [MM03] is an OMG initiative to software development publicly available since 2000. Basic idea is to «clearly separate the specification of the operation of a system from the details of the way that system uses the capabilities of its platform». To do so, MDA defines a specification architecture structured in Platform Independent Models (PIMs) and Platform Specific Models (PSMs).

Such architecture makes it possible to deploy a system on various platforms thanks to standard projections, and supports platforms and techniques evolution. Applications may interoperate exchanging models. MDA is entirely implemented by means of models and model transformations.

To promote MDA, OMG has held an important standardization process regarding modeling techniques. It was first proposed to use UML (see section 2.1.2.5) as a universal language. However, this approach rapidly appeared far too rigid and the profiling tag mechanism (see section A.1.1 on page 131) was proposed to add new notions to the UML language. As these extensions grew, MDA community opted for a domain specific language approach (see section 2.1.2.4) and OMG proposed new standards to modeling language definition and manipulation (see section 2.1.3).

2.1.2.3 Software Factories

Software Factories [GSCK04] are Microsoft's vision to LDE. They were inspired by assembly lines in industry. Main inspiring ideas are the following:

- Assembly lines usually manufacture only one kind of product with small variations points. An example is the automotive industry where a factory produces one kind of car, with some possible variations in the color, or option combination.
- Workers are often specialized. If one can meet a worker with a broad scope of activities, those activities never cover the full assembly line.
- Tools are very specialized and automated; they cannot be reused in other assembly lines than the one they were conceived for.
- Components to be assembled or tooled often come from third parties. Automotive assembly lines usually only assemble pieces that are normalized or produced in other factories.

These principles have long proven their effectiveness in manufacturing hardware product families. Software factories propose to apply these characteristics to software development. Following the first and second points, software suppliers and software developers should be highly specialized. Third point suggests that (modeling) tools should also be specialized (i.e. domain specific), including (modeling) languages, wizards, and transformations. Last point motivates the (re)use of off-the-shelf components. The Microsoft Visual Studio .NET 2005 IDE now features these ideas and propose an extensible framework that may be configured to scope a specific domain.

2.1.2.4 Domain Specific Languages (DSL)

Domain Specific Languages [vDKV00] are small languages with a rather poor number of different abstractions as they are used only in a very specific technological space by a modest number of users; that's why they are often referred to as micro-languages or little languages [Ben86]. If they usually lack flexibility, they are very adapted to the problem they are designed for, being also more readable for specialists of the domain than general purpose languages. Many illustrative examples of how to design and use DSLs are available in the literature, for instance in [KMB+96].

LDE extensively promotes the usage of DSL [CESW05, KP02]: for sake of productivity and understandability, language of models should be adapted to the problem domain and the considered level of abstraction. Note that for a given level of abstraction, many different DSLs may be used. This idea is especially promulgated by the Domain Specific Modeling community.

One of the problem with DSLs is that, because of their relatively restricted communities, it is economically hard to develop and maintain corresponding integrated environments (IDEs) or CASE tools of high quality. This is starting from this idea that many CASE tools

generators started to appear like GME [Dav03] (see section 2.1.2.1), DOME [Hon92], MetaEdit [Poh03], XMF-Mosaic [CESW05] from Xactium, or more recently DSL Tools [GSCK04] from Microsoft. All these tools are of great interest as they feature a graphical concrete syntax definition facility, often separate abstract (domain data) from concrete syntax (surface language), and sometimes permit to define textual concrete syntax (like for Xactium).

2.1.2.5 UML

The Unified Modeling Language (UML) [AAB+07] is a general purpose language standardized by the OMG. A first draft version was available in 1995 (Unified Method v0.8). It was born from a unification concern regarding object-oriented modeling languages. Indeed, at that time, many had developed modeling languages to model object-oriented applications. Not only those languages targeted the same requirements, but they featured the same concepts too (e.g. objects, classes, association, use cases, or subsystems). Fragmentation in the languages, and consequently in the tools was a brake on the development of modeling [Iiv96]. UML is a fusion of some of the most interesting languages of that time (OMT, Booch, OOSE, Harel's statecharts, etc.). One can say objectives are reached as its success became rapidly widespread and UML is now the de facto standard for modeling object-oriented applications.

UML is actually a set of coherent modeling notations. One may capture requirements using use case diagrams and scenarios, data structure using class and object diagrams complemented with constraints written in OCL [ABF+06], message side effects using statecharts, hardware and software architecture using deployment and components diagrams, etc. Possible motivations to use UML are documentation, communication between actors of the project (e.g. architects, developers, testers, clients), prototyping (in case of executable UML models), abstraction (permitting, trough code generation, improvement in productivity compared to code handwriting - as shown in chapter 3), or merely because it is the notation prescribed by the method chosen by the project manager. It is important to note that UML is a notation, and not a methodology, even though many methodologies use UML as a notation (e.g. RUP [Kru03] or Fondue [SS99]). Appendix A will provide an example of a system specified using UML class diagram notation, extended with profiles and clarified by OCL constraints.

Even though UML extensively promotes modeling for software engineering, one can say it breaks LDE principles in its original philosophy. Indeed, LDE promotes language proliferation while UML targets an universal notation for modeling. If UML proved it is well suited to model general object oriented applications, its lacks comprehensive concepts while diving into details of domain specificities. To supply UML with more flexibility, UML proposes UML profiles as a tag mechanism so that one can add such missing concepts.

2.1.3 Key Technologies

One of the key points in productivity gains of LDE techniques is that they are supported by automation tools. We complement our description of LDE with the description of those key technologies as identified in section 2.1.1, that are metamodeling, model transformation, code generation, and concrete syntax specification.

2.1.3.1 Metamodeling

Metamodels are models about models. Many different definitions may be found, but in the context of this thesis, we will consider metamodels as the specification of the abstract syntax of a language, that is the lexicon of concepts of the languages (vocabulary), properties of those concepts, and relations between the concepts (taxonomy).

A metamodel is the description of a language through its concepts. A metamodel is described by mean of a language (a metamodeling language). Thus a metamodeling language, which has an abstract syntax, has a metamodel. The latter class of metamodel is called meta-metamodel. As a meta-metamodel is a mean to specify abstract syntax of languages, a meta-metamodel must be able to describe its own abstract syntax, thus breaking the necessity to create other models adding yet other meta prefixes.

We show here an illustrative example for what is described above, known in the MDA terminology as the 4-layer architecture. In this terminology, the modeling level is named M1, the metamodeling level M2, and the meta-metamodeling level M3. One may simplify the problem by considering only two level at same time (Mx-1 and Mx): an object level and a class level. The following example makes use of the UML object diagram and class diagram, respectively, to describe those two levels. The goal of the example is to describe an abstract syntax for a relational database design language. A concrete sentence of the language is given in figure 2.3. A `Teams` table lists teams with the column `name`, which is a primary key. Table `Match` consists of `Score1` and `Score2` columns, in addition to `Team1`, `Team2`, `Place` and `Date` primary keys, `Team1` and `Team2` being foreign keys referencing a line in table `Teams` according to `name` value.

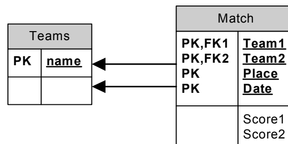


Figure 2.3: M1: A Relational Database Language Sentence

Figure 2.4 represents the abstract syntax of the relational database description language. Concepts of the language are `DataBases`, `Tables`, and `Columns`. A `DataBase` owns `Tables`, and a `Table` owns `Columns`. All these concepts have a `name`, and a column

has a type (i.e. text, number, or even column in case of foreign keys). Figure 2.5 shows an

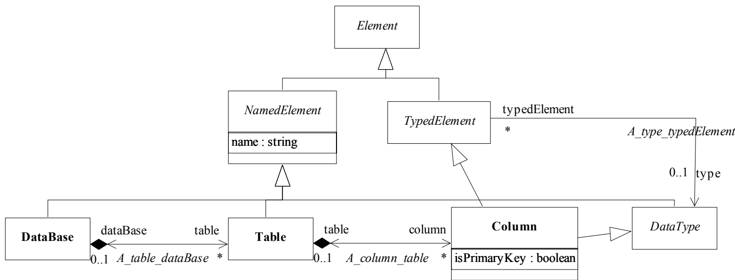


Figure 2.4: M2: Relational Database Language Abstract Syntax

excerpt of the abstract syntax tree for the sentence shown in figure 2.3 as occurrences of concepts of figure 2.4 using the UML object diagram formalism. We can see here that all information may not be explicitly represented in the concrete syntax, following the example of the `DataBase` `name` or types of `Columns`. Experts in metamodeling may have noticed that the M1 abstract syntax graph (ASG) of figure 2.5 is represented as objects conforming to the M2 (meta-)classes defined in figure 2.4.

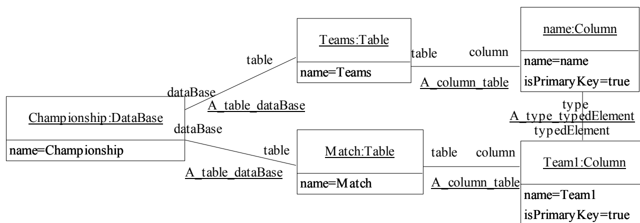


Figure 2.5: M1 conforming to M2: Figure 2.3 as an ASG (Excerpt)

As said above, to define metamodel as we did in figure 2.4, we used a metamodeling language, and figure 2.6 may be seen as a concrete representation of occurrences of concepts of a given metamodeling language with which can be described concepts of meta-class, meta-attribute, etc. This follows exactly the same pattern as before, shifted by an M-number. One such metamodeling language as standardized by the OMG in the context of MDA (see section 2.1.2.2) is MOF [ISO05, ACC+06], a simplified variant of UML class

diagrams. We show in figure 2.7 an excerpt for MOF 1.4 (corresponding to level M3), and in figure 2.6 an excerpt of abstract syntax graph for metamodel (M2) as instances of MOF 1.4. Although not shown here, concepts of M3 may be represented using M3, e.g. MOF

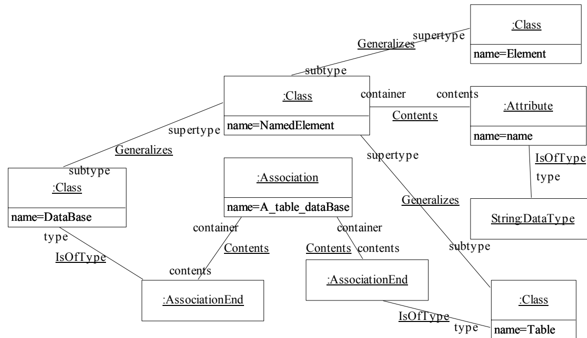


Figure 2.6: M2 conforming to M3: Figure 2.4 as a MOF ASG (Excerpt)

may represent concepts of MOF. As a consequence, there is no need to apply the Mx-1/Mx conformance pattern again and again. In this example, we didn't show an example for M0, the real world, but it is again the same pattern. One may think of most interesting matches of his/her favorite sport as M0 sentences, which are instances of M1, represented by lines in a relational database formatted as prescribed in figure 2.3.

The success of MOF as a metamodeling language is as broad as it is hard to find a really different approach nowadays. Actually, if one can find alternatives to the 4-layer architecture [Hof79], other tools and researches are subsets and/or extensions to MOF as EMF [Ecl05], Coral [Por03], KerMeta [MFJ05], or XMF-Mosaic [CESW05]. Even metamodeling tools which neglected abstract syntax before are aligning themselves to the standard, as GenGED [Bar98] which evolved into Tiger [EEHT05]. However, MOF cannot pretend being the historical metamodeling language, and tools may use older modeling languages or even traditional programming language. Two such examples are AToM^3 [dLV02], which uses Entity-Relationship diagrams and/or Python, and FAMIX [DMNS97], which introduced a language-independent metamodel that could be indifferently implemented in SmallTalk, C++, Java, or Ada. Still, we stick to the object paradigm as a mean to describe concepts.

If metamodeling languages permit to describe abstract syntax of modeling languages, one need implementations to store models. Politic applied by most approaches (MOF-Based repositories, EMF, AToM^3) goes beyond code generation. Actually, those tools generate a

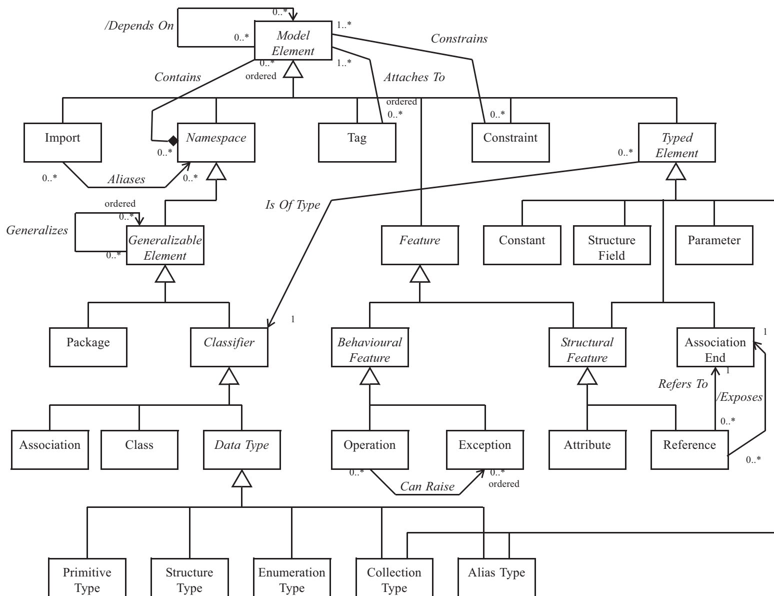


Figure 2.7: M3: MOF 1.4 (© OMG)

storage artefact for models with a specific API to make it possible to manipulate and/or listen changes in the model, independently from the storage technology. For instance, some tools, like MDR [Sun05], can be configured to target either in-memory hashtables, in-file b-trees, or relational databases, while offering the same data access API. These software components, together with their accompanying API, are called model repositories. The MOF standard defines precise transformation rules from MOF metamodels to CORBA APIs, which de facto standardizes CORBA-based model repositories. JMI [Jav02] is another standard for transforming MOF metamodels into raw Java APIs, which is in use in numerous tools like MDR. Model repositories are at the core of modeling tools and are most of the time a requirement for using model transformations, if not included in model transformation engines.

Moreover, to exchange models, the OMG issued the XMI standard [ACD+05], which is a way to represent a model in a textual XML file. XMI is flexible enough to comply to any metamodel as soon as this latter is described using MOF. Actually, a metamodel specifies an XML schema (DTD or XSD) of XMI representations of conforming models.

Because this technique is dedicated to tool communication and is rather hardly readable by humans, OMG also issued HUTN [DDF+04], which follows equivalent principles, but using a more human-readable grammar.

For abstract syntax to be rendered as precisely as possible, metamodeling standards and tools often propose to complement metamodels with constraints. Constraints may be expressed using in any sort of language capable to query a model, for instance Java code that uses JMI interfaces, or Python in the case of $AToM^3$. MDA proposes OCL [ABF+06], a side-effect free language dedicated to navigate class-based models, regardless they are MOF metamodels or UML class diagrams (see section 2.1.2.5).

2.1.3.2 Model Transformation

Model transformation is the second important technique to LDE as underlined in section 2.1.1 and in [SK03]. Many survey of model transformation techniques and standards are available, for instance in [CH06].

Model transformations make it possible to build bridges between modeling languages for dedicated purposes. They may be compared to compilers in traditional language engineering as they can translate models of a certain metamodel into models of other metamodels, for example transforming a class model into a relational model [MFV+05]. However, their usage is much broader as they can cover a full exploration and manipulation of models. For instance, one may compute metrics about models (e.g. in [VBBJ06]), perform model refactoring (e.g. in [MB05]), or precise models according to a platform (e.g. section A.2 on page 134).

Regarding model transformation, different approaches may be taken. First, as models may be represented by XML files, many used the XML transformation language XSL. If this was a solution at the early times of model transformation, its has shown neither to be scalable nor agile from a software engineering point of view. Another way to transform models are transformation languages which are dedicated to a given metamodel, following the example of J from Objecteering that is dedicated to UML model transformation. If they particularly fit to transform a certain type of model, since one may not need to know the underlying metamodel to write a transformation, they cannot be applied to other metamodels and are thus not general solutions to model transformation. A third way to transform models is to take advantage of model repositories as described in previous section: as a model repository proposes an API for model manipulation, one may use a general language (e.g. Java or Python) to access the API and thus manipulate models. Last technique is to use a general purpose transformation language, i.e. a DSL whose domain is model transformation.

Regarding that last possibility, OMG is establishing the Query/View/Transformation (QVT) standard [CCD+06]. Actually, QVT proposes various solutions to transformation. It proposes compatibility standard to make use of non-QVT transformations, it proposes two

declarative languages based on pattern matching as inspired from graph grammars (see section 2.2.2.1), and an imperative language. Moreover, an impressive number of languages and associated tools appeared those last years. Examples are ATL [BDJ+03], the Borland Together tool [Fon06], MTL [fRiCSI05, VJ04], MOLA [KBC04], and BOTL [BM03].

Another way to transform models has been proposed. One of the KerMerta's metalanguage goal is to add executability into metamodels [MFJ05]; as a side effect Kermeta is able to transform models whose metamodel is defined using KerMeta [Fle06, MFV+05].

2.1.3.3 Code Generators

Resulting artefacts of the application of an LDE process are models. However, models are rarely usable as is and there is often a need for a translation to artefacts that are directly understandable by execution platforms. An example of such artefact is program code, which can be compiled into an executable file. To do so, there exist several code generators from models.

Apart from modeling tools dedicated to only one sort of model, and featuring a specific code generator wired to a certain platform, most code generators has applied a template-based generation. This technique makes it possible to specify, for each metaclass, some text in which are interleaved queries to the model (repository). One may compare such technique with dynamic web applications, written in scripting languages as JSP or ASP, which are HTML templates in which are interleaved queries to a database. Examples of tools that apply such template-based technique, and which are taking advantage of the Apache Velocity template engine, are Fujaba [GSR05], and AndroMDA [Boh07] that both offer an evolved template management (through cartridges). Other examples of tools offering a specific template syntax are Acceleo [Obe07], XMF-Mosaic [CESW05], and openArchitectureWare [Voe06]. OMG, in the context of MDA, is studying the MOF2Text standard to code generation [CIM+06] that is also based on text templates interleaving instructions and OCL expressions. Even though MOF2Text Request for Proposal proposed bidirectional specification (from model to text and back again), its current status allows only one way generation (from model to text).

Parallax [SS05] is a code generator dedicated to UML models. Parallax offers an interesting management of platform specificities (crosscutting concerns). Actually, a code generator is a simple program that builds an abstract syntax tree of the target language (e.g. Java). Nevertheless, when the model has to be adapted to a specific needs (e.g. distribution), for instance thanks to a profile, code generation facility needs to be reconsidered to deal with that new profile. Parallax promotes the usage of aspects [KLM+97] to modify the generation process. To solve that kind of problem, openArchitectureWare [Voe06] also offers an aspect-oriented mechanism to adapt code generation, but this time directly at template level.

Problem with code generation is that tools tend to integrate platform specification within the generation, for instance a generator that transforms a UML model into an Java EJB application. We believe that it is not an agile approach because one need to deal with two different concerns at same time. On one hand, one need to consider how concepts of the source language (e.g. the UML metamodel) should be mapped into concepts of the target language (e.g. the Java metamodel). On the other hand, one need to deal with concrete syntax of the target language (the Java textual grammar). A better approach could be to make an additional model transformation and then a code generation. In this case, model transformation works at concepts level, and code generation does not crosses anymore the language boundaries. For instance, one may better propose a model transformation to distribute his/her UML model, then a transformation to an EJB model, and finally a code generation to Java/EJB. Going one step further with this philosophy, one may regret the lack of bidirectional code generation. Indeed, if one introduces metamodels for common languages (as Netbeans did for Java [Net05]), it may be interesting to be able to generate code out of the model, but also to get the model from some source text.

2.1.3.4 Graphical Concrete Syntax

To define a language in general, and a modeling language in particular, one need to specify concepts (i.e. abstract syntax, that is metamodel in the case of modeling languages), semantics, and concrete syntax. A concrete syntax is a surface language that acts as an interface between the instances of the concepts, and the human being supposed to produce or read them. Concrete syntax may be textual or graphical, but is often a mix of both.

Up to a recent time, graphical concrete syntax was hand coded in diagram editors using two-dimensional drawing libraries such as Swing or GME. Problem with this approach was that it was hard to read for humans, and that it was also long to develop and maintain.

Most solutions to graphical concrete syntax definition on top of a metamodel are based on ad-hoc symbol editors, following the examples of $AToM^3$ [dLV02], GME [Dav03], or MetaEdit [Poh03]. For each representable element of the metamodel, one defines an icon and indicates properties to be displayed. Such tools make a clear distinction between entities and relations, thus making possible to design only a nodes-and-bounds type of graph, so called connection based language (see section 2.2.2).

Other solutions propose to map the abstract syntax to a graphical language described as a metamodel, as proposed in [DV02]. For instance, XMF-Mosaic [CESW05] proposes to write a bidirectional transformation to a metamodel which has a graphical representation semantics, with squares, lines, etc. A similar approach does not require a model transformation, but rather models that capture information regarding abstract syntax representation metamodel mapping (see figure 1.1 on page 5). Topcased [VPF+06], or GMF [Ecl06] are

such tools, so does Tiger [EEHT05] even though representation metamodel is not explicit. Still, all these solutions stick to connection based graphical languages.

When it come to concrete modeling, concepts are manipulated by mean of their representation. To do so, one needs to define how icons can be manipulated. As we will discuss in more details in section 2.2.2, graphical models, in contrary to textual specifications, are built interactively by the modeler who sends events to the modeling tool, e.g. to create a modeling element, or to move it. Actually, numerous DSL tools, based on concrete syntax description, offer standard behavior to defined representations (e.g. to create, delete, move, or connect icons). Other approaches take advantage of graph grammars to describe such behaviors [BGdL06], following the example of Tiger or AToM^3.

One should not confuse graphical concrete syntax specification with diagram interchange, as XMI-DI [ADG+06], which is a standard of the OMG. XMI-DI enters the category of `Representation Data` as shown in figure 1.1 on page 5. If graphical concrete syntax deals with concrete representations of concepts given by a metamodel, diagram interchange is only a mean to store graphical information of a diagram. Actually, diagram interchange is not a mean to describe how abstract and concrete syntax relate. On the contrary, it is a mean to tell, for a given model, which elements of the abstract syntax tree are represented, and what are the chosen variations in the representation (e.g. place, size, color, provided that information is not reflected in the abstract syntax graph). To summarize, if concrete syntax makes explicit all possible representations for a given concept, diagram interchange makes explicit what is the chosen variant in representation for rendering concept instances.

Regarding textual concrete syntax, current solutions mix code generation with manual text parsing, both of them having to be consistent, even though some tools may help in parsing texts to create models [HRS02, CESW05]. We do not consider automatic syntax generation, as in MOF2Text or [AP04, GRS06] since concrete syntax must follow specific rules in its form. We are only aware of TCS [JBK06] that recently offered a comparable approach to the one we will describe in chapter 4 and that we studied in [FSGM06] and in [MFF+06].

2.2 Traditional Concrete Syntax Engineering

This thesis attempts to give some preliminary answers to concrete syntax definition for modeling languages, answers that will be inspired by solutions outside the LDE community. Textual and graphical definition of concrete syntax are technical spaces by themselves and deserve to be introduced in section 2.2.1 and section 2.2.2. We also introduce in section 2.2.3 the SVG standard, which will be the basis for a solution to graphical concrete syntax specification.

2.2.1 Traditional Textual Language Engineering

Compiler construction was one of the very first way abstraction appeared in computer science: in early 60s, it was a revolutionary idea to make an automatic translation from a "computer scientist readable" textual specification to computer understandable binary code. Productivity gains triggered by simplifying the program specification led to extensive research on finding the best possible abstraction for programming a computer (Algol, Fortran, C, etc.). Compiler construction techniques evolved as the number of experimented languages increased.

Among the most interesting results of compiler construction experience are generative grammars [Cho56], attributed grammars [Knu68], and Backus Naur form (BNF) [BBG+60], improved by the Extended BNF (EBNF) [Int01], which are all able to describe structured texts. Those results are used in compiler compilers, i.e. compilers that take as input an EBNF-like grammar, and which generates a program able to analyze a text conforming to that grammar. Such compiler compilers are Lex/Yacc [Joh79], or ANTLR [PQ95], and nowadays their success is as universal as that they are taught in universities.

Their general principles are the following. First, one should describe so called terminal symbols, such as numbers, comments, keywords, etc., in a lexer grammar. From this grammar, the compiler compiler generates a lexer, that is a program which takes as input a text and transforms it into a string of such defined symbols (tokens) - this phase is called the *lexical analysis*. A token is associated to its type (`Number`, `Comment`, `Keyword`, etc.), a text (e.g. `"12.85"` for a Number), and eventually a location (e.g. a line and a column). As a second step, a parser should be defined usually in the form of EBNF-like rules (an `expression` is either an `addition`, or a `subtraction`; an `addition` is a `Number` terminal, a terminal `+`, and another `Number` terminal). Again, the compiler compiler generates a program. This latter program, the parser, takes as input the token stream obtained from the lexer and analyses it according to the grammar - this phase is called *syntactic analysis*. The outcome is a *concrete syntax tree* that organizes the tokens into a hierarchy following the order of triggered rules. Most of the time, actions can be embedded in the parser rules so that one may build an abstract syntax tree that may be later walked through to generate code (e.g. if parser analyses an addition, it should build an addition object and associate the corresponding operands as analyzed from terminal numbers). Unfortunately, abstract syntax tree construction (*semantic analysis*) and code generation are poorly automated tasks and should be programed using general purpose languages.

Parsers may be classified in two categories. *Top-down parsers* (*LL* parsers - from Left to right using Leftmost derivation) attempt to trigger the topmost rule (the main rule), and move down into grammar rules. For instance, if `expression` is the topmost rule, an LL parser will trigger the rule and then choose among the alternatives. On the contrary, *bottom-up* parsers (*LR* parsers - from Left to right using Rightmost derivation), also known as shift-reduce parsers, will store tokens into a stack up to a rule is matched. For instance, it is only

when the token stack contains a `Number`, a `+`, then another `Number`, that the `addition` rule then the `expression` rule will be recognized.

Parsers usually drive their associated lexer, that is, they consume token at same time they analyze rules. This raises an important drawback in that some rules may be conflicting. For instance, at the moment an LL parser consumes a `Number` terminal, it cannot decide whether the triggered rule should be addition or subtraction. To solve that problem, two politics are possible. *Lookahead* parsers can analyze more than one token at a given moment. In the operation example, the parser can decide between addition and subtraction looking at the next token, that should be either a `+` or a `-` terminal. Depending on the grammar complexity, that number *k* of token to be accessible in advance is different, but the more important that number is, the more complex and the less efficient the parser is. The parser is said to be *LL(k)* or *LR(k)*. An usual value for k for common programming languages is 3. *Backtracking* provides a second solution to the conflicting rule problem. Actually, in case of conflicting rules, such parser chooses an alternative and continue analysis "hoping" the right alternative was chosen. In case an analysis fails, the parser goes back to its last choice and takes another alternative. For instance, such parser will chose the `addition` rule each time it will encounter a `Number` terminal. If the next token is a `-` terminal, the `addition` rule fails, and the parser backtracks to state at the time of the choice and starts again analysis using the `subtraction` rule alternative. Note that both lookahead and backtrack technologies are compatible.

A thorough description of compiler construction may be found in [ALSU06].

2.2.2 Graphical Language Engineering

Almost each of today's modeling languages comes with a graphical representation in order to improve readability and usability. Thus, the concrete syntax of modeling languages should be defined in terms of a visual language. For this reason, we summarize here the relevant basic terms from visual language theory.

A *visual language* describes a set of visual sentences which in turn are given by a set of visual elements. A *visual element* can be seen as an object characterized by values of some attributes. It depends from the language which attributes are important for a graphical element[1], some of the most frequently used attributes are *shape*, *color*, *size*, *position*, *attach regions*. We will further refer to such data as *representation data* in the rest of the dissertation.

Visual elements are arranged in the two- or higher-dimensional space. For some languages (which are classified in [CLOP02] as geometric-based languages) the position of visual elements is an important information, e.g. a sentence consisting of a circle and a

1. There is a common classification of attributes into graphical, syntactical and semantic attributes. Only the first two classes of attributes are relevant for our approach because semantic attributes are already captured by the abstract syntax definition.

square where the circle is placed at point (1,0) and the square is placed at (2,0) is another sentence than the sentence where the square is placed at (1.0) and the circle is placed at (2,0). However, for real-world languages, the absolute coordinates are much less important than the spatial relationships between graphical objects. Some of the most frequently used spatial relationships are `RIGHT`, `UP`, `CONTAIN`, `OVERLAP` (see [CLOP02] for a more complete list). It heavily depends on the visual language which of the spatial relationships are considered to be important. An example is the chessboard language as introduced in section 5.2.2 on page 87. Sometimes, languages are geometric-based even if it seems that the visual elements can be arranged freely. One example is the language of UML class diagrams. At a first glance, rectangles for classes can be placed freely at any point in the space. For instance, a diagram consisting of two rectangles labeled with A and B would always be read as the same sentence no matter where the (rectangles for) class A and B are placed. However, there is one exception from this rule: If - let's say - the rectangle for B appears completely inside the rectangle for A, then the class A is read to be composed of class B. Thus, the spatial relation `CONTAIN` is important to define the visual representation of class diagrams whereas the relations `RIGHT`, `UP`, etc. do not play any role here.

In addition to geometric-based languages, there is another group of languages called connection-based languages. This languages allow the visual elements to be completely freely placed in the space and, thus, none of the spatial relationships is important when a sentence of such languages has to be read. Instead, it is an important information whether two elements are connected by a *connector* (usually a line, polyline, curved line) or not. Connectors start and end in special regions of visual elements, so called *attach regions*. A visual object can have one or more attach regions which can sometimes collapse to attach points. As already mentioned, visual language definitions formalize the attach regions of an visual element just as attributes of it. This abstracts from the problem to define where an attach region is exactly located in respect of the visual element (e.g. in the lower right corner). However, there are symbol editors available, e.g. as part of VLDESK [CDP04] or $AToM^3$ [dLV02], that allow to exactly define attach regions for a visual element as well as to solve the very similar problem of defining a shape for it. In fact, most real-world visual languages show characteristics of both geometric-based and connection-based languages and are thus called *hybrid* languages.

To summarize, a visual sentence is given by a set of visual elements together with their values of (language dependent) attributes and information on holding (language dependent) relationships between them. The definition of a visual language might restrict the possible relationships between elements and, for instance, say that the rectangles representing a class A must contain the rectangles for all classes A is composed of.

Finally, visual elements can completely hide other elements. Thus, by looking on the visual sentence, it is impossible to distinguish between the visual sentence containing the hidden element from a sentence where the hidden element is dropped. This problem can easily be fixed by introducing in the language definition a spatial relation `HIDDEN_BY` and

adding a constraint, that visual elements should never hide completely another visual element.

2.2.2.1 Graph Grammars

As explained above, a diagram, i.e. a visual sentence, is a set of parametrized component instances, together with spatial relationships (`RIGHT`, `UP`, `CONTAIN`, `OVERLAP`). Thus, a diagram may be seen as a directed graph and may be defined and transformed using graph grammars.

Graph grammars [Nag76, Göt82, EEKR99] (and graph transformations) are constructive approaches to graph evolution based on pattern matching and have its roots in textual language engineering (see section 2.2.1), and was source of inspiration for QVT (see section 2.1.3.2). A graph grammar is a collection of *production rules*. Production rules, which may be attributed, are composed of a left-hand side, which provides a pattern to match (i.e. search criterion) in the graph, and a right-hand side, which describes the modifications to apply to the matching region.

Graph grammars are able to define formally visual languages [Min02]. The same technique may apply to define how to construct a valid language sentence (i.e to add a new element to a diagram) and to define what is the semantics of the language (i.e. to make a system state evolve according to a language sentence) [And96]. To make the sentence evolve corresponds to triggering one of the rule in the graph grammar: a user interaction, like adding a new element, connecting elements together, renaming an element, changing colors, etc., actually correspond to a production rule triggering. The interested reader may find an introduction and some examples of visual language definition in [Jon90]. Moreover, some tools implement those ideas as GenGED [Bar98] and AToM^3 [dLV02].

Triple graph grammars [Sch94] are an extension of graph grammars to keep consistent manipulated data. Production rules contain intermediate rules, the correspondence production, that relates left to right hand sides. This allows to maintain correspondence data that are necessary to keep source and target model consistent according to the grammar when one of the two model changes. Unlike simple graph grammars, triple graph grammar allows one to separate clearly abstract syntax from concrete syntax: the grammar states how the abstract syntax tree may be represented by the concrete syntax tree; if the concrete (respectively abstract) syntax tree is changed, the abstract (respectively concrete) syntax tree is automatically changed accordingly.

2.2.3 Scalable Vector Graphics

Scalable Vector Graphics (SVG) [JN05] is the only open standard for describing two-dimensional vector graphics. SVG is not related to a platform, thus it can be in use on platforms as various as computers (regardless of the operating system) or mobile devices. The scope of standard is very broad makes it possible to draw basic shapes (circles, rectangles,

Bézier curves, etc.), to integrate and mix raster images, etc., with a large number of possible customizations (e.g. animation and transparency).

One of the most important characteristic is that SVG is an XML [BPSM+06] dialect. The eXtensible Markup Language (XML) is another open standard for structured data interchange based on text files. As a consequence, as any XML specification, SVG may be manipulated by the Document Object Model API (DOM) [HHW+04]. The DOM API allows to access and alter XML data using languages (like Java) or scripts (like JavaScript). This means that a script or a program may manipulate an SVG document that an SVG interpreter will be in charge of rendering dynamically.

DoPIDOM [Bea06] is a framework for developing DOM-based components able to manipulate an SVG document at same time the SVG document is rendered. It is based on Batik [Apa], and organizes DOM components in two categories: query consumer for navigating the SVG document and action producer for altering the SVG document.

2.3 Conclusions

LDE is a set of model-based techniques to software development. Main artefacts of a software development process are models, not only used for documentation purposes, but also as development artifacts. The process is stepwise, each step consisting in adapting result of the previous step (a model) with details of the deployment on execution platforms, following the examples of RUP [Kru03] and B [Abr96]. Input and output models are described by metamodels, which represent the abstract syntax of the modeling language, and are related by model transformations, which help in the improvement. First step is to describe the desired system into a model, and last step consists in generating code to be deployed in an execution platform. Beside clear advantage of the improvement process (e.g. adaptation to changes, traceability, etc.), the LDE model-based approach promotes to use the right (modeling) language for the right purpose, thus avoiding to use a too large or inappropriate language, without that necessity to stick to only one general purpose language for the whole process.

Thus, to define such a model-based process, methodologists need to clearly specify those languages, and developers need to correctly understand them. Moreover, to scale, such approach must be supported by comprehensive modeling tools. Drawback is that those tools require a lot of energy in development, and are only designed for a limited community.

To correctly specify a language, one need to define its abstract syntax, its semantics, and its concrete syntax. Abstract syntax may be given by mean of a metamodel, which allows to build model repositories (responsible for model interchange), which are the pillars for those bridges that are model transformations. Some solutions for semantics are already around (e.g. using set theory, or model transformations), though intensive research is still on

the way [HR04]. However, we found that concrete syntax definition still lacks maturity in the LDE community.

Regarding graphical concrete syntax, techniques and tools are already around, as graph grammars. Some has even ported results of this technological space to the LDE community (e.g. [EEHT05]). Moreover, model-based technology evolved rapidly last years (especially since the start of this thesis). Nevertheless, we feel that these solutions, if they at nice at tooling, still lacks universality and are too imperative to serve as specifications.

Regarding textual concrete syntax, the situation is rather worse and has not gained the attention it deserves. Indeed, one need to specify consistent code generation and text processors using poorly specialized tools. However, there exist solutions in another technological space, namely compiler compilers, that may help in finding a comprehensive specification to concrete syntax.

In this thesis, we propose solutions for graphical and textual concrete syntaxes that we feel could be used as specifications.

In next chapter, we continue our presentation of LDE with an example of an LDE process definition and application. Another example is given in section A.2 on page 134.

Chapter 3:
Netsilon: LDE Example and Embryonic Solution

Netsilon is a web application modeler and generator. Construction of the tool has followed the LDE principles to support an LDE methodology for web application engineering. In addition to reusing a subset of UML, it integrates notably two new DSLs, one graphical (the Hypertext Model) and another one textual (Xion). We developed Netsilon in a small company, ObjeXion Software, between 2001 and 2002, using technologies of that time: model repositories are hand-coded using the java language and model transformations are java programs. After introducing general remarks about web applications in section 3.1, we give general principles of Netsilon in section 3.2 and implementation insights in section 3.3 as an example for language engineering. We also show that Netsilon may be seen as a generic code generator able to produce some text from a model. An example, developed in section 3.4, shows generation of text following rules of the SVG language to graphically depict a simple class diagram model. However, Netsilon cannot be seen as a complete solution as it is specialized in generating text representations: it is not possible to reverse the process (from text to model), and graphical syntaxes are not natively handled. Nevertheless, experience presented here is at the root of solutions developed in next chapters.

> *Section 3.1, 3.2 and 3.3 of this chapter are inspired from [MSFB05] as published in the journal of Software and System Modeling (SoSyM) in 2005.*

3.1 Web Applications

A web application is an information system which supports user-interaction through web based interfaces. Typical web applications feature data persistence, transaction support and dynamic web page composition.

A web application is split into a *client-side* part, which is running in a web browser, and a *server-side* part, which is running on a web server. The client-side is responsible for page rendering while the server-side is responsible for business process execution and web page construction. The process of page construction varies widely in dynamicity, ranging from completely static, in the case of predefined web pages, to totally dynamically constructed pages (which vary in terms of content, presentation and navigation), when the web pages are the result of some computation on the server: programs (that can be written in languages like PHP, JSP, Java, or ASP.NET) integrate into web pages the graphic elements

and the information coming from many kinds of data sources (like databases, files, sessions or context variables…).

3.2 LDE for Web Application Engineering

Netsilon is a web application modeler independent from deployment platforms: one can describe a complete dynamic web application at the analysis level without integrating any detail depending on the application server or the database server. The architecture of Netsilon is given in figure 3.1.

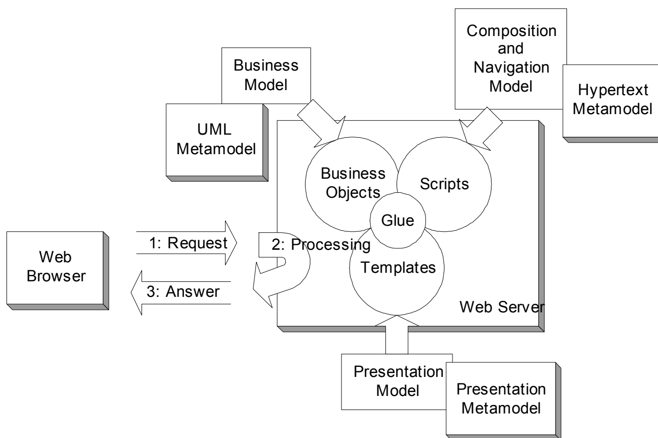


Figure 3.1: Web Application Modeling with Netsilon

Once the requirements have been gathered (via techniques such as use-cases and activity diagrams), the web application is modeled from three points of view, the *Business Model*, the *Hypertext Model*, and the *Presentation Model*. These models are independent from the web platform (i.e. application server and data persistency technique); they capture a comprehensive description of the web application. With the help of *Xion*, an action language designed to query and manipulate the business model, they contain enough information to drive the generation of the final web application. In addition to these three models, a *Deployment Model* gives information about the target platform, i.e. the kind of application server (PHP, JSP or Java), the kind of data repository (relational database), and information like the URL of the web application, or password to access database. Models and their relations are given in figure 3.2.

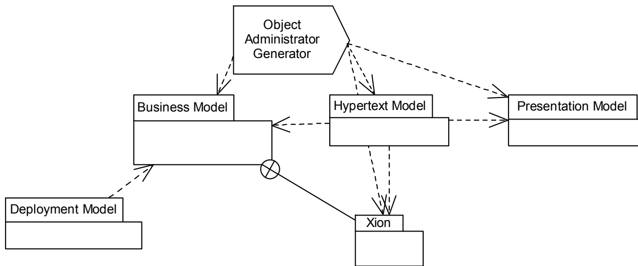


Figure 3.2: Netsilon Set of Models

We continue the section by describing in more details the business model, the hypertext model and the presentation model. We also present the Xion action language.

3.2.1 Business Model

The business model describes the organization of the business concepts managed by the web application. UML class diagrams are used to represent business classes, their attributes, operations and relations. The implementation of methods is specified with the action language Xion.

The business model is used as an input of the model-driven process which generates the business layer of the web application, using guidelines described in the literature [MVC03]. Object persistence is provided by a relational database, whose schema is derived from the business model. The object-to-relational mapping is designed so that incremental modifications to the business model have as little impact as possible on existing information in the database; we talk of M0 preserving transformations. The relational database is completely abstracted away. The business model is a subset of UML class diagrams in that it does not integrate multiple inheritance, n-ary associations, or derived attributes. Concrete syntax is that one of UML class diagrams. The classes from the business model are all persistent by default, the code is completely generated.

The advantage is that the designer does not have to care about persistence, the process of creating and updating the database schema is completely automated, and implementation classes for the business model are generated in the web platform target language (either Java or PHP).

An example of business model is shown in figure 3.3. Designed web application is a simple genealogy store. `Persons` may be married and have two parents. Operation `Person:marry` is implemented in Xion and creates the corresponding `Marriage` object, if `gender` is actually different between the people to marry.

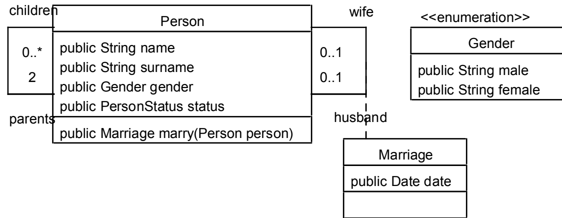


Figure 3.3: An Example of Business Model

Netsilon features an *Object Administrator* generator. The object administrator is a generic web application for the purpose of administrating the objects stored in the database. Data are organized in objects according to the business model, and objects can be created and deleted, their attributes can be updated, links can be created between objects, and operations can be executed. This administration tool is very helpful, and is used both for initialization of the object database and for data maintenance purposes. The generator is actually a model transformation that creates a specific presentation and hypertext model from the business model. Such derivation is achievable since the user interface of the object administrator does not have to be customized, and can therefore follow simple user interaction patterns.

3.2.2 Hypertext Model

The second model, the hypertext model, is an abstract description of composition and navigation between document elements and fragments of document elements. In the context of web modeling this model describes how web pages are built (*composition*) and linked (*navigation*). Hypertext model makes use of a completely new graphical language.

Composition describes the way the various web pages are composed from other pages (fragments) as well as the inclusion of information coming from the business model or current context (like sessions). Again, Xion is used as a query language to extract information from the business model and as a constraint language to express the various rules, which govern page composition.

Navigation details the hyperlink between the web pages, including the specification of the parameters that will be transferred from page to page, as well as the ability to specify actions to be applied when triggering a link (typically to execute a method defined in the business model). The Xion language allows the specification of the actions to be performed when transitioning from one page to another and the declaration of predicates that lead to the selection of a particular path between pages according to the current context and the business model.

Concepts of composition and navigation are unified under the concept of *Decision Centers*. Regarding composition, we identified the simple *Composer*, which includes the result of a web page execution into another, and the *Iterative Composer*, which performs this work for each element of a collection computed from a Xion expression. *Value Displayers* make possible to integrate a value computed from a Xion expression into a web file. Regarding navigation, we identified *Links*, which compute an URL to reach another web file, and *Forms*, which are links that gather data from HTML forms. In addition to this, web files may need some parameters that pass through decision centers thanks to Xion expressions.

The hypertext model makes it possible for a tool to check the coherence and the correctness of the navigation at model code generation time. This removes all the troubles related to parameter passing and implementation language boundary crossing (mix of interpreted languages, un-interpreted strings, parameter passing conventions…) often encountered when programming web applications by hand.

As an example, an excerpt of the hypertext model for our genealogy application server is given in figure 3.4. A web page `peopleList` is composed of the list of people stored in the genealogy web application. This is realized by an iterative composer (`peopleListDisplay`) which will integrate another web page (`personInList`) for each person that will be found according to a Xion expression (not shown here). This latter web page displays some information about the iterated person passed as parameter by `peopleListDisplay`. This information is displayed thanks to value composers like `personIsMarried` or `personName`, which have knowledge of the value to display thanks to some Xion expressions. `personInList` also offers a link to another web page (`person`) to display more information on a given person. The `peopleListHead` and `peopleListFoot` web files (and composers) open and close the HTML table in which are represented the people. An example for `peopleList` applied in such a genealogy web application is given in figure 3.5.

3.2.3 Presentation Model

The third model, the presentation model, contains the details of the graphic appearance of web applications. We do not want to restrict the field of possible graphic designs. We want to be able to generate any kind of web user interface.

A dynamic web application is composed of *Fragments*, which can be developed as static HTML, supplemented with some special placeholders, easily identifiable by graphic designers and HTML integrators. Whenever some dynamic information must be inserted into a web page, the graphic designers simply designate the spot in the file where this information must be inserted: compositions and links as defined by the hypertext model as given in figure 3.5 are placed using tags following the pattern `!-!/objexion/<identifier> <name>/`. The presentation model is not limited to HTML and can be

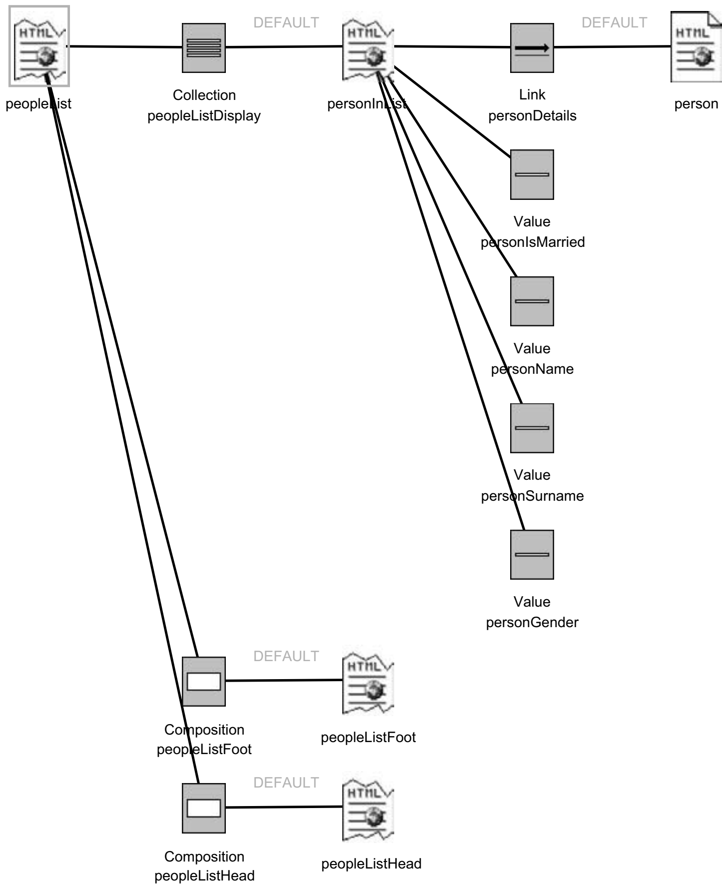
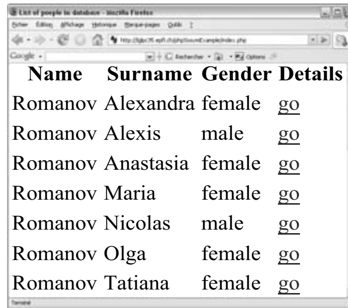


Figure 3.4: Hypertext Model for Listing People

used to manage any other textual formalism like Javascript, XML or SVG. An example of hypertext model for the `personInList` web file is given in figure 3.6. In the example, the pattern follows the HTML rules, thus the generated web application will follow the HTML rules.



Name	Surname	Gender	Details
Romanov	Alexandra	female	go
Romanov	Alexis	male	go
Romanov	Anastasia	female	go
Romanov	Maria	female	go
Romanov	Nicolas	male	go
Romanov	Olga	female	go
Romanov	Tatiana	female	go

Figure 3.5: Genealogy Web Application at Work

```
<tr>
<td>!-!/objexion/5 personName/</td>
<td>!-!/objexion/6 personSurname/</td>
<td>!-!/objexion/22 personGender/</td>
<!--td>!--!/objexion/7 personIsMarried/</td-->
<td><a href="!-!/objexion/8 personDetails/">go</a></td>
</tr>
```

Figure 3.6: Hypertext Model for `personInList`

3.2.4 Xion

The behavior of dynamic web applications depends on the overall state of the system. This state is stored for part in databases (which implement persistence of the business model) and files (HTML templates or cookies for instance), but also in volatile memory (data of the currently active sessions and parameters). The Xion language was designed to provide uniform access to all kinds of state in the system, and is used to describe queries, constraints and actions. A typical example would be to have the possibility to look for a specific set of objects according to some constraints, like retrieving a customer cart from a character string carried by a web page parameter or contained by a cookie.

Xion has to provide support to query the business model and to express methods and state changes. OCL is a very good candidate for querying instances of the business model however (because of its side-effect free nature) it is not well adapted for defining method bodies or any change in the model that the hypertext model may need to define. Some approaches known as declarative, like the B predicates [Abr96] or the Alloy language [Jac02], support the definition of side-effects procedures by constraints defining the state

before and after the procedure has proceeded. We did not choose that solution because generating efficient code out of a declarative specification is still an open issue [MTAL98], and because most software developers are more used to the imperative approach, like the action semantics, with a well-defined sequence of actions to perform.

In order to define Xion, we decided to extend the OCL query expressions to define a new imperative language for our actions and queries needs. This means to add side-effects capabilities to OCL, and to provide imperative constructs, like blocks and control flows. In the context of the business model, supporting side-effects means:

- Create and delete an object,
- Change an attribute value,
- Create and delete links,
- Change a variable value,
- Call non-query operations.

It was also necessary to remove some constructs of the OCL, which are out of the scope of the approach:

- Context declaration, only useful for defining constraints,
- `@pre` operator and message management, only meaningful in the context of an operation post-condition,
- State machine querying, as there is no equivalent concept in the web architecture Netsilon proposes.

Since most web application developers are already familiar with the Java language, we re-used part of its concrete syntax. Constructs we took from Java are:

- Instruction blocks, i.e. sequences of expressions,
- Control flow (`if`, `while`, `do`, `for`),
- `return` statement for exiting an operation possibly sending a value,
- `super` call of ancestor constructor.

Moreover, for Xion to look like Java as much as possible, we decided to keep Java variable declaration, and operators (==, `!=`, +=, >>, `?` ternary operator, etc.) rather than those defined by OCL. The standard OCL library was also slightly extended, by adding the `Double`, `Float`, `Long`, `Int`, `Short` and `Byte` primitive types, whose size is clearly defined unlike the OCL `Integer` or `Real`. As web application often deals with time, we also added `Date` and `Time` predefined types.

Considering the business model shown in figure 3.3, an example for the Xion language regarding the `marry` method implementation is provided in figure 3.7. As one can see here, Xion looks like Java with `if/else` control blocks, the `null` value and the `return` statement. Notice that `this` and `self` can be used indifferently. We can also see that enumeration literals are treated as OCL 1.3 prescribes, starting with a `#` sign.

```
//person is parameter of the operation
Marriage ret = null;
if (this.gender == person.gender) {
  ret = null;
} else {
  ret = new Marriage();
  ret.date = Date.getCurrent();
  Person wife, husband;
  if (self.gender == #female) {
    wife = this;
husband = person;
  } else {
    wife = person;
    husband = this;
  }
  ret.wife = wife;
  ret.husband = husband;
}
return ret;
```

Figure 3.7: Implementing `Person::marry` in Xion

Another example is provided in figure 3.8. Here, we can better feel the OCL affiliation of the language. This example is a parameter transmitted to the `peopleList` web file of the hypertext model of figure 3.4. `peopleList` is in charge of displaying the given list of `Person`. This fragment web file is integrated into a calling web file by means of a composer decision center. The purpose, here, is to display sisters of a given person, represented by the `person` variable. First we navigate the `parents` association end. This navigation returns the set of `Person` instances representing the parents of the `Person` instance. To get all children of these parents, we then navigate from the obtained instances through the `children` association end. As a consequence, the `Person` instance and his/her siblings, with the same father and the same mother, will appear twice in the resulting collection. Half siblings will only appear once for that they have only one common parent. As we are not interested in making a difference between sisters and half sisters, we remove duplicate instances with the `asSet` predefined operation. To remove `person` from this collection, we use the `excluding` predefined operation. In this list, we only want sisters of `person`; this can be done by selecting only instances which are declared to have a female literal value in their `gender` slot. This is achieved by the `select` predefined operation. Finally, we want the obtained collection to be ordered by `name`. This is achieved by another predefined operation `sortedBy`.

```
person.parents.children->asSet()->excluding(person)
   ->select(p : p.gender == #female)->sortedBy(p : p.name)
```

Figure 3.8: Sisters found by a Xion Expression

3.3 Implementation Insights

Our goal is to achieve total code generation from models, while making no restrictions on the visual appearance of the web application. Therefore, we defined novel modeling constructs, for the description of the composition of web pages from various sources of information, and for the specification of the navigation between pages. These models are precise, abstract and independent from the target platform. They are later transformed into executable code which runs on the web platform. The new modeling elements are packaged in a new metamodel.

3.3.1 Metamodel

As the business model is a clone of UML class diagrams, we merely reused the UML metamodel that describes class diagrams. The presentation model is nothing but a set of textual files, and does not deserve a metamodel since file systems natively handle such data. On the contrary, the hypertext metamodel is a completely new DSL that needs to be described from scratch, and we decided to use metamodeling techniques to state its concepts, i.e. its abstract syntax. The metamodel is presented in figure 3.9.

The central element is `WebFile`, which describes a document element (or a fragment). A `WebFile` can be treated as a static or a dynamic element: if static, it is simply copied unmodified during deployment; if dynamic, it is generated as server-side code and participates in the dynamic part of the web application. HTML tag filtering and striping makes it possible to use web-authoring tools for the design of fragments as easily as for the design of entire pages.

A `Zone` is a specialized `WebFile` for which associated content is obtained dynamically by the evaluation of an expression that returns either the URL of the content or the content itself.

Another specialization is `PolymorphicZone` as a mean to introduce the notion of polymorphism in the hypertext model. A `PolymorphicZone` is associated to an operation defined in a class (of the business model) which is in charge of producing the content. Since the operation can be implemented by overridden methods in the subclasses, the content can be generated according to the real class of an instance. To reinforce the separation between the business model and the hypertext model, we introduce a subclass of `Method` named

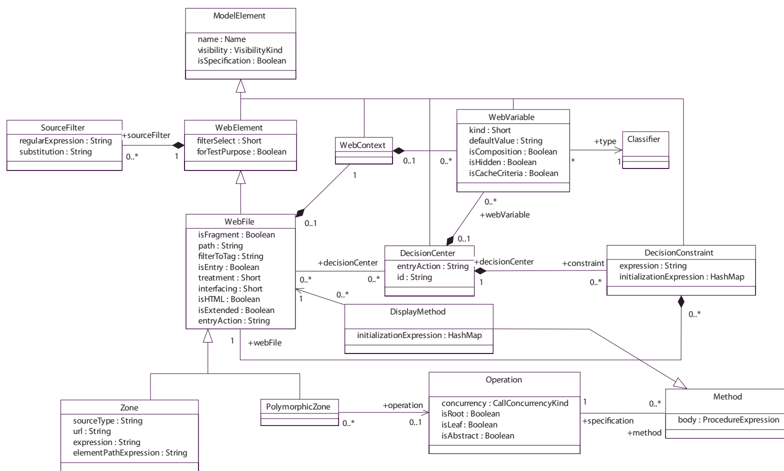


Figure 3.9: Hypertext Metamodel (Excerpt)

DisplayMethod that is associated to a WebFile. The production of content by an operation implemented by one or more display methods is thus realized by web files. At runtime, a polymorphic zone is replaced by the web page resulting from invocation of the display method.

Each WebFile has a context WebContext that is placeholder for entry parameters. A parameter is described by the WebVariable metaclass and has a type, which is an instance of a Classifier found in the business model.

DecisionCenter define variation points in the hypertext model. A decision center has an entryAction written in Xion, a unique identifier to identify its placeholder in the presentation model, local variables (WebVariable) and an ordered sequence of DecisionConstraint. A decision constraint defines a guard whose evaluation to true leads to the selection of its associated WebFile.

Metamodel for deployment model is given in figure 3.10. The Site metaclass is the main container of the complete deployment information. It can define several DeploymentSite, which conform either to JSPDeploymentSite, ServletDeploymentSite, or PHPDeploymentSite, for defining which application server is to be targeted, and where to export generated code. Each deployment site uses a certain number of databases for storing business information. The chosen platform for generation is referenced in the site by the currentDeploymentSite association end. Deploying the same web application on another platform is done by changing the DeploymentSite referenced in the current-

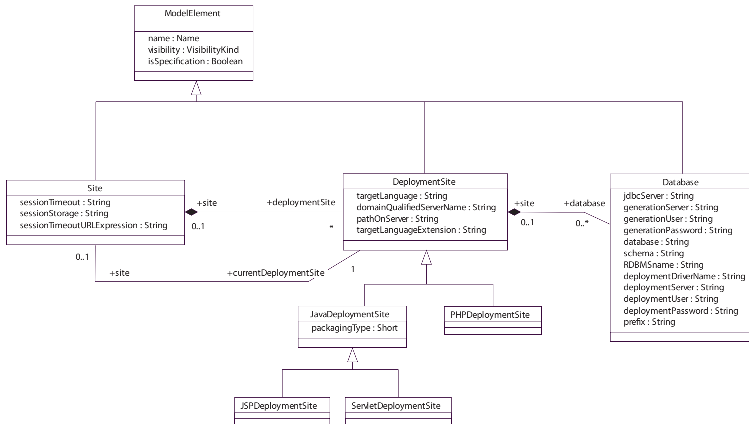


Figure 3.10: Deployment Metamodel (Excerpt)

`DeploymentSite` association end. Creating a new deployment platform is done by providing information about the new platform, without changing anything in the web application description.

Xion also defines a metamodel to state its abstract syntax, but we will avoid its description here for sake of brevity. In order to implement Netsilon, we modified the ArgoUML UML modeler [BRTK]. In this program, metamodels are encoded by Java classes, and representations are programmatically prescribed with help of a generic 2D graphical API. We followed that precise technique to encode abstract syntax and representation for hypertext model. Xion was completely described using ANTLR [Par05]: abstract syntax was described by an ANTLR tree grammar, and concrete syntax by an ANTLR text grammar. Obviously, coherence rules (e.g. type checking for Xion) was completely handcoded using Java or ANTLR. The same remark holds for model transformations and code generation described in next section.

3.3.2 Improvements and Code Generation

Target platform have both static (i.e. data persistency) and dynamic (i.e. behavior) dimensions. Target technologies may be an Oracle, a MySQL or a PostgreSQL relational database regarding static aspects and PHP, JSP or Java Servlet regarding dynamic aspects. As a consequence, the possible number of different deployment platforms is the Cartesian product of the supported databases and application servers. Code generation is performed using a two-step process by using intermediate languages.

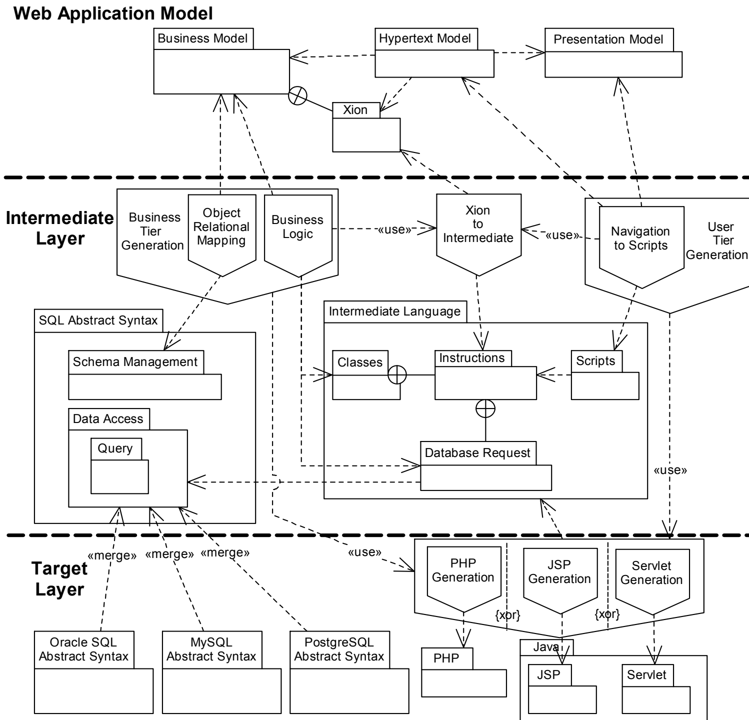


Figure 3.11: Code Generation Process

The intermediate layer is the combination of an `SQL Abstract Syntax` and a generic application server script, merely called `Intermediate Language`. The SQL abstract syntax is a subset of the SQL 92 language, supposed to be managed by most RDBMS. This syntax allows managing database schemas (`Schema Management`), for creating, deleting, or modifying table schemas. This SQL abstract syntax also allows data access to the recordsets (`Data Access`), for instance by `Select` queries (`Query`). The `Intermediate Language` is an abstraction of the concepts of scripts that application servers handle and that code generation can finally target. The `Classes` can define structure of web application classes. Behavior is expressed by `Instructions`, possibly database manipulations (`DatabaseRequest`), depending on the above-mentioned SQL `Data Access`. `Scripts` are specialized in integrating pieces of server-side behavior in files to be sent to the client, expressed with `Instructions`.

Since business information is to be stored in a database, a transformation `Object Relational Mapping` creates a database schema from the `Business Model`. In case a database schema already exists, it will be altered by the transformation as necessary. The business information, to ease integration and reuse, is encapsulated into proxy server-side classes by the `Business Logic` transformation. Behavior in the `Business Model`, described in `Xion` statements as bodies of constructors and methods, are also translated by this `Business Logic` transformation inside corresponding server-side classes, taking advantage of the `Xion to Intermediate` transformation. The `Presentation Model` and the depending `Hypertext Model` are compiled together by the `Navigation to Scripts` transformation to produce `Scripts` of the `Intermediate Language`.

Target layer is generated from the intermediate layer through a transformation selected from information contained in the deployment model (`PHP Generation`, `JSP Generation` or `Servlet Generation`). Target is actually some text, even though represented as a model here. The correct SQL model (specialized SQL for `Oracle`, `MySQL` or `PostgreSQL`) is transparently selected by the `Object to Relational Mapping` and `Xion to Intermediate` transformations through a higher-order hierarchy mechanism, implemented by a factory design pattern.

3.4 Experiencing SVG Model Representation

Netsilon is basically a web application modeler and generator, but may be seen from various viewpoints. One of these possible viewpoints is the following. The business model is some kind of model description facility, i.e. it may be used as a metamodeling language. By generating object administrator and deploying the web application, one may edit a model, so called the abstract syntax graph, in a rapid prototyping fashion. The fact that the deployed web application is actually another modeling tool is merely a technical detail. The presentation model may be described following rules of any textual representation as soon as it complies with the template-based hypertext model. Netsilon also offers a mean to represent the model in a textual form by text templates. As a resume, a model may be edited using the object administrator and textually represented according to a template-based description.

In this part, we experience such an architecture in which we targeted the SVG textual language for 2D vector graphics in order to depict an abstract syntax graph. Indeed, a textual language such as SVG may represent graphics, so Netsilon may be used to generate diagrams. We will illustrate our approach by an example, that is a simplification of the UML class diagrams. A similar experiment succeeded in defining the chess language (as presented in section 5.2.2 on page 87) and related user interactions making a chess game available for playing online.

3.4.1 Simplified UML Class Diagrams

UML class diagrams are the best known part of UML. They are designed to describe class systems of object-oriented applications. We show a slightly customized excerpt of the metamodel of UML 1.4 class diagrams in figure 3.12. Difference is that GeneralizableElement inherits Namespace while in the original metamodel Classifier inherits both GeneralizableElement and Namespace. We had to alter this fact since Netsilon does not support multiple inheritance. In addition, Feature is given an order attribute to simulate the {ordered} constraint in UML metamodel.

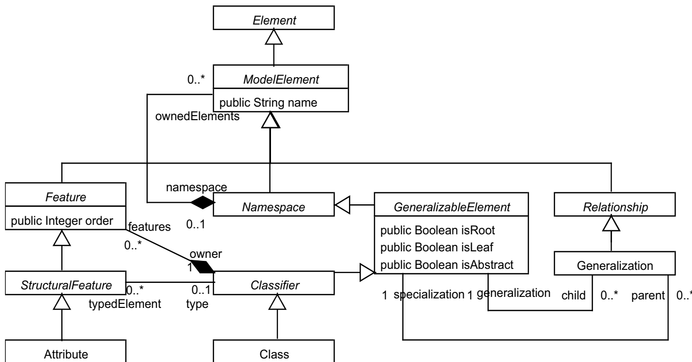


Figure 3.12: Simplified UML Class Diagrams Metamodel

To have a brief description, a Class is an element with a name that may contain some Attributes, and that may be integrated into a Generalization hierarchy. An Attribute has a name and a type. The interested reader may refer to the UML standard to have complete description of UML class diagrams [AAB+07]. It is interesting to see that the UML abstract syntax is easily representable by UML class diagrams. This is this property that makes reasonable to represent metamodels with Netsilon business models. This property is also the root idea in the definition of the MOF metamodeling language [ACC+06].

Regarding graphical depicting of elements in a diagram, a Class is represented by rectangle, in which the name of the Class is shown. That rectangle can be of any size and can appear anywhere in a class diagram. Moreover, different representations of the same class may exist together in the same class diagram. A Class representation may depict some of the Attributes owned by the Class in a line-separated part at the bottom of the representation rectangle. An Attribute is shown as a text using the following template: <attribute_name> : <attribute_type_name> (note that this rule is not applied for representing the business model, for instance as shown in figure 3.12). A Generali-

zation is depicted by an arrow between a representation of each one of the connected classes. The arrow is oriented to the representation of the generalization. The arrow can follow any route between its ends, provided that the arrow ends touch the borders of the representations of the connected elements.

Neither absolute position of representation for elements, nor the route followed by representation arrows of Generalizations are important in terms of abstract syntax. Important points are the position of an Attribute representation (within a representation of the holding Classifier), and that the arrow depicting a Generalization actually connects representations for the associated Classes. In this, the simplified UML class diagrams enter the category of connection-based languages[1].

3.4.2 Representation Using Netsilon

Previous section ended with a description of the representation rules for the simplified UML class diagrams. The description was given in plain English. Here, we provide this description in terms of Netsilon business, hypertext and presentation models.

3.4.2.1 Representation Framework

Representations require information that is not given by the metamodel for abstract syntax. For instance, abstract syntax does not state the number of representations, or where representation are in the diagram. This requires to enhance the business model. We show in figure 3.13 an abstract framework to handle representation data.

Graphics is an abstract class that declares two abstract operations: display and toggleChange. display should be implemented by display methods that must return the web file supposed to provide concrete representation for the Graphics object. toggleChange is an operation that should be called anytime the represented element or its representation (i.e. a Class instance or its representation) is modified (e.g. by changing its name or the size of its representation).

Diagram is a concrete realization of Graphics for diagrams. A diagram is a representation container. Note that many diagrams may exist together to represent a same model, but a representation cannot cross the boundaries of a diagram. A Diagram has a name, and maintains a timestamp (thanks to the lastChangeD and lastChangeT attributes) that are updated each time a toggleChange is called. The interest is to know when model or representation has changed and when a diagram representation (as shown in a web browser) should be regenerated.

RepresentationElement is the abstract class for any representation element, and should be further specialized by concrete representation classes. It declares abstract opera-

1. This example is a simplification for UML class diagrams, and does not define graphical class composition, which makes UML class diagrams enter the category of hybrid languages (see section 2.2.2 on page 27).

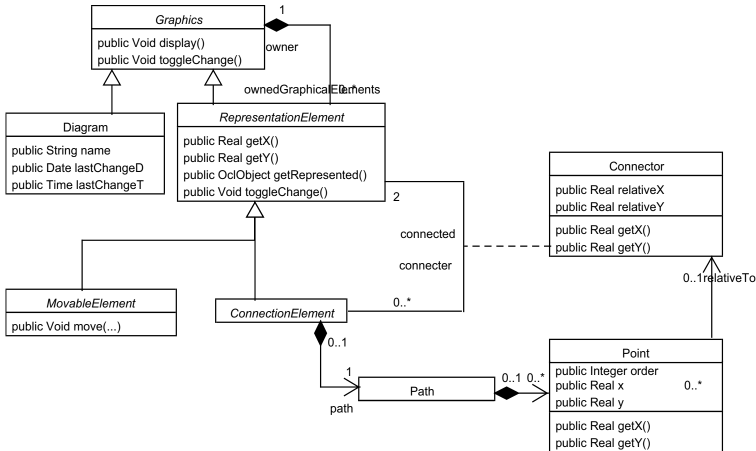


Figure 3.13: Diagramming Framework

tions able to provide concrete location of the representation element (`getX` and `getY`). `getRepresented` is another abstract operation that should return the represented model element (a concept instance in the abstract syntax graph). A `RepresentationElement` is owned either by another `RepresentationElement`, or by a `Diagram`. If a `RepresentationElement` instance or its represented element in the abstract syntax graph is changed, the `toggleChange` operation must be called; message is transmitted to the owner `Graphics` instance, be it another `RepresentationElement` instance or a `Diagram` instance, so that the information is propagated up to the owning diagram.

`Connector` is a class that may be used for connection-based languages. It connects two `RepresentationElements` at a given place relative to the origin of the connected element, so that if the connected element moves, a `Connector` instance does not need to be updated in terms of representation data, as held by the `relativeX` and `relativeY` attributes. `getX` and `getY` are operations that return the absolute place of the connection, according to the `relativeX` and `relativeY` attributes, and the actual position of the `connected` element.

`MovableElement` is a specialization of `RepresentationElement`. It declares an abstract operation that should be called whenever the representation object is moved on the scene (i.e. when the user drags a representation on the diagram scene). `ConnectionElement` is another specialization that encapsulates the connection concern in connection-based languages. A `ConnectionElement` owns a `Path` of ordered `Points`, which are at

a given position, be it absolute or relative to a Connector. The Path represents the route of the connection and states what are the intermediate points between Connectors.

Framework is extended to hypertext model. A generic web file written in HTML, index, accepts (as a diagram parameter) the Diagram to show. If not provided, the web page proposes the list of registered Diagrams in the system. Each one of these diagrams is shown as an hyperlink, which calls back the index web file (through a link decision center), this time with the diagram parameter filled. In this latter case, the index web file embeds the diagram web file. Moreover, thanks to Javascript code and a Java applet, the index web file regularly calls a requestReload web file, which returns “true” or “false” (thanks to a value displayer) depending on whether the represented diagram has changed or not since last load of the diagram web file. In other words, it returns “true” if a toggleChange was called on the diagram since the diagram was loaded. In case the response is “true”, index reloads the diagram web file.

The diagram web file renders a Diagram provided as mandatory parameter, using the SVG textual language. It is the implementation web file for the Diagram::display display method. The web file integrates (through a collection displayer) each RepresentationElement owned by the diagram by calling their display operation. Remember that the display operation returns the web file able to display the RepresentationElement on which it is invoked. In addition, diagram integrates Javascript code that handle “drag and drop” for those SVG nodes that declare .movable. in a class XML attribute. Once the SVG node is dropped, script sends in background a request to the move web file with the moved element and the coordinates of the move as parameter. Consequence is that the move method is invoked on the sent MovableElement with the new coordinate as argument.

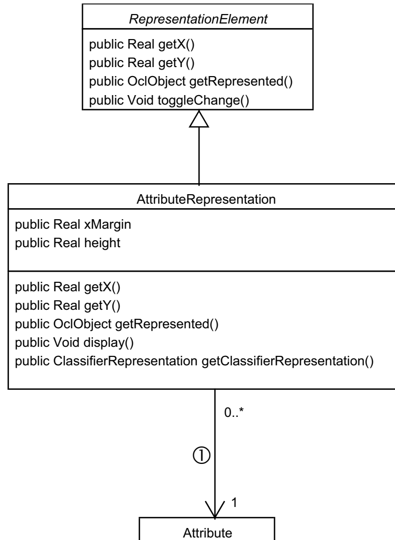
display is a polymorphic web file associated to the Graphics::display operation: the resulting web file (e.g. diagram in case of a Diagram) constitutes the content of the “virtual” display web file.

Finally, line is a web file that is iteratively composed by the path collection displayer to represent a Path, as found in a parameter of the decision center. line is merely an SVG line that takes as parameter the coordinates of the end points, and the color that should be used for the drawing.

3.4.2.2 Business Model: Concrete Representations

We present here the specialization of the RepresentationElement of the framework for each representable class of the metamodel, namely Attribute, Classifier (as a generalization for Class), and Generalization.

The RepresentationElement specialization for Attribute is shown in figure 3.14. As expressed by an OCL constraint, an AttributeRepresentation should be owned by a ClassifierRepresentation, which is the representation element for

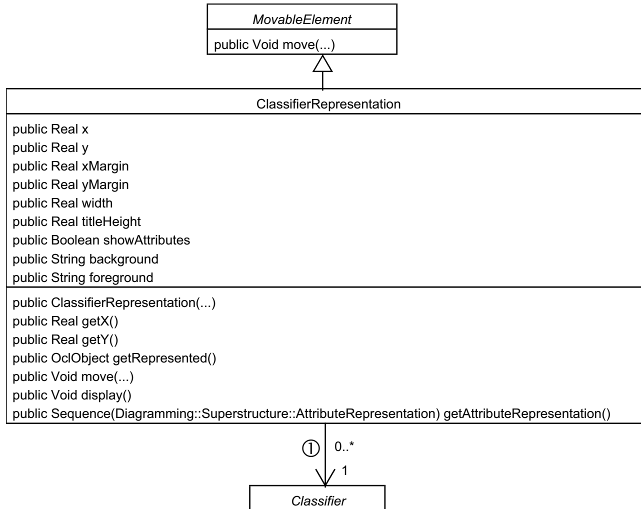



```
context AttributeRepresentation inv:
self.owner.oclIsKindOf(ClassifierRepresentation)
```

Figure 3.14: `Attribute` Representation Element

`Classes`. The represented object is referenced through an association ①. According to multiplicities of the association ends, the representation may only depict one `Attribute`. Its is also stated (through a `0..*` multiplicity) that an `Attribute` may be represented any number of times. There exist only two representation data held by attributes `xMargin` and `height`. `xMargin` expresses the margin to apply from the left-hand side before writing the text, and `height` the height of the text box. Those data make it possible to implement the abstract operations required by superclass regarding position (depending on the container that should be a class representation, found by the `getClassifierRepresentation` helper function) and represented model element (accessed through ①). The `display` display method will be discussed later in section 3.4.2.3.

The `RepresentationElement` specialization for `Classifier` is shown in figure 3.15. We decided to follow the UML specification by defining representation on `Classifier` rather than on `Class` for extensibility reasons: if a new kind of `Classifier` is added to the abstract syntax, it will automatically "inherit" representation. As the

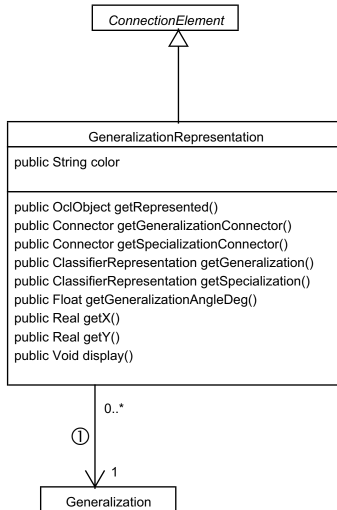



```
context ClassifierRepresentation inv:
self.owner.oclIsKindOf(Diagram)
```

Figure 3.15: `Classifier` Representation Element

representation for a `Classifier` may be moved freely, we make `ClassifierRepresentation` inherit `MovableElement` instead of directly inheriting `RepresentationElement`. An OCL constraint enforces that a `ClassifierRepresentation` may only be owned by a `Diagram`. Again, the represented model element is accessible through an association ① which states that a `Classifier` may be represented several times although a `ClassifierRepresentation` must represent only one `Classifier`. Data handled by the representation are the position (`x` and `y`), margins and height for the name compartment (`xMargin`, `yMargin` and `titleHeight`), the `width` of the rectangle, `background` and `foreground` colors, and the presence or not of an attribute compartment (`showAttributes`). This latter attribute is a representation design decision, as one could imagine computing whether `ClassifierRepresentation` owns some `AttributeRepresentation` or not. The `getAttributeRepresentation` helper function finds all `AttributeRepresentations` owned by this `ClassifierRepresentation`.

The `RepresentationElement` specialization for `Generalization` is shown in figure 3.16. As a `GeneralizationRepresentation` is a connection, it inherits the




```
context GeneralizationRepresentation
inv: self.owner.oclIsKindOf(Diagram)
inv: self.connected.getRepresented() =
        Set{ self.generalization.generalization,
             self.generalization.specialization}
```

Figure 3.16: `Generalization` Representation Element

`ConnectionElement` class. An OCL constraint states that `GeneralizationRepresentations` must be owned by a `Diagram`, and that connected elements should be the representations for those `GeneralizableElements` participating in the represented `Generalization`. A representation datum is relative to the `color` of the representing arrow, but one should not forget the route path and connection points that are also representation data. In addition to implementation of inherited abstract operations, it defines helper functions to find connectors and representations for `generalization` and `specialization` of the represented `Generalization`. Xion code for `getSpecializationConnector` is shown in figure 3.17. Code navigates to the `Connector` class through the `connected` association end. Among those `Connectors`, it selects that one whose `connected` element is the representation for the `specialization` of the represented `Generalization`. Note that Xion was designed before the `any` operation appeared in OCL.

```
return this.connector[connected]->select(i:
    ((ClassifierRepresentation)i.connected).classifier
     == this.generalization.specialization)->getOne();
```

Figure 3.17: `getSpecializationConnector` Xion Implementation

3.4.2.3 Hypertext and Presentation: Concrete Representations

If representation classes are defined to handle representation data, one still needs to implement the `display` operation by specifying a template web file responsible for SVG production.

```
<tspan
  id="!-!/objexion/463 RepresentationId/1"
  x="!-!/objexion/471 xMargin/2"
  dy="!-!/objexion/476 attributeHeight/3">
    !-!/objexion/472 AttributeName/4 :
    !-!/objexion/473 AttributeType/5
</tspan>
```

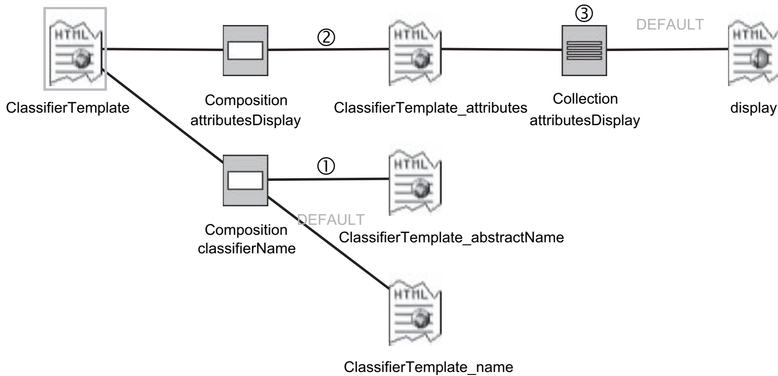
```
1.represented.getOID();
2.represented.xMargin;
3.represented.height;
4.represented.attribute.name;
5.represented.attribute.type;
```

Figure 3.18: `Attribute` Representation Presentation Model

SVG template for the `AttributeTemplate` web file, which implements the `AttributeRepresentation::display` display method, is shown in figure 3.18. The reference for each value displayer is shown in italic with a footnote that provides the Xion expression that the value displayer evaluates to. `represented` is a parameter of the web file, which is the `AttributeRepresentation` object that is sent the `display` display message. The template is a `tspan` SVG XML node. The node declares the attributes `id` that is filled with object identifier of `represented`, an attribute `x` that is filled with the `xMargin` representation data, and an attribute `dy` that is filled with the `height` representation data. The node, which is an SVG text part, defines a template for the text to be displayed:

```
<name of the type of the attribute to show> : <name of the attri
```

bute to show>, as given by Xion expressions navigating from the represented object.



① *represented.classifier.isAbstract*
② *represented.showAttributes*
③ *represented.getAttributeRepresentation();*

Figure 3.19: Classifier Representation Hypertext Model (Excerpt)

```
<g id="!-!/objexion/463 RepresentationId/" class=".movable."
  transform="translate( !-!/objexion/464 x/,
                        !-!/objexion/465 y/)">
    <rect width="!-!/objexion/466 width/"
          height="!-!/objexion/467 titleHeight/"
          fill="!-!/objexion/779 background/"
          stroke="!-!/objexion/775 foreground/"/>
    !-!/objexion/778 ClassifierName/
!-!/objexion/479 attributesDisplay/
</g>
```

Figure 3.20: Classifier Main Representation Presentation Model

An excerpt of the hypertext model for Classifiers is shown in figure 3.19. Value displayers are removed for sake of readability. Here, the hypertext model is slightly more important as representation concerns are split between different web files. Important Xion expressions to state decision constraints (① and ②) and collections (③) are also given in the figure. All those files are transmitted the represented web variable that holds the Clas-

sifierRepresentation object to depict. The main web file, i.e. the implementation of the ClassifierRepresentation::display display method, is ClassifierTemplate, for which presentation model is shown in figure 3.20. The template is an SVG group that takes as identifier that one of the representation object. The group is translated to the actual position of the representation. Moreover, it declares the .movable. class, which makes it possible to drag the group as drawn in an SVG renderer. As ClassifierRepresentation is a MovableElement, it will be possible to update representation data about position by mean of a drag. The group contains an SVG rectangle which follows prescriptions of representation data. The group also contains the classifierName and attributesDisplay decision centers shown in figure 3.19.

The ClassifierTemplate_abstractName web file is responsible for displaying the name of the represented classifier (in italic) in case the classifier is abstract as stated by expression ①. If not the case, the ClassifierTemplate_name web file represents the name in standard font. In case the representation object holds that representation data stating that a compartment for attributes should be depicted (expression ②), a composer includes the ClassifierTemplate_attributes web file. In addition to creating the rectangle compartment for depicting attributes, this latter web file iteratively calls the display display method of each one of the contained AttributeRepresentations (expression ③), i.e. the AttributeTemplate web file described above.

```
<g id="!-!/objexion/463 RepresentationId/">
  !-!/objexion/926 path/
  <g transform=" translate(!-!/objexion/1314 genGenX/,
                           !-!/objexion/1315 genGenY/)
                 rotate(!-!/objexion/1316 genGenRot/)">
    <polygon  fill="#FFFFFF" stroke="!-!/objexion/1317 color/"
              points="-10,5 0,0 -10,-5"/>
  </g>
</g>
```

Figure 3.21: Generalization Representation Presentation Model

Presentation model for Generalization is given in figure 3.21. It consists in an SVG group that contains a reference to the path predefined decision center (seen in section 3.4.2.1) in order to draw the path of the generalization. The group contains another group responsible for depicting the end arrow. That second group is placed and rotated according to some computation described in the business model by mean of the getX, getY and getAngleDeg methods of the GeneralizationRepresentation class.

3.4.2.4 Generating Representation

We provide here an overview of the system described above deployed as a web application. Once at run, the first action should be to enter an abstract syntax graph (i.e. a model) in the system using the object administrator. An example of such model is given in figure 3.22, as an instance of business model of figure 3.3.

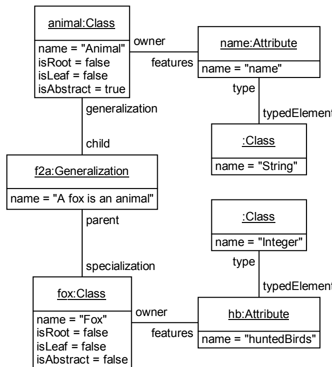


Figure 3.22: An Example Model for Figure 3.3

For that model to be represented, one needs to provide a representation model, as an instance of metamodel discussed in section 3.4.2.2. Again, this model may be entered in the system using the object administrator. An example of model to handle representation data is provided in figure 3.23.

Once defined the abstract syntax graph as in figure 3.22 and the representation data as in figure 3.23, the representation can be generated and rendered in an SVG-capable web browser. Application of the `index` main page offers the list of possible diagrams, that is the `CD1` diagram, as stated by the representation model. If one clicks on the hyperlink for `CD1`, the `index` web page is now applied again with that information it should embed representation for the `CD1` diagram, that is the `diagram` page. This latter page contains any representation for owned graphical elements, that are the `ar`, `fr` and `gr` representation objects.

The `ar` and `fr` representation objects are `ClassifierRepresentations`, so they will be displayed by two different application of the `ClassifierTemplate` web page. For the `fr` representation object, the expression ① of figure 3.19 returns false, and it is the `ClassifierTemplate_name` that will apply and represent the name of the represented `Class` (`fox`) in standard font. As the `fr` representation object indicates that attributes should be represented (expression ② is "true"), the `AttributeTemplate` web file is

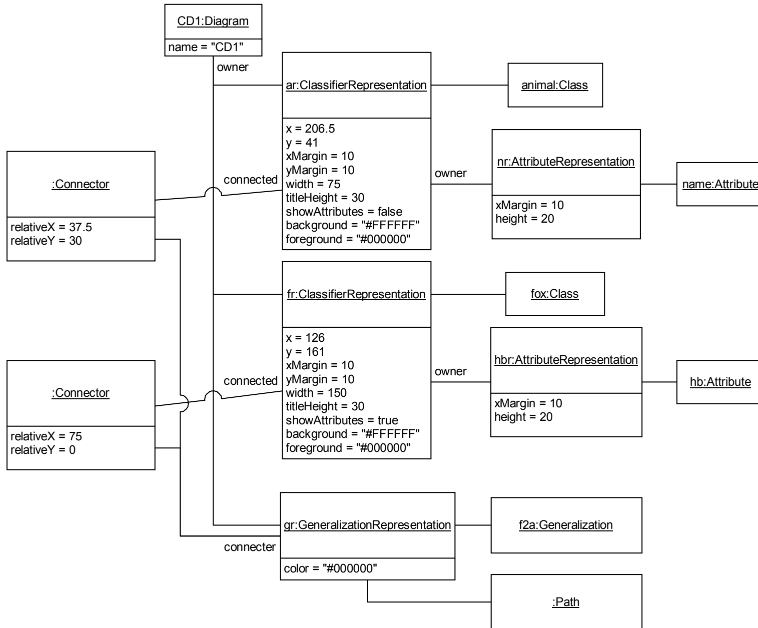


Figure 3.23: An Example Representation Model for Figure 3.22

applied for the contained `hbr` attribute representation object: name and type of the represented `hb` `Attribute` appears in a dedicated rectangle.

The `ar` representation object is also applied the `ClassifierTemplate` web page, but the result is different. On one hand, the represented `Class` (`animal`) is abstract, thus, according to expression ① of figure 3.19, name is shown in italic as prescribed by the `ClassifierTemplate_abstractName` template. On the other hand, despite the `ar` object owns an `nr` `AttributeRepresentation` object, the `showAttributes` representation data is set to false. This makes the expression ② return false and prevent representation for the second compartment (`ClassifierTemplate_attributes`) that would have owned representation for `nr`.

The `gr` representation object is merely given by applying the web file shown in figure 3.21. As there are no intermediate point, it merely consists in a line between representations for `ar` and `fr`. The final representation for this model is an SVG XML document that can be interpreted as the 2D vector graphics as in figure 3.24.

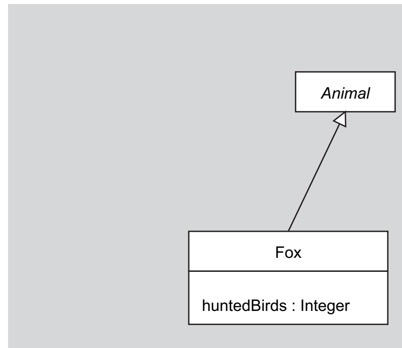


Figure 3.24: Graphical Representation for Figure 3.23

In case representation for a classifier is moved, e.g. Animal, the Javascript code responsible for the move calls in background the `move` web file. As a result, the `ar` representation object is sent the `move` message. `ar::x` and `ar::y` slots are updated and a `toggleChange` message is propagated up to the `CD1` representation object. At that precise moment, generalization representation has not moved yet on the SVG scene, so that the end of the representation arrow may point in vacuum. Since `index` regularly calls the `requiresUpdate` web file that now returns “true”, the complete diagram is regenerated and includes a generalization arrow that is now correctly placed.

3.5 Conclusion

Netsilon is an interesting example for three main reasons in the context of this thesis: it promotes an LDE methodology, it is built following metamodelling techniques to define new languages, and it is able generate a customized representation for a given model.

Beside the tool, Netsilon supports a clear LDE methodology regarding web application development following principles described in section 2.1.1 on page 11 and shown in figure 3.25. From system engineer point of view, there are four levels of abstraction: web application described in two tiers (business, hypertext), optional generation of an object administrator web application, graphical appearance (presentation, that is the third tier), and deployment directives. Modeling notations are defined by metamodels together with a template-based approach to textual output specification (i.e. presentation model). Target platforms are also described in terms of metamodels. Improvements may be defined by tooling (object administrator generation), or by enhancing the model (by specifying presentation).

Code generation and deployment are also handled by the tool in accordance with directives given by a deployment model. Netsilon even integrates some consistency checking at code generation time to enforce coherence of the model.

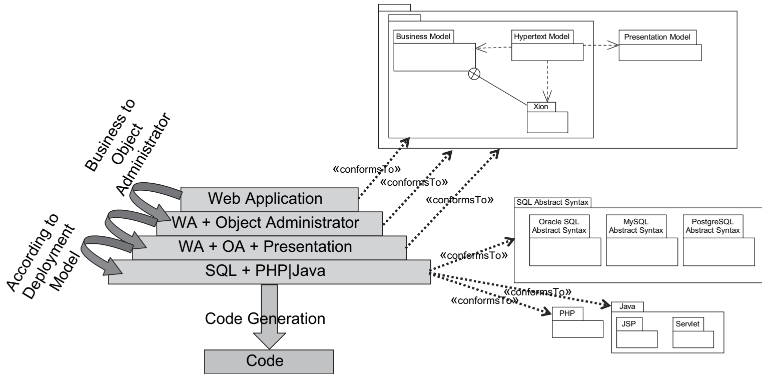


Figure 3.25: The Netsilon LDE Process

A second aspect of Netsilon is that it was built using a metamodeling philosophy. Although neither model repositories nor model transformations were available at that time, it defines some new DSL languages. To do so, it uses internally metamodels (described by Java classes, or ANTLR trees), and applies transformations (written in Java). Note that model transformations may be considered as a mean to specify semantics when it targets a language that defines semantics [KW03], for instance by targeting PHP (which has semantics through interpreter) or Java. Such architecture allowed a clear separation of internal Netsilon software components (with metamodels as interfaces), and facilitated communication among developers. However, it underlined some practical limitations of the LDE approach. Netsilon basically introduces a set of modeling languages, either textual or graphical, either build from scratch or as improvement of existing ones. Examples are the business model, which is a slight refinement of UML class diagrams, the hypertext model, which is a completely new graphical language, and Xion, which is a textual language mixing concrete syntax and semantics of OCL and Java. If ideas were new, the new languages introduced a relatively few number of really new concepts. Nevertheless, development required around 10 men-years and more that one hundred thousand lines of code. This poor productivity relative to purely technical novelties can be explained by a lack of tools and standards to apply model-based language engineering. If some techniques has evolved since then, especially regarding the definition and handling of abstract syntax (e.g. metamodeling languages, model transformation languages, metamodel merge, profiling and higher-order hierarchies [Ern03] - see appendix A), there still lacks some industry-quality mean and well

agreed standard to specify and use a *concrete syntax*, and to state and/or compose language semantics. The purpose of this thesis is to find specification techniques to cover both specification of textual and graphical concrete syntaxes. Moreover, we explore in appendix A some preliminary directions regarding reusability of semantically-rich elements (namely abstract syntaxes and model transformations).

An interesting result of Netsilon is that the business model may be used as a metamodeling language. If done so, the hypertext model (including that one generated for object administrator) may be compared to a representation specification. Moreover, an extension to the metamodel for abstract syntax is necessary to handle representation data. The deployed web application may be compared to a model representation engine. An advantage is that any metamodel and any textual representation (and representation language) may be used together: classical use of Netsilon prescribes HTML as target representation, but some other experiments showed that the model (i.e. the abstract syntax graph) may be represented in HUTN [MH05], and one may imagine some new transformation from business model to hypertext model that would generate an XMI representation instead of an object administrator. We experienced custom generation of SVG/DOM Javascript representations which showed that, with that price of adding new classes for handling representation data, one may depict and even alter a model. However, this approach is not a comprehensive mean to represent model as it is a generative approach only. Indeed, if one may "pretty-print" a model according to a given text structure and representation data, the process may not be reversed to construct a model from a textual specification. Situation is even worse regarding graphical representations: generated text must be a graphical specification that has to be regenerated in case any information is updated, be it about the model or the representation. We propose in chapter 4 a new approach, inspired from Netsilon concepts, in order to have a reversible textual representation for a model. A second new approach to graphical concrete syntax definition, based on SVG templates, will be developed in chapter 5.

Chapter 4:
Textual Concrete Syntax

Textual concrete syntaxes are traditionally expressed with rules, conforming to EBNF-like grammars, which can be processed by compiler compilers to generate text processors. Unfortunately, these generated text processors produce concrete syntax trees, leaving a gap with the abstract syntax defined by metamodels. This gap is usually filled by time consuming ad-hoc hand-coding. We have seen in previous chapter a mean to represent a model by using a model-based specification to organize text templates. However, this approach handles only one direction, from model to text. In this chapter we propose an improved kind of specification for concrete syntaxes that takes advantage of metamodels to generate tools (such as parsers or text generators) which directly manipulate abstract syntax trees. The principle is to map abstract syntaxes to concrete syntaxes via rules à la Netsilon that explain how to render an abstract concept into a given concrete syntax, and how to trigger other rules to handle the properties of the concepts. In contrary to Netsilon, rules are reversible thus permitting both analysis and synthesis of models from texts.

This work was presented at the MOLDELS conference held at Genova in 2006 [MFF+06], after some preliminary study available in the LGL-REPORT-2006-005 technical report of EPFL [FSGM06].

4.1 Introduction

While metamodeling is now well understood for the definition of abstract syntax, formal definition of concrete syntax is still a challenge, even though some authors consider concrete syntax definition as an important part of metamodeling [AK02].

Being able to parse a text and transform it into a model, or being able to generate text from a model are concerns on which more and more attention is paid in industry. For instance Microsoft with the DSL Tools [GSCK04] or Xactium with XMF Mosaic [CESW05] in the domain-specific language engineering community, are two industrial solutions for language engineering that involve specifications used for the generation of tools such as parsers and editors. A new OMG standard, MOF2Text [CIM+06], is also being developed regarding textual rendering of an abstract syntax graph.

Here is described an evolution of the previous chapter, in which we saw that a specification may formally describe how a model is to be represented. The philosophy is preserved, in that we used a metamodel-based approach, with composition and choice mechanism based on an action language. However, we redesigned the system to target a

slightly different goal, that is defining *reversible* textual representations, i.e. being able to generate text from model and to generate model from text. Moreover, we introduce concepts that are dedicated to this goal, and we remove concepts dedicated to web application modeling (e.g. link decision centers, and session management).

The work presented in this chapter was implemented by two tools. The fist one is named Sintaks (which stands for syntax in Breton, a Celtic language) and takes place in the context of the Kermeta project [Fle06]. Kermeta is an executable DSL (Domain Specific Language) for meta-modeling, which can be used to specify both abstract syntax and operational semantics of a language. Together, Kermeta and Sintaks provide a comprehensive platform for model-driven language engineering. The second tool, TCSSL tools, was developed at the French Atomic Commission (CEA). It provides a pretty printer, and generates a compiler compiler specification able to build a model from a text.

The chapter is organized as follows: after this introduction, section 4.2 presents our motivation and examines some related works, section 4.3 presents our metamodel for concrete syntax, and explains the mechanics which are behind. Section 4.4 presents two examples which illustrate the way concrete syntax can be modeled and associated to models of the abstract syntax. Section 4.5 provides an overview of the prototype implementations. Finally section 4.6 draws some general conclusions.

4.2 Motivations

We present here the context of the work, and we identify what are the requirements for the specification we propose.

4.2.1 Abstract Syntax versus Concrete Syntax

As previously said, defining a language can be decomposed into three related activities: defining the syntax, the semantic domain, and the mapping between syntax and semantic domain. D. Harel and B. Rumpe give a good introduction to the issues surrounding these activities in their paper about defining complex modeling languages [HR04]. In this part we focus on syntax definition; defining semantic domain and mapping syntax to semantic domain is out of the scope of the work presented in this thesis.

Syntax can be further decomposed into abstract syntax and concrete syntax. Abstract syntax describes the concepts of a given language independently of the source representation (concrete syntax) of that language and is primarily used by tools such as compilers for internal representation. Concrete syntax, also called surface syntax, provides a user friendly way of writing programs; it is the kind of syntax programmers are familiar with.

Object-oriented meta-modeling languages (such as EMOF, Ecore or Kermeta) can be used for representing abstract syntax; concepts of languages are then represented in terms of

classes and relations. A given program can be represented by a model conforming to the meta-model which represents the abstract syntax of the language used to write the program. Writing this program, in other words building the model which represents the program, requires some way to instantiate the concepts defined in the meta-model.

This can be achieved either at the abstract syntax level, or at the concrete syntax level. The difference is that in the first case, the user has to manipulate the concepts available in the meta-modeling environment (for instance via reflexive editors) while in the second case, the user may use a surface language which is made of those concepts. While the end-result is the same, it is much simpler for users (such as programmers) to write programs in terms of concrete syntax, rather than directly using instances of meta-modeling concepts.

For textual languages, there must be some way to link the information in the text (the concrete syntax) with the information in the model (the abstract syntax). We have seen in section 2.2.1 on page 26 that the issue of analyzing text to produce abstract syntax trees had already received much attention in the compiler community. Most notably, efficient tools are available to generate parsers from EBNF-like grammars. Unfortunately, these generated parsers produce concrete syntax trees, leaving a gap with the abstract syntax defined by meta-models (under the shape of graphs), and further ad-hoc hand-coding is required.

In the next sections we propose a new kind of specification for concrete syntaxes, which takes advantage of meta-models to generate fully operational tools (such as parsers or text generators). The principle is to map abstract syntaxes to concrete syntaxes via bidirectional mapping-models with support for both model-to-text, and text-to-model transformations.

4.2.2 Model-Driven Compilers

As stated in section 2.1.3.3 on page 23, code generators are usually built for one specific source language (e.g. UML) read as model, and for one specific target language generated as text (e.g. Java). This two-dimensions dependency outnumbers the necessary code generators by a cartesian product factor. Moreover, appendix A shows that this architecture can raise several problems when source model needs to be customized, for instance when using a profile.

We believe that a better approach would be to pass through an intermediate model. In the example of a Java code generation from a UML model, the approach we propose would be the following: the UML model would be transformed into a Java model through a model transformation, and then the Java model would be synthesized into Java text files by mean of a code generator. Advantage of this approach is that the semantic domain translation is achieved by a model transformation, which is a dedicated technology, while code generators need to deal with the concrete syntax of the target language. The process separates two distinct tasks (transformation and synthesis) that are performed using appropriate tools.

In case of compilers, a source textual specification expressed in a language A has to be transformed into a target textual specification in language B. To perform this task using a model-based approach to benefit from model transformation technology, an additional step is required compared to model to text transformation: source code must be analyzed into a model, for instance by mean of a text processor. Problem with that approach is that, for a given language L, one needs to provide either a text analyzer, in case L is source language, a text synthesizer, in case L is target language, or even both, in case L may indifferently play the role of source and target language. In this latter case, which appears for instance in round-trip engineering processes, analyzer and synthesizer have to be consistent. This chapter proposes to define synthesizer and analyzer using a unified specification.

Process for compiling a specification given in the `A` language into a `B` specification is given in figure 4.1. In the MDA terminology, white models reside at the M2 level (meta-level) and grayed models reside at the M1 level. Example for `A/B` couples are Java/C++, UML (represented with HUTN or XMI)/Java, and B/C++. One may notice that in these examples Java may play the role of a source or a target language. A textual `A` specification is analyzed into an `A` model. The `A` model is transformed into a `B` model, and the `B` model is synthesized into a `B` representation. It is interesting to see that if the `A/B` transformation is reversible, one may reverse the complete process (from `B` textual representation to `A` textual representation) with no additional development.

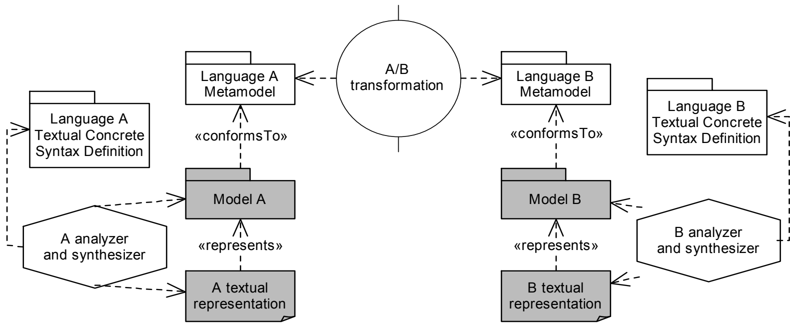


Figure 4.1: Model-Based Compiler Architecture

4.2.3 Related Works

As seen in section 2.2.1 on page 26, text structure may be captured by grammars expressive enough to be specification for automatic text processor generation (compiler compilers). Our problem is to relate text structure to a metamodel, which is a work that is usually performed by hand.

Section 2.1.3.4 on page 24 took a rapid tour of existing techniques related to meta-modeling. We did not consider model interchange, such as XMI or HUTN, or works like [AP04, GRS06], as being flexible enough. If they actually relate a metamodel to some text structure, there is little choice in the form this text structure is given. Oppositely, xText [Voe06b] and Gymnast [Dal05] make it possible to describe any text structure, but there is little choice in the form the metamodel is given.

Code generators, presented in section 2.1.3.3 on page 23, including Netsilon, presented in chapter 3, only solve half of the problem (synthesis). Indeed, they are only able to provide a textual representation for a given model, without the possibility to reverse the operation. AntiYACC [HRS02] is a tool that performs code generation, or more precisely text representation, taking advantage of EBNF. XBNF [CESW05] is another tool that solves the other half of the problem (analysis), that is getting a model from some text. However, to come up with a complete solution, one need to provide two different specifications that have to be kept consistent.

TCS [JBK06] is a textual concrete syntax specification language that was developed in parallel of the approach proposed here, almost at same time. TCS targets the same problem and proposes a comparable solution. Difference is that they do not make their metamodel explicit and expose only a concrete syntax for TCS. Moreover, their principle is to "bridge" modelware and grammarware technological spaces [KBA02] by using compiler technology, similarly to what we achieved in the TCSSL Tools. We tried another mechanism in Sintaks and results are reported in section 4.5.

TEF [Sch07] is a novel framework that is not released yet. It proposes a simple template-based approach to defining textual concrete syntax on top of metamodels.

4.2.4 Requirement for Bidirectional Mapping

Modeling the mapping between abstract and concrete syntax means expressing how a given piece of information can either be stored into an object model (considering that we used object-oriented meta-languages to define the abstract syntax) or represented in text (as we focus on textual concrete syntax). One must consider that there is no one-to-one mapping between abstract and concrete syntax, and further, that there is no single solution either to store information in a model or to represent it in text.

The multiple ways to capture information into an object model are genuinely addressed by object-oriented meta-models. Elementary information can be modeled as a class, an attribute, a relation, or a role. Attributes, relations, and roles may be shared among classes (and relations) in presence of inheritance. Storing more complex information in a given model is then achieved by creating clusters of instances of the modeling-elements defined in their corresponding meta-model.

As seen earlier (section 2.2.1 on page 26), representing information in text follows structures known as grammars. Building a mapping between models and texts therefore

implies understanding how information can be mapped between models (graphs of instances) and texts (sequences of characters).

An instance of modeling element, just as an object, is characterized by an identifier and a state. In an object model, state is stored in the slots of the instances (i. e. values, either of primitive types, or references to other instances of modelling-elements).

Going from abstract syntax to concrete syntax (or the reverse) is then a matter of explaining how pieces of abstract syntax (i. e. values held by slots of modeling elements) are to be serialized (or conversely de-serialized) to pieces of concrete syntax (actually character strings, as we target textual concrete syntaxes).

This means that, in addition to what traditional text analysis tools provide (e.g. terminal, sequences, and alternatives), concrete syntax definition should also offer the possibility to state how individual slots of instances of modeling elements are mapped to concrete syntax.

4.2.5 Towards a Specification

Let's consider the meta-model in figure 4.2 of a language which defines models as a collection of types where types have attributes, which in turn have a type. A possible model and a possible representation for it is given in figure 4.3.

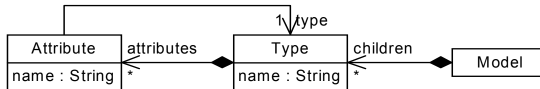


Figure 4.2: Abstract Syntax of a Simple Language

The metamodel on figure 4.2 defines the abstract syntax (the concepts of the language), but nothing is said about concrete syntax. Therefore we have introduced the concept of template which is associated to meta-classes. Each instantiable modeling element (i. e. each concrete meta-class in the meta-model) defines a template whose role is to organize the serialization (de-serialization) of the instances of its (meta-)features, regardless if they are defined in the meta-class or inherited. Each (meta-)feature further defines a feature-mapper, which knows how to fill a given slot with elementary data coming from a text (and conversely, how to generate text from the data).

Templates should also embed facilities for iteration (repeating the same sequence a given number of times, following the example of Netsilon Iterative Composer decision centers) and for alternatives (alternate textual representations according to a given constraint, following the example of Netsilon decision constraints). Moreover, it must be possible to define the value of a slot depending on an alternative; for instance, setting a slot to true or false depending on the occurrence of a given sequence of tokens in the textual representation (or conversely, generating a given sequence of tokens depending on the value of a slot).

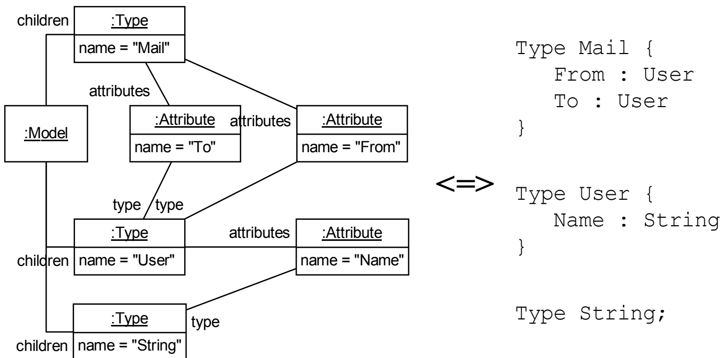


Figure 4.3: Example of Model and a Expected Textual Representation

Alternatives can also be used to specialize children class descriptions based on a single mother class.

Beyond the basic alternative and sequence capabilities, the various mapping options are described by 3 major properties of the feature-mappers: the kind of values held by the slots, the multiplicity of the information to be represented, and the way to share and parameterize feature-mappers. These properties are presented in detail in the following subsections.

4.2.5.1 Kinds of Data

Slots may contain three different kinds of data:

- Attributes values which refer to primitive types; i. e. either data types (such as String, Integer, Real, and Boolean) or enumerations. A single-value feature-mapper, with automatic type translations (to and from text) can handle these attributes.
- Compositions which physically embed the slots of the contained instance of modelling-element into a slot of the containing instance. Representing such relation can be realized straightforwardly by embedding the template of the owned meta-class into the template of the owning meta-class.
- Simple references are a little bit more subtle a problem, and denote that a source instance of a modelling-element refers to a target instance of another modelling-element, using a given key which is in fact a specific slot value. In practice, the textual representation of the referenced instance of modelling-element may appear after the representation of the reference, as in the example presented in figure 4.2 where `Type String` is represented after the `Attribute Name` of `Type User`, whose `type` is

`String`. This referencing capability is also required to implement bidirectional associations (`A` references `B` which in turn references `A`).

4.2.5.2 Multiplicity of the Data

Features are either mandatory or optional, and single or multiple. The optional nature of a feature is rendered by the lower bound of the multiplicity (0 for optional, and 1 for mandatory), the single/multiple nature is rendered by the upper bound (1 or *).

- Representing the mandatory/optional nature can be done by reusing the alternative rule: optional information will be represented by an alternative with an empty branch.
- Representing the single/multiple nature can be done by using either a single value feature-mapper or an iteration.

4.2.5.3 Shared and parameterized feature-mappers

In practice, properties for different classes often share the same concrete syntax. This is especially true regarding inherited properties. Thus, it may be interesting to introduce shared definition in the feature-mapper. Feature-mappers may then be defined outside the scope of given meta-class template, and further called by several meta-class templates; in the same way a procedure may reference a sub-procedure in an imperative language.

Two cases may be distinguished, representing whether a feature-mapper knows the feature to map or not. The first case typically allows reusing the mapper for the same feature within an inheritance hierarchy of meta-classes. The second case (we talk about parameterized feature-mappers) permits to share the same feature-mapper not only by different templates but also by different features, even across meta-class hierarchies.

4.3 Modeling Concrete Syntax

As seen in the previous section, when defining a language, the meta-model of the abstract syntax has to be complemented with concrete syntax information. In our case, this information will be defined in terms of another meta-model, which has to be used as a companion of the metamodel already used for defining the abstract syntax of the language under specification.

Figure 4.4 summarizes the approach, and shows how a tool can automate both analysis and synthesis. Following the MDA terminology, grayed items reside at M1 level (model level), white items reside at M2 level (metamodel level), and black items at the M3 (meta-metamodeling) level. At runtime, the models of abstract and concrete syntax are interpreted by a generic machine (written only in terms of both meta-models) which performs the bidirectional transformation between texts and models. Figure 4.4 can be compared with

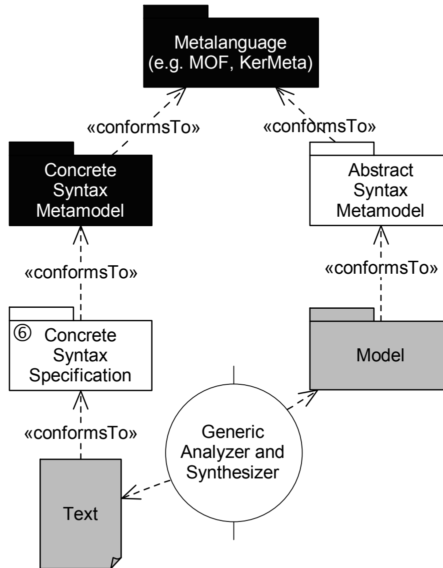


Figure 4.4: Automatic Bidirectional Model Representation

figure 1.2 on page 7. Generic analyzer and synthesizer corresponds to the reversible text processor.

Fully defining the syntax of a language is achieved by combining the abstract syntax meta-model with one or more concrete syntax model(s). The effect of parsing a text (conforming to a concrete syntax model) is to create a model (conforming to the abstract syntax meta-model). Conversely, the text can be synthesized from the model.

Interestingly, both processes of analysis and synthesis are highly symmetric, and since they share the same description, they are reversible. Indeed, a good validation exercise is to perform two synthesis-parse sequences, and observe that there are no significant differences in both generated texts.

4.3.1 Overview of our Proposal

Our meta-model for concrete syntax is displayed on figure 4.5. Given concrete syntax has a top-level entry point, materialized by the `TCSSpec` (Textual Concrete Syntax Specification) class which owns top-level rule fragments and meta-classes. A model of concrete syntax is built as a set of rules (the sub-classes of abstract class `Rule`). The bridge between the meta-model of a language and the model of its concrete syntax is based on two meta-classes:

Class and Feature referencing the class of the abstract syntax metamodel and their properties, respectively. Class Template makes the connection between a class of the meta-model and its corresponding rules. Class Value (and its sub-classes) and class Iteration make the connection between the properties of a class and their values. Class Iteration is used for properties whose multiplicity is greater than 1. The remaining classes of the meta-model provide the usual constructions for the specification of concrete syntax such as terminals, sequences and alternatives.

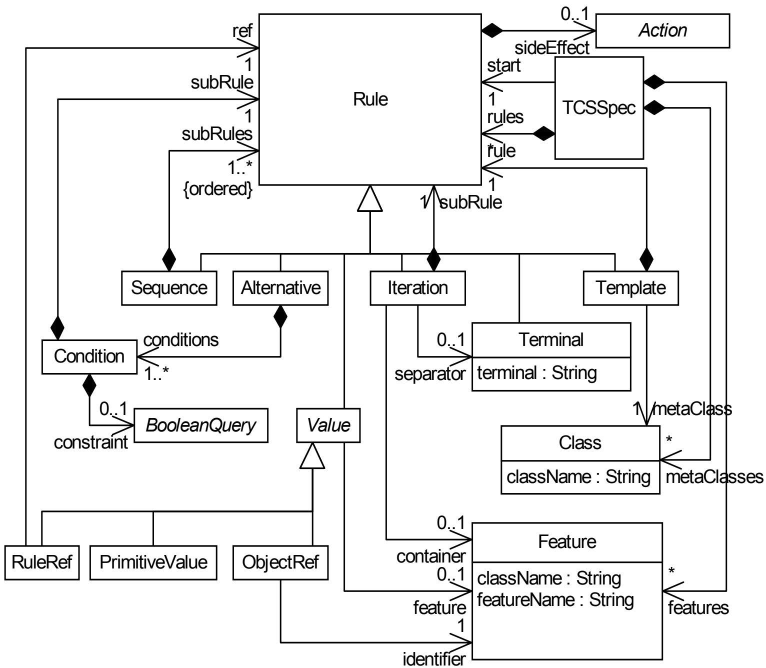


Figure 4.5: Overview of the Metamodel for Textual Concrete Syntax

The following sub-sections detail the semantics associated to each elements of our concrete syntax meta-model. Semantics is given in plain English by providing a short description and an overview of the behavior at analysis and synthesis time.

4.3.1.1 `Template` Rule

A `Template` rule makes the connection between a class of the metamodel (property `metaClass`) and a sub-rule.

Analysis semantics: The template specifies that an object should be created. The metaclass is instantiated and the object (i.e. the concept occurrence) is set as the current object (`self`). The sub-rule is invoked and the current object is initialized. If an error occurs the current object is dismissed.

Synthesis semantics: The template specifies which object to serialize. The sub-rule is invoked to generate the corresponding text.

4.3.1.2 `Terminal` Rule

A `Terminal` rule represents a text whose value is constant and known at modeling time. The text value is stored in the property `terminal` of type `String` in class `Terminal`. It is the counterpart of EBNF terminals.

Analysis semantics: The text in the input stream must be equal to the terminal value. The text is simply consumed. If the text does not correspond an exception is thrown.

Synthesis semantics: The terminal value is appended to the output stream along with formatting information, such as white spaces.

4.3.1.3 `Sequence` Rule

A `Sequence` rule specifies an ordered collection of sub-rules. A sequence has at least one sub-rule. It is comparable to classic EBNF concatenations.

Analysis semantics: The sub-rules are invoked successively. If any sub-rule fails the whole sequence is dismissed.

Synthesis semantics: The sub-rules are invoked successively.

4.3.1.4 `Iteration` rule

`Iterations` specify the repetition of a sub-rule an arbitrary number of times. An iteration uses a collection (property `container` of type `Feature`), and may have a terminal to be used as a separator between elements (property `separator` of type `Terminal`). It is the counterpart of the Netsilon iterative composer, with that difference we rely on an object property (reference), instead of a complex expression. However, for complex expression, one may refer to derived properties if model repository supports it.

Analysis semantics: The sub-rule (and separator, if specified) is invoked repetitively, until the sub-rule fails. For each successful invocation the collection specified by the container feature is updated.

Synthesis semantics: The sub-rule is applied to each object in the referenced collection, and the optional separator (if specified) is inserted between the texts which are synthesized for two consecutive elements.

4.3.1.5 `Alternative` rule

`Alternatives` capture variations in the concrete syntax. An alternative has an ordered set of `Conditions` which refer each to a given sub-rule. A boolean expression in a query language may constrain choice for the correct condition. Queries may be written using languages like OCL, KerMeta, MTL, Xion, or even JMI Java code; for instance TCSSL tools uses EMF Java code, and Syntax defines its own language for comparing a feature with a value, or testing the object's metaclass. One may compare alternatives with Netsilon decision centers, and conditions to decision constraints.

Analysis semantics: This is the most complex operation. Often there is no clue in the input stream to determine the condition (in the sense defined in the metamodel for textual concrete syntax) which held when the text was created. It is therefore necessary to infer this condition while parsing the input stream. The simplest solution (but also the most time consuming) is to try each branch of the alternative until there is a match. If TCSSL tools require alternatives not to be ambiguous following principles of LL(k) parsing, Syntax implements such backtracking algorithm (see section 2.2.1 on page 26). It is worth noticing that in the latter case, the conditions' order can also be used to handle priorities between conflicting sub-rules. Conditions' queries should be enforced once the analysis is completed. If Syntax does not make use of the conditions' queries at analysis, TCSSL tools force them to be a comparison with an attribute: at analysis, the comparison is interpreted as an affectation. More experiments should decide whether constraint solving or constraint enforcement are helpful in choosing an alternative.

Synthesis semantics: The conditions are evaluated in the order defined in the collection, and the first one which evaluates to true, triggers the associated rule.

4.3.1.6 `PrimitiveValue` rule

The rule `PrimitiveValue` specifies that the value of a feature is a literal. The type of the referenced feature should be a primitive type such as Boolean, Integer or String. It is the counterpart of the Netsilon value displayer decision constraint.

Analysis semantics: The literal value corresponding to the type of the feature is parsed in the input stream. The result is assigned to the corresponding feature of the current object unless the type conversion failed.

Synthesis semantics: The value of the feature in the current object is converted to a string and appended to the output stream.

4.3.1.7 `ObjectReference` rule

This rule implements the de-referentiation of textual identifiers to objects. Identifiers (such as names or numbers) are used in texts to reference objects which bear an attribute whose value contains such identifiers.

Analysis semantics: The reference which is extracted from the input stream is used as a key to query the model so as to find a matching element. If there is a match, the parser updates the element under construction. If there is no match, the parser assumes that the referenced item does not yet exist (because it might be defined later in the text) and creates a ghost to be referenced in place, and finally updates the element under construction with a reference to that ghost. By the end of the parsing process, all ghosts have to be resolved unless there is a parsing error.

Synthesis semantics: The identifier is printed to the output stream.

4.3.1.8 `RuleRef` rule

The rule `RuleReference` references a top-level template, stored under the root of the concrete syntax model, i.e. a `TCSSpec` occurrence. It is the counterpart of the Netsilon composer.

Analysis semantics: The ref rule is triggered and the result is assigned to the feature of the current object if defined, or passed as the resulting object of the rule (e.g. when included in an `Iteration`).

Synthesis semantics: The ref rule is triggered.

4.3.1.9 `Action` side effect

An action is an instruction that has a certain impact on the model. Following the example of `BooleanQuery`, it may be written in any language capable to impact a model. Examples of such languages are KerMeta, Xion, and JMI Java code.

Analysis semantics: The action is performed on model at the end of the rule application. The contextual object (`self`) is the one of the rule.

Synthesis semantics: Action is merely ignored.

4.4 Examples

The following section shows how the concrete syntax metamodel is used for specifying concrete syntax.

4.4.1 A very simple example of concrete syntax specification

Going back to our small language example of figure 4.3, we will now use our meta-model of concrete syntax to specify the textual representation. In the example, there is no specific materialization of the model in the text. A type is declared by a keyword followed by a name and an optional collection of attributes. A collection is denoted by curly braces; an empty collection is specified by a semi-column. Notice that the notation allows forward references to `User` and `String`.

A straightforward model of this concrete syntax is shown in figure 4.6. In this model, there is only one top-level rule which describes the concrete syntax of the language. The model ① is built as a cascade of rules. The model starts with an iteration over types ②. The sequence explains that types start with the keyword "Type" ③, followed by a name ④, and then an alternative ⑤ whose conditions are expressed in OCL, because types may have a collection of attributes. The collection of attributes is expressed by an iteration ⑥, which in turn contains a sequence made of a name, followed by a separator (terminal ":") and finally a reference to a type according to this latter's name ⑦. Attributes, when present, are delimited by curly braces. If ④ is an example for a mandatory attribute mapper, ⑥ is an example for an optional multiple reference mapper, as classified in section 4.2.5.1.

Often, it is desirable to share some part of the concrete syntax. Therefore templates do not have to be nested, and can be defined individually at the top level of the model of the concrete syntax. Figure 4.7 represents such variation, for the same concrete syntax. Here, `TT` and `AI` rules are now owned by the `root` specification, and links between independently defined templates are realized with rule references (`RuleRef`). Actually, ④ and ⑥ are promoted to shared feature-mappers. Both representations are totally equivalent. The parsed models or the generated texts are identical, and such transformation may be seen as a model refactoring using techniques as in [MB05].

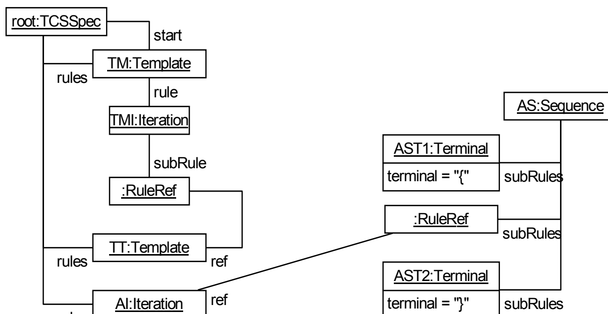


Figure 4.7: Variation with Top-Level Reusable Templates

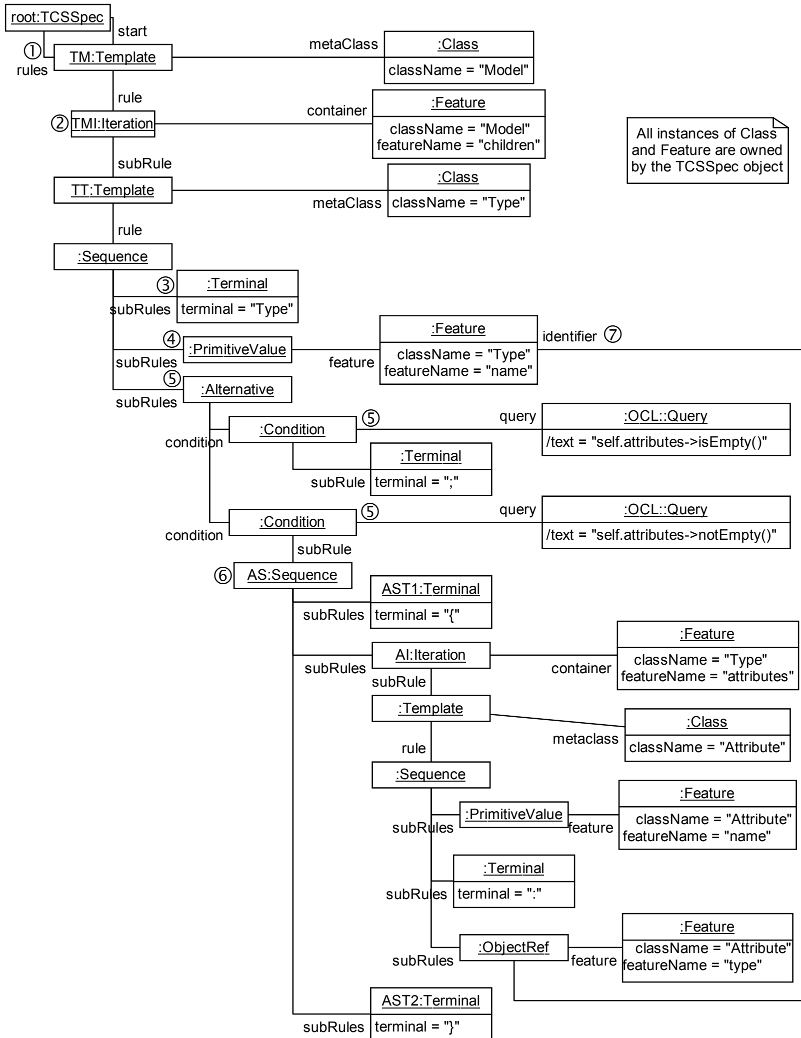
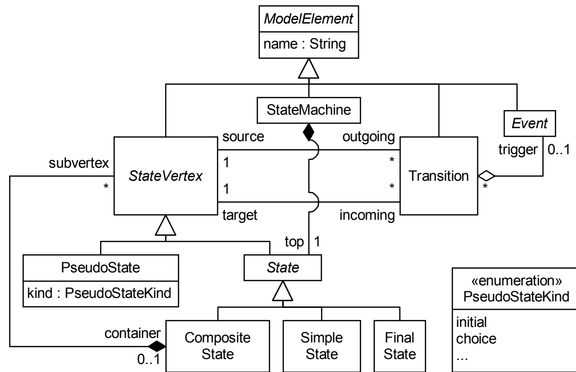


Figure 4.6: Straightforward Textual Concrete Syntax Model




```
context FinalState inv: self.outgoing->isEmpty()

context PseudoState inv:
  (self.kind = PseudoStateKind::initial) implies
    ((self.outgoing->size <= 1) and (self.incoming->isEmpty))
```

Figure 4.8: The Simplified Statechart Metamodel

4.4.2 Statechart Textual Concrete Syntax Example

We propose here another example for the more complex statechart language. Statecharts [Har87] allow to represent state machines visually, but we propose here a textual version.

Statecharts are part of the UML specification [AAB+07]. Thus, concepts are already captured in the form of a metamodel. For sake of simplicity and readability, we will restrict ourselves to a simplified subset of these concepts, as shown by figure 4.8. State vertices might be connected by transitions. A transition has exactly one source vertex and one target vertex. A vertex is either a pseudo state (initial state, choice,...) or a state, which is in turn either a composite state (i.e. containing other vertices and transitions), a simple state, or a final state. Transitions are triggered by events. A state machine is given by its top state. Figure 4.8 presents those concepts and their relationships using MOF [ACC+06], the OMG language for defining metamodels. This metamodel is complemented with well-formedness rules in form of OCL constraints enforcing its correctness and its consistency. Two constraints are represented here using the OCL. Figure 4.8 shows 2 examples, the first constraint ensures that no final state is a source for any transition. The second constraint states that an initial pseudo-state has only one outgoing transition, and that it is never the target of a transition.

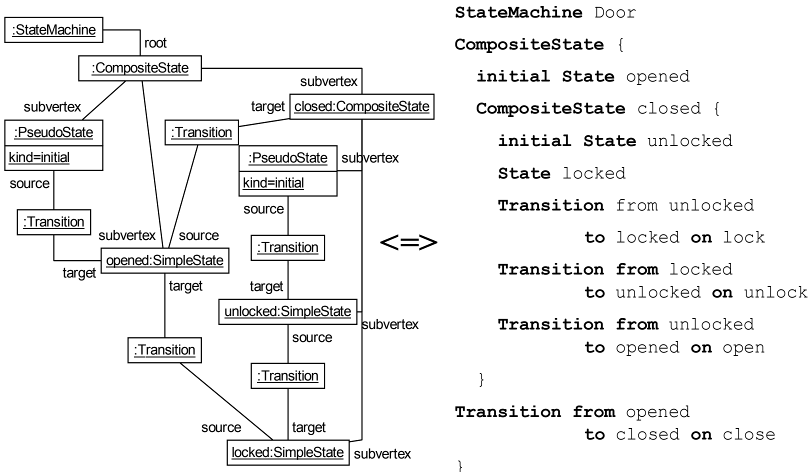


Figure 4.9: A Statechart Sentence and a Textual Representation

An example for a language sentence is given in figure 4.9. The same sentence is represented using MOF to give a good overview of the model (i.e. the abstract syntax tree), and in a textual form that we propose to define here. The interested reader can find the same specification using the standard graphical notation in figure 5.3 on page 86. The modeled system is a door. It states that a door may be opened or closed. If door is closed, it may be locked or unlocked. For sake of readability, events were omitted in the MOF representation.

An excerpt of the textual concrete syntax we propose is shown in figure 4.10. The specification starts with a template rule `SM` for `StateMachines`. The rule is a sequence of a "StateMachine" terminal `smt`, the name of the state machine `smn` (`Door` in the example of figure 4.9), and a reference `smtr` to a rule `S` to provide the `top` state of the state machine. Rule `S` is an alternative between template rules for `SimpleState` and `CompositeState`, depending on the state kind, thanks to OCL `oclIsKindOf` queries `ssq` and `csq`. Template rule `CS` for `CompositeState` is a sequence of a rule reference `init`, which manages the optional "initial" keyword, a "CompositeState" terminal `cst`, the state's name `csn`, and, between curly brackets `ocb` and `ccb`, an iteration `CSS` on contained states. `CSS` is making reference to the rule `S`, which thus makes possible, in the concrete syntax, to specify both composite and simple states within a composite state. This is the case in the example of figure 4.9 since the top composite state contains the `opened` simple state and the `closed` composite state. The presence of the "initial" keyword `ik` is managed by the `init` alternative, which is a rule that is called by the `CS` rule. In case the "initial" terminal is recognized,

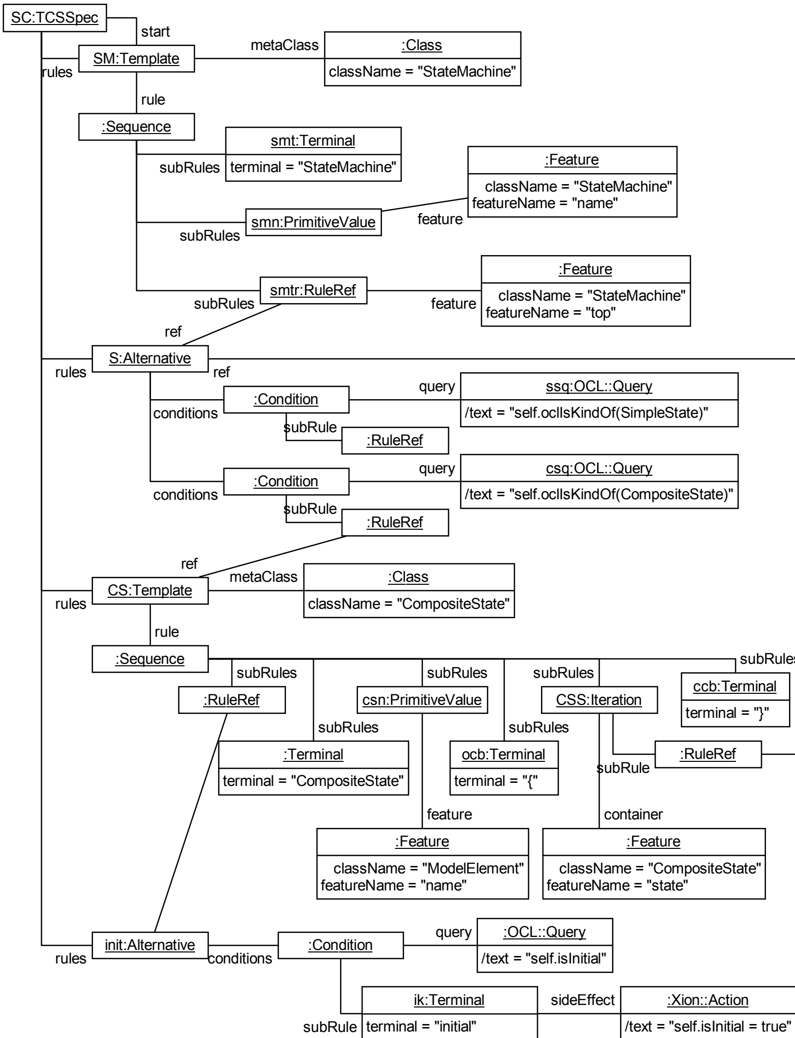


Figure 4.10: Textual Concrete Syntax Specification for the Statechart Language

the considered state should be an initial state. An initial state is a state which is targeted by a transition whose source is an initial `PseudoState`, following the examples of the `opened` and `unlocked` states. This `init` alternative should also be referred to by a rule template for `SimpleState`, which is not represented in the figure, as a simple state may also be an initial state. Actually, we cheated with the metamodel in figure 4.5, as both the test query and the action to be taken make use of an `isInitial` attribute, which is not available in the statechart metamodel of figure 4.8. Instead, we should have defined a more complex query and a more complex action, which would have tested or updated the model properly. It is important to see that actions and queries should be kept consistent; we face here problem of over specification and coherence checking. Note that this attribute may be defined on the metamodel, as a derived attribute. However, this would pollute the metamodel to simplify the concrete syntax specification process. In OCL, query to test whether a state is an initial state could be the following:

```
self.incoming.source->exists(s| s.oclIsKindOf(PseudoState)
   and s.oclAsType(PseudoState).kind = PseudoStateKind::initial)
```

In Xion, the action could be the following:

```
PseudoState init = new PseudoState();
init.kind = #initial;
Transition t = new Transition();
t.source =  init;
t.target = self;
```

4.5 Prototype Implementations

Two prototype tools were implemented following this approach, based on two variants of the metamodel we propose here: Sintaks and TCSSL Tools.

The Sintaks prototype is based on recursive descent, and realizes both analysis and synthesis of concrete syntax. It has been implemented on top of EMF in Eclipse. We have not been trying to achieve high parsing performance; we have simply been investigating how modeling could be used to describe concrete syntax. Queries are formulated in a dedicated language able to test attribute values and object types. Actions are not supported. This prototype has been used to parse and pretty-print several DSLs.

TCSSL Tools was developed at CEA and is a set of two different tools that make use of the same concrete syntax specification. Actions are given as EMF Java instructions, and queries are attribute value comparison to a EMF Java expression. The first tool compiles a concrete syntax specification into a compiler compiler specification (namely ANTLR). The result is an LL(k) text processor that builds an EMF model from a textual representation. The second tool is a pretty printer. A concrete syntax was defined for the specification, and it was possible to create the parser with TCSSL Tools itself, thus following a bootstrapped

approach. The tool made it possible to define a concrete syntax for UML Action Language. More implementation details about TCSSL Tools are available in [FSGM06].

Advantage of TCSSL Tools is to offer more flexibility in the defined syntaxes than Sintaks, as it embeds a more complex test and action mechanism. Its also more efficient as it makes use of the latest compiler compiler technology. The main drawback is the drawback of lookahead parser technology in general, and LL(k) in particular, that is neither able to backtrack in case of parsing error, nor to accept an ambiguous specification (as defined by the LL(k) technology) as input.

4.6 Conclusion

This work may be viewed as an experimentation for the specification of concrete syntax in the context of meta-modeling applied to language engineering.

We have proposed a novel approach, based on meta-models, which supports a formal bidirectional mapping of both concrete-to-abstract, and abstract-to-concrete syntax. Our main contribution is the definition of a meta-model of the mapping between abstract syntax and concrete syntax.

Our work is obviously far from bringing definitive answers to the complex problems of applying meta-models to language engineering but, along with the capabilities of executable meta-languages such as Kermeta, it suggests that languages can be fully specified in terms of meta-models, and that tools can be automatically derived from these meta-models to support these languages.

A lot of work is still beyond us to make tools based on this approach as robust and efficient as the one in the grammarware space. For instance, one important challenge would be to build a bidirectional and incremental text processor that would update the model when the text change, and that would update the text while the model changes. However, the presented material may contribute, with many other ongoing research works to a better understanding of metamodeling applied to textual language engineering.

Next chapter will explore the slightly different problem of graphical concrete syntax specification.

Chapter 5:
Graphical Concrete Syntax

Model Driven Development and Domain Specific Languages are two trends in software engineering which cause a proliferation of modeling languages when used together. New modeling languages need a precise specification of their syntax and semantics to gain acceptance. While metamodeling is a comprehensive mean for defining the abstract syntax, i.e. the concepts of a modeling language, most language specifications are held informally for the description of the semantic and for the (graphical) concrete syntax. This chapter is tackling the problem of defining graphical syntaxes on top of a metamodel stating abstract syntax using a two-step process: specification and realization. For the specification part, we propose to build a second metamodel to express concrete syntax. Declarative relationships put in relation metamodel for the abstract syntax and metamodel for concrete syntax. Regarding realization, we propose to define in a constructive way shape of graphical elements using the XML-based standard Scalable Vector Graphics (SVG). Thus, the graphical representation of a model is an SVG document that may automatically be controlled using the Document Object Model (DOM) technology. In this part, we also identify a set of predefined components making use of DOM that can participate to the SVG shapes for specifying possible user interactions. Relationship between representation, defined in the specification step, and concrete syntax, defined in the realization step, is performed by events triggering action language scripts, such that the complete specification is machine understandable.

This chapter was partly published in the Model Driven Architecture - Foundations and Applications, First European Conference 2005 [FB05] and is complemented by the bachelor semester projects [Hon05] and [RH06].

5.1 Introduction

Previous chapters introduced motivation for defining concrete syntax for modeling languages, for example in section 4.2.1 page 64. In the last chapter, we proposed a mean to define textual concrete syntaxes. In this chapter, we propose an approach to specify graphical concrete syntaxes for languages whose abstract syntax is provided as a metamodel. We use here metamodeling for modeling concrete syntax and the SVG [JN05] / DOM [HHW+04] technology regarding concrete visualization and interactions.

The approach is divided in two distinct parts: *specification* and *realization*. In the specification part, language engineer models a graphical concrete syntax in terms of a com-

panion metamodel of the abstract syntax. This metamodel is intended to capture structure of representation data. In contrary to previous chapter, in which we fixed a semantically rich metamodeling language to concrete syntax definition, the graphical concrete syntax may be described using object-oriented metamodeling languages as MOF, KerMeta, EMF, or Alloy. The metamodel for concrete syntax can be improved by constraints for stating graphical rules regarding spatial relationships. To keep metamodels for abstract and concrete syntax consistent, we propose an approach making use of an intermediate matching metamodel, also extended by constraints. As a summary, goal of the specification part is to give all possible sentences of graphical concrete syntax in a restrictive way.

The specification step states the representation data and data coherence rules, and the task of icon definition and user interaction is let to the realization step. Do do so, we propose to define icons using the SVG language for two-dimensional vector graphics. User interactions may be defined in libraries of predefined DOM components capable to alter the representation. Relation between the specification and the realization is achieved through an event system: the execution of a DOM component may trigger an action on the representation data, and an alteration of the representation data may be listened by the icons. Finally, we should end up with a specification including abstract syntax, specification of concrete syntax and realization of concrete syntax. The complete specification may be interpretable by a tool, so that can be offered a CASE tool (i.e. a graphical modeling tool) for a such developed language.

Both from system engineers (who make use of the modeling language) and from automated processors (computer programs that create the model) viewpoints, major difference between textual and graphical language is that textual specifications are compiled, while graphical specifications are interactive. This means that if a textual specification is written and then transformed into a model (as seen in chapter 4), graphical specification is promoted in the model on an incremental basis. Usually, model is built interactively while the system engineer interacts with CASE tool to graphically draw his/her specification (see section 2.2.2 on page 27). Advantage is that the CASE tool may reason on the model at same time it is built (e.g. a model element is created, two model elements are put into relation). One may remark that such an interactive technique is more and more applied to textual languages in modern IDEs which maintain an abstract syntax graph of the textual specification to support system engineers in code production. We propose here such an interactive technique to achieve the realization step that creates a model according to user interactions.

Figure 5.1 summarizes the approach. Following the MDA terminology, grayed items reside at the M1 level (modeling level), white items reside at the M2 (metamodeling) level, while black items reside at the M2 (metamodeling) level. Abstract syntax of the language is developed in terms of a metamodel to represent concepts and their relations (i.e. vocabulary and taxonomy). A concrete syntax is developed also in terms of metamodel. Neither abstract syntax has knowledge of concrete syntax, nor concrete syntax has knowledge of

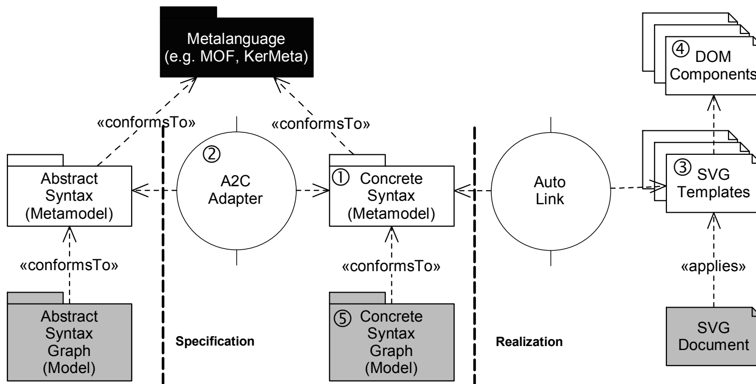


Figure 5.1: General Architecture

abstract syntax: only bridge is an abstract to concrete syntax adapter which is in charge of keeping both models consistent. The abstract to concrete adapter has to be specifically developed for a given abstract syntax and a given concrete syntax. Major advantage is that neither abstract syntax is polluted by concrete syntax consideration, nor concrete syntax is a decoration for the abstract syntax. This makes it possible to reuse the same concrete syntax for different abstract syntaxes (e.g. class diagram notation which is in use in various languages and language versions as UML, MOF, Netsilon, Fondue, etc.). An abstract syntax may also define various concrete syntaxes (e.g. UML models that may be represented using Booch notation). Realization of the concrete syntax must clearly state what is the graphical appearance of elements of concrete syntax and how they may be manipulated (e.g. moved, resized).

Figure 5.1 can be compared with figure 1.2 on page 7. Abstract syntax, model, SVG templates, and DOM components are rendered in both figures. Concrete syntax corresponds to the graphical syntax metamodel (①), while concrete syntax graph embodies the representation data (⑤). The SVG document is that document which is rendered by SVG renderer. As detailed later, the synchronization between abstract and concrete syntax (A2C Adapter) will be performed by a mapping model (②).

We first introduce two simple yet illustrative examples, the statechart and the chessboard languages, in section 5.2 and provide an overview of the specification step in section 5.3. Then, we describe the realization step in section 5.3. We finish with comparison of related approaches in section 5.5 before concluding in section 5.6.

5.2 Language examples

We give here an overview of two simple examples: statecharts and chessboards that are connection-based and geometric-based languages, respectively.

5.2.1 Statecharts

The concepts of the statechart language as introduced in section 4.4.2 page 78 are represented by the symbols shown in figure 5.2. There is no need to define the `StateMachine`

Transition	SimpleState	Composite State	FinalState	PseudoState (initial)	PseudoState (choice)
–event→	name	name / contents	◉	●	○

Figure 5.2: Symbols for the Statechart Concepts

symbol since a state machine cannot be represented. Figure 5.3 presents a statechart sentence both as an instance of the statechart metamodel, and as an "instance" of its concrete syntax. For the sake of readability, events are suppressed in the metamodel instance.

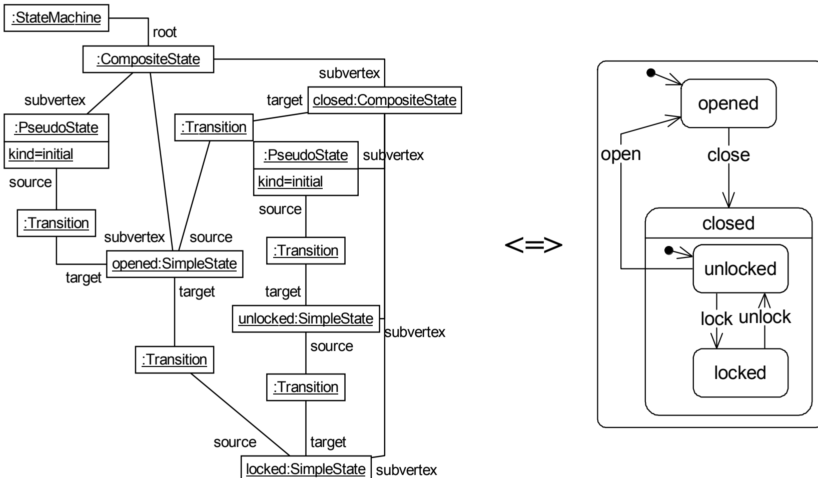


Figure 5.3: A Statechart Sentence and its Graphical Representation

This simple example already reveals some of the weaknesses of informal concrete syntax definitions. The definition given figure 5.2 is incomplete because, for example, it was not stated yet that an event should be shown *nearby* the transition it triggers. Moreover, the name of a simple state should be contained in the representation. The same problem appears for the composite state: the name should be contained in the upper part of the composite state icon, the lower part being reserved for the composite state contents, that is the representations of its sub-states. A transition representation (an arrow) has to go from the representation of the transition's source to the representation of its target. Some symbols contain parameters (as `event` in Transition) as placeholders for additional information. It is, however, not specified yet where the information comes from, e.g. that the event attached to a transition is indeed that event that triggers the transition. Another problem is that there may be different icons for the same concepts, as for the PseudoState. Again, the choice, which in this particular case depends on the PseudoState kind, is not captured by the figures. It can also be the case that one icon may present some variants in its representation. For instance, it may be possible to represent a composite state without its name part, or without its contents part.

5.2.2 Chessboard

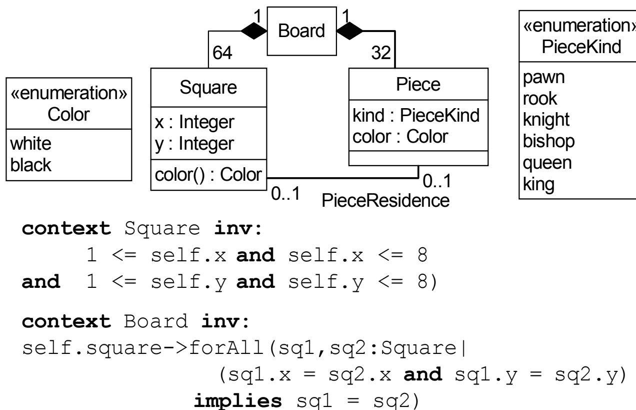


Figure 5.4: The Chessboard Metamodel

We define in figure 5.4 metamodel for a very simple language whose sentences are all possible positions of pieces on chess boards. A situation refers to a board, so we introduce the metaclass `Board` to represent the board concept. This board consists in 64 squares, so we introduce a metaclass `Square` and an association forcing each `Board` model element to be

related to 64 `Square` model elements. Squares can be identified by their position; this is modeled by the `x` and `y` integer meta-attributes. Of course, `Square` model elements must define values for `x` and `y` between `1` and `8`; that can be modeled with a constraint written in OCL. Moreover, two squares of the same board must not have the same position. Again, this can be stated by a constraint. Squares are also qualified by a `color` (`black` or `white`) that can be computed from their position; this can be represented using a meta-operation. The board also consists in 32 pieces, of a given color, and of different kinds: `pawn`, `rook`, `knight`, `bishop`, `queen`, and `king`; this can be modeled in a same way by introducing a `Piece` metaclass with a `color` and a `kind` attribute, together with an association. Moreover, if not captured, a piece resides on a square, thus preventing any other piece to occupy this square, as could be shown by a `PieceResidence` meta-association. If captured, a piece is aside and does not reside on any square. Some additional constraints should be defined to state the kind of pieces that have to be part of the game (16 pawns, 4 rooks, 2 queens, etc.), but they have been omitted for sake of brevity. Of course, the presented abstract syntax is not at all the unique possible one for the chess board language. For instance, one could prefer avoiding the `Square` metaclass and directly assigning the position to pieces.

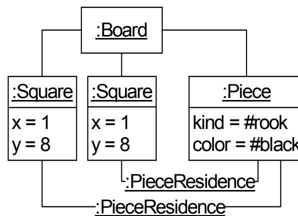


Figure 5.5: An Incorrect Chessboard Model

Figure 5.5 is an attempt to provide a conforming model. It presents a `Board` model element, related to two `Square` model elements and one `Piece` model element. Each square is related to the piece by the mean of two links of the `PieceResidence` type. However, that model does not conform to the chessboard metamodel for three different reasons. First, the metamodel states that a `Board` model element must be related to 32 different `Piece` model elements and 64 `Square` model elements. In this case, the board is only connected with one piece and two squares. Second, a piece is located on two different squares, which is not allowed by the `PieceResidence` meta-association `0..1` multiplicities: a piece must be located at most on only one square. Third, the two square abstract representations have the same position as indicated by the values they provide for the `x` and `y` meta-attributes, thus violating the constraint depicted in figure 5.4.

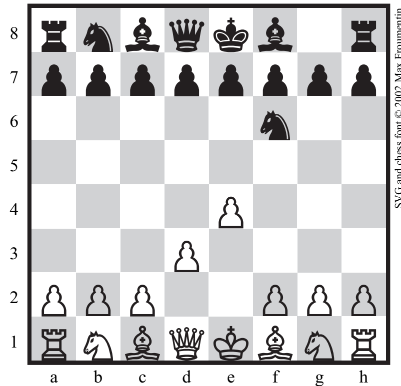


Figure 5.6: A Chessboard Sentence

Figure 5.6, on the contrary, depicts a correct chessboard sentence representation, but this time using the chessboard concrete syntax instead of raw instances of the metamodel given as object diagrams. An informal version of chessboard concrete syntax should illustrate the gap between the abstract and concrete syntax. Such a definition could state that (1) the board is displayed by the two, orthogonal x and y axes annotated with 1..8 and a..f, (2) the squares are represented by a white or gray square depending on the (derived) attribute color of the model element at a position between the two axes of the board according to the values of the x and y attributes; the size of the squares corresponds to the size of the board, and (3) the pieces that are not captured yet are shown at the corresponding squares by a figure of well-known shapes.

Note, that in the particular case of the chess board language the concrete syntax basically fixes the visual appearance of a sentence, i.e. an instance of the metamodel. Once the position and size of the board is fixed, the position of all other visual objects are determined and there is no alternative way to draw the sentence.

5.3 Concrete Syntax Specification

As shown by the statechart sentence in figure 5.3, each sentence of a language can be represented as an object diagram conforming to the language's metamodel. While language definitions given in form of a metamodel are easy to read and to understand, the sentences of the language are not accessible by humans if given in form of object diagrams. This section will solve the problem of defining a concrete syntax contract, i.e. a more human-friendly

representation for the sentences of a visual language. Missing information, as pointed in section 5.2, will be added, so that the representation will be defined in a non-ambiguous way what dramatically improves the language's usability. For example, the concrete syntax contract definition for the statechart notation must allow representations as shown in the right part of figure 5.3 instead of raw instances of the metamodel given as object diagrams shown at the left of the figure.

It is worth to stress that our aim is different from the complete definition of a visual language that is often tackled by graph grammars. Our approach does not aim in defining a visual language from scratch but assumes that the abstract syntax of the language is already given.

Some languages intentionally allow different representations of the same sentence. Hence, we do not aim to define an algorithm able to display a sentence which is given as an instance of the abstract syntax in a more understandable, visual form. Our approach rather goes the opposite direction and allows to decide whether a given diagram is a correct representation of a given object diagram or not. Thus, our approach sees the concrete syntax contract as a specification how sentences can be displayed. The concrete syntax contract definition will be based on the same principles as well-known abstract syntax definitions given in form of metamodels and will yield an algorithm to decide the correctness of a representation of a sentence. That is why this definition can be considered as a contract to be fulfilled by an representation tool, regardless its technology.

5.3.1 Scheme-based Definition of Concrete Syntax

The definition of a concrete syntax contract means to define (1) a visual language, i.e. visual elements with relevant attributes and relationships between them and (2) how the visual elements are connected to the concepts of the language they are supposed to represent. Figure 5.7 gives an overview how both goals are basically achieved by our approach.

As stated before, language is given by its abstract and concrete syntax (for sake of simplicity, we forget here about semantics). *Abstract syntax* is defined in terms of *metamodel* and a language sentence abstract syntax graph is a model conforming to that metamodel. Concrete syntax is another metamodel with two different parts: *display classes* and *display manager classes*. The display classes are in charge of concrete representation by stating what are graphical elements to be displayed, and can also declare relevant attribute of their objects such as shape, color, size, position, attach regions, etc. For sake of reusability, the display classes do not depend on the metamodel; display manager classes are in charge of putting them in relation. Concrete syntax graph of a sentence is given by a set of *display objects*, each being an instance of a display class, and a set of *display managers* as instances of display manager classes.

The formalism to define display classes is intentionally left open in the specification step, and should be introduced only in the realization step. However, as we will see later,

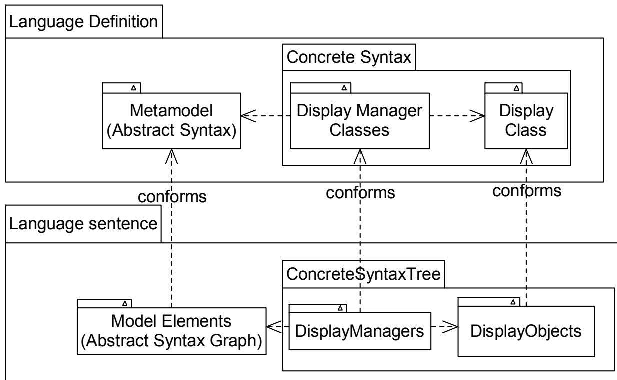


Figure 5.7: Scheme Definition Architecture

display classes are supposed to implement some operations corresponding to (spatial) relationships such as `CONTAIN`, `OVERLAP`, etc. We propose in section 5.4 an implementation based on SVG and DOM components. In other words, display classes are merely specifications of what the realization step is supposed to provide as a service. This approach is equivalent to defining interfaces (specification) that are further realized by components (realization).

Mapping between visual objects and model elements is formulated by display manager classes using a constraint mechanism. For each possible representable metaclass there exists a unique display manager class whose purpose is to restrict the way how instances of the metaclass, i.e. model elements, can be displayed. These restrictions are formulated as invariants written in OCL. Display managers are responsible for consistency between the abstract syntax graph (the model elements) and the concrete syntax graph (the display objects): a display object can exist only in the context of a model element.

5.3.2 Statechart Concrete Syntax

We illustrate our approach by formally defining the concrete syntax of the statechart notation whose abstract syntax was given in section 5.2.1.

Prior to giving the formal definition of the concrete syntax, an informal version of it should illustrate the gap between the abstract and concrete syntax, as already introduced in section 5.2:

Problem 1: A text is shown on the top of transitions to explicit the triggering event if it exists;

Problem 2: A text shows the name of a simple state;

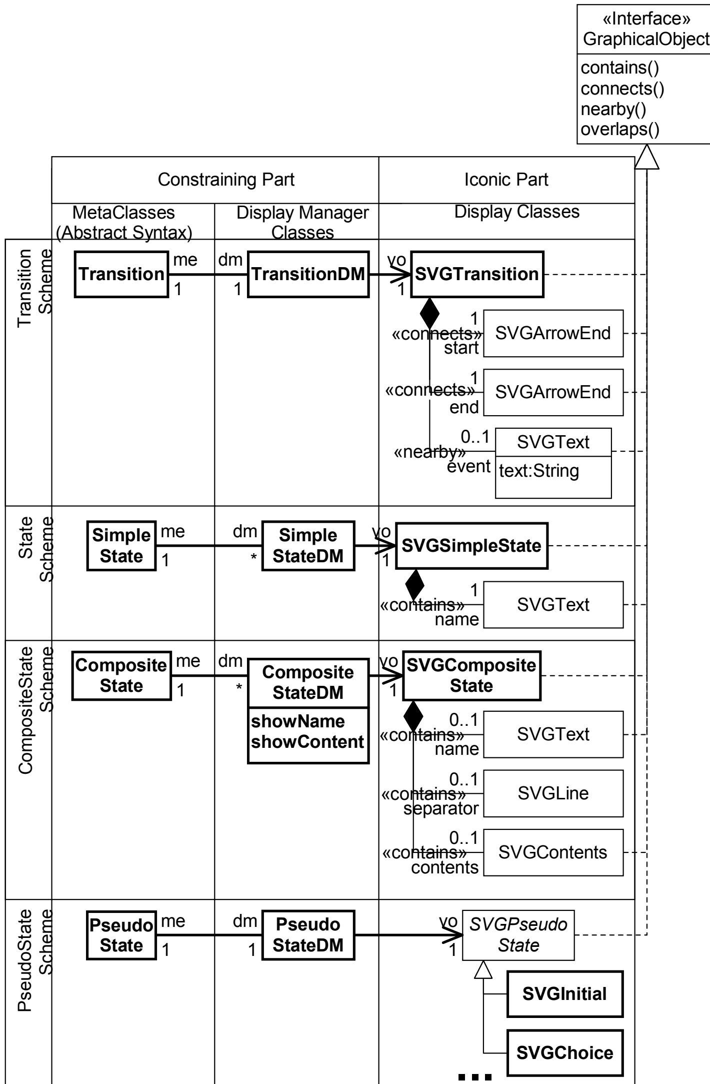


Figure 5.8: Statechart Schemes

Problem 3: To depict a composite state, depending on the viewer's choice, either a text shows the name of the composite state, or a region shows the contents of the composite state (i.e. its contained states), or both; in the latter case, the two regions are separated by a line;

Problem 4: The plain side of the transition icon is connected to a representation of its source state; the arrow side is connected to a representation of its target state;

Problem 5: The shape of a pseudo-state representation depends on its kind.

Figure 5.8 shows the architecture of the statechart concrete syntax definition. We have shown four display schemes related to four graphically representable concepts as defined in the metamodel: `Transition`, `State`, `CompositeState`, and `PseudoState`. `FinalState` have been omitted for the sake of brevity. The other concepts are either abstract (as for `ModelElement`) and thus will be depicted by the scheme of their subclass, or cannot be graphically rendered (as for `Event`). The case of `StateMachine` is a bit particular since it can only be represented by a diagram, which is actually being defined by the schemes.

As proposed in section 5.3.1, the schemes are defined using two layers. In one hand, the display classes, are contracts for the visualization tool/language. The main purpose of this layer is graphical rendering, i.e. iconic definition. In the other hand, the display manager classes fill the gap between those display classes and the abstract syntax by connecting them explicitly. A display manager class is connected to exactly one metaclass through an `me` (for *model element*) association end; thus, there exist a display manager object for each model element representation.

The cardinality of the opposite association end has also its importance. In the case of figure 5.8, a `Transition` or a `PseudoState` model element can only be depicted once, because of the cardinality of the respective `dm` (for *display manager*) association ends. On the contrary, a `SimpleState` or a `CompositeState` model element may be represented several times with different instances of the display manager class (thus different instances of the display class).

5.3.3 Icon-definition within a Scheme

Besides the association to metaclasses, display manager classes are associated with display classes. For each display manager class there is always a standard association to the corresponding display class with multiplicity `1` and role name `vo` (abbreviation for *visual object*) on the end of the displayed class. This reflects the fact that each display manager object is connected with exactly one main display object. This connected display object depicts that model element which is connected with the display manager as well. A display class defines through representation data how a model element is actually displayed in terms of shape, color, etc. It also provides some query facilities. Some standard queries (e.g. `CONTAIN`, `OVERLAP`), as introduced in section 2.2.2 page 27, are declared by the generic `Graphi-`

`calObject` interface. This enforces that any custom display object (like `SVGTransition`, `SVGText`, or `SVGContents`) is capable responding the queries.

Display classes are specifications for the realization step, and do neither depend on the display managers classes, nor on the metaclasses. This enforces a relative independence of the realization language/tool from metamodeling concerns. Indeed, stakeholders concerned by graphical rendering are rarely also specialists in language construction. This independence mechanism is meant to ease the collaboration between *domain specialists*, whose concern is to describe the concepts of the language, i.e. the metamodel, and the *graphical modelers* who aim at graphically rendering the display objects. Moreover, this clear separation guarantees that neither concrete syntax architecture pollutes conception of abstract syntax nor abstract syntax architecture pollutes concrete syntax conception. It also permits a better reusability of display classes for other concrete syntax.

Often, a model element is not displayed just by one atomic visual object but rather by a composition of such objects. However, all involved objects are arranged by one composition object whose class corresponds to the metaclass it should visualize. Figure 5.9 illustrates the definitions done in the display classes related to the metaclasses `CompositeState` and `Transition`.

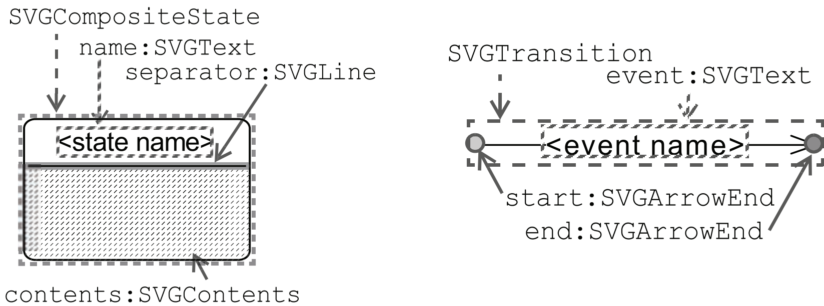


Figure 5.9: The Composite State and Transition Icons

The graphical representations for metaclasses `CompositeState` and `Transition` are defined by the classes `SVGCompositeState` and `SVGTransition`, respectively. A display object of class `SVGCompositeState` may compose three objects to display the name of the state (`SVGText`), the region where the contained states can be arranged (`SVGContents`), and the line separator (`SVGLine`). Those three graphical elements that take part in the `SVGCompositeState` display object are composed by the latter with multiplicity `0..1`. This means that the figure is still valid in the case that some parts are missing for depicting a composite state.

Similarly, objects of SVGTransition are composed of one object to display the event and two objects representing attach points. The difference is that the composed objects of SVGArrowEnd are mandatory: it is an incorrect sentence of the statechart notation to show a transition with no arrow at the end. SVGPseudoState is a little bit different: there exist many radically different shapes to represent it. Each one of these shapes is represented by a display object (e.g. SVGInitial for an initial state, or SVGChoice for a choice) that inherits from SVGPseudoState. As SVGPseudoState does not represent a figure, it is declared as abstract.

Constraints have well proven their ability to help diagram layout activities [BEdLT04, MMM04]. In our example, there are also some issues in enforcing layout within the figures. For instance, *problem 3* stated the conditions under the separator (display object) have to be part of the composite state rendering. This can be defined using a constraint as follows:

```
context SVGCompositeState
inv: (self.name->notEmpty() and self.contents->notEmpty())
  = self.separator->notEmpty()
```

Moreover, there are often condition rules for displaying object parts. For instance, the name, the contents and the separator part of the SVGCompositeState have to be *contained* by the representation of the root display object, here an SVGCompositeState display object. This can be expressed by the following constraint:

```
context SVGCompositeState
inv: self.name->notEmpty() implies self.contains(self.name)
inv: self.contents->notEmpty() implies
              self.contains(self.contents)
inv: self.separator->notEmpty() implies
              self.contains(self.separator)
```

The above constraints take advantage of the «standard» graphical relationship operation as defined in the GraphicalObject interface. This interface is realized by all display classes, such as SVGCompositeState. Since this situation is very common, we introduce stereotypes that can be placed on the compositions that relate parts of a main display object: the name of the stereotype indicates the operation responsible for checking the correct graphical composition of display objects. For instance, by stereotyping the association composing SVGTransition to SVGText with <<nearby>>, we implicitly define the constraint stating that the triggering event name should be depicted near the path representing a transition.

Constraints are also able to cope with the representation variants. For instance, depending on the decision of the designer, a composite state may appear showing only its contents, only its name, or both. This decision is not related to the abstract syntax since it does not change the semantics of the so designed system. To specify this variance in the syntax, we introduce attributes, so called *syntactic attributes*, in the display manager. To render the composite state variants described in *problem 3*, we introduce the attributes

`showName` and `showContents` in class `CompositeStateDM`. The following constraint ensures that the syntactic decision of the modeler is enforced:

```
context CompositeStateDM
inv: self.showName = self.vo.name->notEmpty()

inv: self.showContents = self.vo.contents->notEmpty()
```

It is not necessary to state again on the presence of the separator, since it is already coped by the previous constraints.

5.3.4 Constraint definition within a Scheme

The second part of a scheme for a metaclass imposes constraints restricting the relationships between model elements, display managers, and visual objects. These constraints are also formulated in OCL and attached as invariants to display manager classes. *Problem 2* is a typical example: showing the name of the simple state in the right compartment of the depicting symbol is ensured by placing the constraint:

```
-- problem 2
  context SimpleStateDM
  inv: self.me.name = self.vo.name.text
```

Problem 1 (the event name near a transition symbol) and *problem 5* (synchronizing the pseudo-state shape with its kind) also belong to that family of synchronizing model elements and visual objects problem:

```
-- problem 1
  context TransitionDM
  inv: if self.me.trigger->isEmpty()
    then self.vo.event->isEmpty()
    else self.vo.event.text = self.me.trigger.name
    endif

-- problem 5
  context PseudoStateDM
  inv: let kindAssociation :
       Set(TupleType(kind: PseudoStateKind, type : OclType)) =
       Set{Tuple{kind = PseudoStateKind::initial,
                type = SVGInitial},
           Tuple{kind = PseudoStateKind::choice,
                type = SVGChoice},
           …} in
       self.vo.oclIsKindOf(
            kindAssociation->any(t| t.kind = self.me.kind).type)
```

Some more advanced constraints which are attached to display manager classes aim to express rules in the concrete syntax that restrict the relationships between visual objects, e.g. that the visualization of a contained state is placed at the expected position in respect of

the position of its containing composite state. It is worthwhile to remember that display objects are concrete visual objects and thus 'know' about their position in the space. Thus, each display class has some attributes such as `xPos`, `yPos` encoding the position of the object in the space. However, this coordinate attributes are not of primary importance for the concrete syntax definition and are, thus, not shown. More important is rather the ability of display classes to decide based on the coordination attributes whether a display object is in a spatial relationship such as `OVERLAP` with another display object. In other words, each display class could implement an operation `overlap(GraphicalObject):Boolean` with the well-known semantics. This is stipulated by the introduction of interface `GraphicalObject` declaring operations as `overlap` (see figure 5.8). The fact that each display class implements the interface `GraphicalObject` allows us to describe restrictions of the concrete syntax in form of restrictions on the spatial relationship between visual elements what is at the same abstraction layer as visual elements are usually characterized when defining a visual language. So, the constraints for expressing the *problem 3* (containment of composite state), and *problem 4* (transition start and end connections) are:

```
-- problem 3
  context CompositeStateDM
  inv: self.vo.name->notEmpty() implies
                                self.me.name = self.vo.name.text
  inv: self.contents->notEmpty() implies
          self.me.subvertex->includesAll(State.allInstances().dm
                      ->select(sdm|self.vo.contains(sdm.vo)).me)

-- problem 4
  context TransitionDM
  inv: self.me.source.dm.vo->one(svo|self.vo.start.connects(svo))
  inv: self.me.target.dm.vo->one(svo|self.vo.end.connects(svo))
```

It is interesting to notice that, at that specification level, there is no fundamental difference between schemes rather describing a node (as the scheme for simple state) and schemes rather describing a relationship (as the scheme for transition). Constraints forces the symbols to keep connected if necessary.

5.3.5 Chessboard Concrete Syntax

We apply here the above-described technique to define the concrete syntax of the geometric-based chessboard language as introduced in section 5.2.2.

The informal definition of the concrete syntax of the chess board language can be formalized using *display schemes* as shown in figure 5.10. A display scheme is defined for every metaclass in the metamodel and consists of exactly one display manager class that usually carries many invariants, and some display classes.

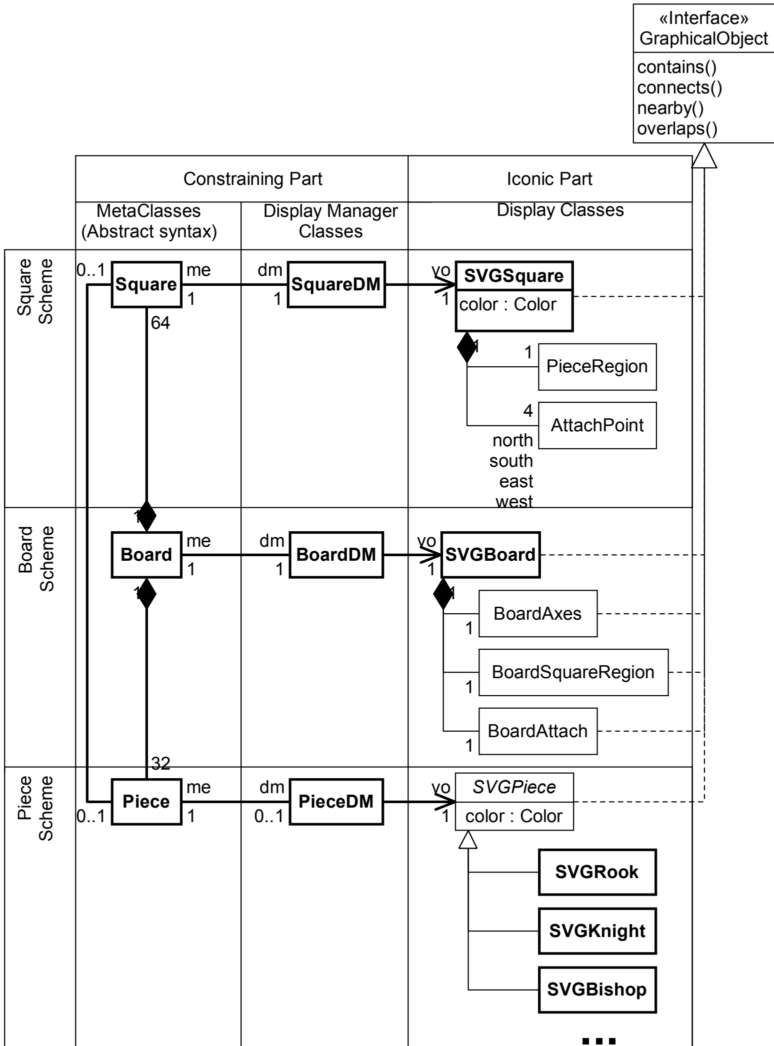


Figure 5.10: The Chessboard Schemes

Each metaclass is connected with its display manager class with an association which has always multiplicity 1 and role name `me` (for *model element*) on the end of the metaclass and usually multiplicity 1 and role name `dm` (for *display manager*) on the end of the display manager class. This latter multiplicity states that each model element *must* appear in the representation. For instance, the association between metaclass `Board` and its display manager class `BoardDM` indicates, that for each model element of type `Board` exactly one display manager object exists. The only exception is `Piece` which may not appear in the representation in case it was captured: in this case, the piece still exist in the model (abstract syntax graph), but should not appear in the representation of the chessboard. It is easy to express that pieces are displayed only if they are not captured yet using a constraint as shown in section 5.3.4 (note that the constraint cannot use `PieceDM` as a context, but has to be stated on the display manager class instead):

```
context Piece inv: self.dm->size() = self.square->size()
```

As for the statechart language, display managers are connected to display classes which represent the iconic part. Figure 5.11 illustrates the definitions done in the display classes related to the chessboard metamodel of figure 5.4.

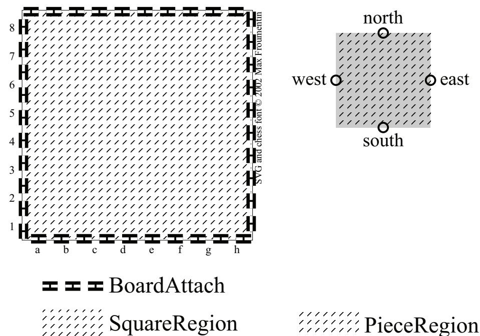


Figure 5.11: The Board and Square Icons

The graphical representation for metaclass `Board` and `Square` is defined by the class `SVGBoard` and `SVGSquare`, respectively. A display object of class `SVGBoard` composes three objects to display the axes (`BoardAxes`), the region where the squares of the board will be arranged (`BoardSquareRegion`), and the attach zone of this region (`BoardAttach`). It is the responsibility of the `SVGBoard` object to ensure that the auxiliary objects for the displaying the axes and square region are always arranged the way as shown in figure 5.11. Similarly, objects of `SVGSquare` are composed of one object to display a

square of the correct color (`PieceRegion`) and four objects representing attach points (`AttachPoint`). Note, that some of the auxiliary display objects, for instance objects representing attach zones, are not shown explicitly in the final diagram (they will further be referred to as *hidden objects*) but such objects nevertheless exist.

Some display classes have attributes to cope with the case that objects of the same class can have a totally different graphical appearance. One example is the attribute `color` in class `SVGSquare` whose objects display both black and white squares. The following constraint ensures that a model element of type `Square` is always represented by a display object of the same color.

```
context SquareDM inv: self.me.color = self.vo.color
```

Note that this rule is given for sake of understandability, but breaks the abstract / concrete syntax disconnection rule. Indeed, `SVGSquare::color` is of kind enumeration `Color` that may be found in the abstract syntax, thus requiring a dependency from concrete to abstract syntax. Even if acceptable, this is a bad display class design as it is no longer possible to reuse the display classes for another abstract syntax just rewriting display manager classes. A better approach would be to redefine a color enumeration in the display classes.

Other constraints which are attached to display manager classes aim to express rules in the concrete syntax that restrict the relationships between visual objects, e.g. that the visualization of a square is placed at the expected position in respect of the position chosen to place the board. For instance, the rule would imply that a square with x=2, y=4 is placed directly right from the square with x=1, y=4. Provided that squares cannot be shown rotated, this is equivalent to saying that the west attach point of the first square overlaps with the east attach point of the second. Our goal to express such constraints formally motivates the two following things.

Again, main display classes all implement the `GraphicalObject` interface. Moreover, display manager are aware of the internal structure of display classes. This allows the formalization of any kind of spatial restrictions, e.g. the above mentioned case of placing the two squares:

```
context s1,s2:SquareDM inv:
   s1.me.x=2 and s1.me.y=4 and
   s2.me.x=1 and s2.me.y=4
implies s1.west.overlaps(s2.east)
```

A more complete solution of the square arrangement problem is achieved by the following invariant:

```
context s1,s2:SquareDM inv:
  (s1.me.x+1=s2.me.x and s1.me.y= s2.me.y
     implies s1.east.overlaps(s2.west)) and
  (s1.me.y+1 = s2.me.y and s1.me.x = s2.me.x
     implies s1.south.overlaps(s2.north)) and
```

```
  (s1.me.x=1
     implies sl.west.overlaps(s1.me.board.dm.vo.boardAttach) and
  (s1.me.x=8
     implies sl.east.overlaps(s1.me.board.dm.vo.boardAttach) and
  (s1.me.y=1
     implies sl.south.overlaps(s1.me.board.dm.vo.boardAttach) and
  (s1.me.y=8
     implies sl.north.overlaps(s1.me.board.dm.vo.boardAttach)
```

Pieces may be represented in various ways depending on their kind. As these representations completely vary, and are not variant of each other as it is the case for square whose only color vary, we preferred to introduce an abstract display class `SVGPiece`, which is realized by different concrete display classes `SVGRook`, `SVGKnight`, `SVGBishop`, etc. However, any kind of piece may also vary in its color. This time, considering a specific kind of piece, it remains a small variation following the example of the square's color. Thus, we introduce a `color` attribute in the abstract display manager `SVGPiece`, that each inheriting concrete classes will have the responsibility to render. To specify choice between possible variants, we place the following constraint:

```
context PieceDM inv:
  (self.me.color=self.vo.color) and
  (self.me.kind=PieceKind::pawn
     implies self.do.oclIsKindOf(SVGPiece)) and
  (self.me.kind=PieceKind::rook
     implies self.do.oclIsKindOf(SVGRook)) and
  (self.me.kind=PieceKind::knight
     implies self.do.oclIsKindOf(SVGKnight)) and
  (self.me.kind=PieceKind::bishop
     implies self.do.oclIsKindOf(SVGBishop)) and
  (self.me.kind=PieceKind::queen
     implies self.do.oclIsKindOf(SVGQueen)) and
  (self.me.kind=PieceKind::king
     implies self.do.oclIsKindOf(SVGKing))
```

5.4 Display Classes Implementation

We have seen a way to specify what is a correct graphical sentence. Nevertheless, we let `GraphicalObject` implementation, actual display classes rendering, and possible user interactions (i.e. user "interactors", that is tools handling user interactions) unspecified. Moreover, we did not state how concrete representation is synchronized with the display objects. We discarded those considerations so far as they are not directly part of concrete syntax. However, they need to be solved when it comes to concrete modeling. We present here an approach based on SVG templates to be used in association with predefined compo-

nents using the DOM API (further referred to as DOM components) to perform all these tasks. A prototype tool was developed to validate the concepts. Note that the proposed approach is one among many possible.

5.4.1 A Template-Based Approach

Scalable Vector Graphics (SVG - see section 2.2.3 on page 29) is an open XML standard for 2D vector graphics. It was engineered by specialists of the domain and has that flexibility to be interactively changed using the DOM architecture as any XML specification.

We propose here a technique based on SVG to define graphical appearance of concrete syntax. Principle of the approach is the following: a diagram is an *SVG document* in which a language engineer may freely add new predefined SVG elements as described in *SVG templates*, and interact with the created elements as allowed by declared DOM components. Each one of these SVG templates corresponds to a *main display class*, i.e. a concrete display class that has a connection to a display manager class. In the example of figure 5.8, main display classes are `SVGTransition`, `SVGSimpleState`, `SVGCompositeState`, `SVGInitial`, and `SVGChoice`. Composed display classes must be described in the template of their topmost container: in the example, a section of the SVG template for `SVG-SimpleState` must describe the `name` part.

When the system engineer decides to add a new element to his/her model, say a `Sim-pleState`, a copy of the SVG template for `SVGSimpleState` is integrated into the SVG document that represents the diagram. In the meantime, an `SVGSimpleState` and an `SVGText` display objects are created, and a relation between the template copy (i.e. the template instance) and the `SVGSimpleState` display object is maintained. According to specification described in section 5.3, the creation of an `SVGSimpleState` display object should trigger the creation of a `SimpleStateDM` display manager. Finally an associated `SimpleState` object, together with a synchronization between the value of the `name` slot of the `State` object and the value of the `text` slot of the `SVGText` display object should be created. We suppose that this last part of the integration of a new model element is realized by a constraint solver at model level, or any model transformation obeying constraints as described in section 5.3. Relation between the template copy (an XML tree as placed in the SVG scene) and the display object (in a model repository) is in charge of synchronizing the value of the `text` slot of the display object and the actual text represented in the scene and that should be described in the SVG template.

Figure 5.12 exemplifies this template-based approach. Main display classes, which are emboldened on the figure, have a corresponding SVG template, and each one of contained display classes has an SVG counterpart in the template according to role name. As an example, a `start` section appears in the SVG template for `SVGTransition`, which corresponds to the contained `start SVGArrowEnd` display object. Note that the SVG section for the `end` display object is different, even though it corresponds to the same `SVGArro-`

wEnd display class: correspondence is given according to role name rather than class. When the system engineer decides to place a new transition in the diagram (i.e. the SVG scene), any $$ occurrence in the SVG template is replaced by an identifier specific to the template instance so that the associated main display object may be found for synchronization purpose. For instance, when value of the event text changes, value of the text slot in the correct SVGText display object must be changed accordingly, including when many SVGText or SVGTransition display objects exist together.

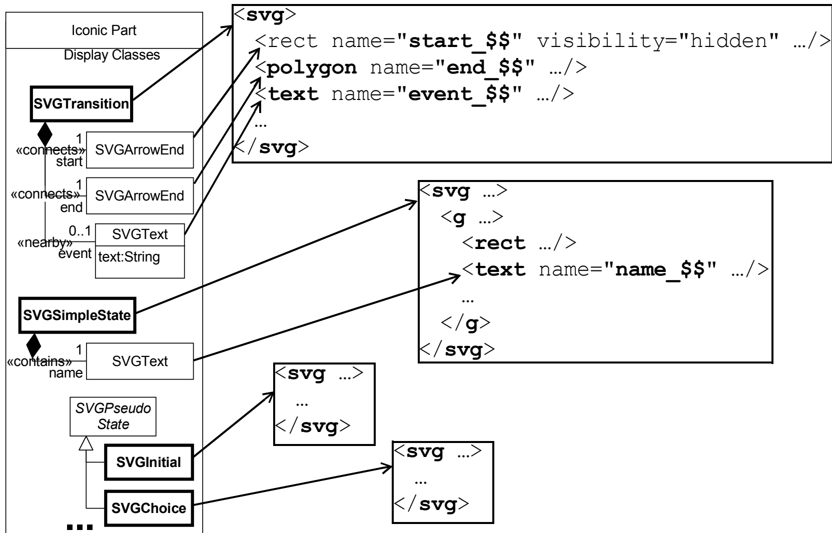


Figure 5.12: SVG Templates for Statecharts

We will see in next sections that template instances are also subject to variations according to user interactions. An example is simple states: if name changes, the containing rectangle needs to grow accordingly. This means that template instance have some dynamic behavior, and may need to be reorganized because of various possible reasons. Templates thus may need to specify a layout mechanism to state how an automatic reorganization may happen. Constraints have long proven to be a comprehensive mean to specify such layout mechanism [Sut63, BMSX97], and we decided to rely on CSVG [MMM04] that is specialized in constraining SVG documents. In the simple state template, the growing-name problem is solved as shown in figure 5.13

First, a variable named w_$$ tracks an arithmetic expression in which the computed width of the name_$$ text plays a role. A first constraint, placed in the rectangle, forces

```
<svg …>
<g …>
  <c:variable name="w_$$"
      value="c:max(c:width(c:bbox(id('name_$$'))) + 20, 150)" />
  <rect …>
    <c:constraint attributeName="width" value="$w_$$"/>
  </rect>
  <text id="name_$$" …>Simple State Name
    <c:constraint attributeName="x" value="$w_$$ div 2 - 75"/>
  </text>
</g>
</svg>
```

Figure 5.13: `SimpleState` SVG Template:
CSVG Constraint to Handle Text Growth

that rectangle to be as wide as the value of the `w_$$` variable. A second constraint, placed in the `name_$$` text, constraints the text to be placed at an `x` position computed so that the text will remain in the center of the rectangle. CSVG tool automatically places listeners in the SVG document. If contents of the `name_$$` text changes, computed value of the `w_$$` variable is updated, which triggers a new computation for the rectangle's width and for the text's position. Note that postfixing any name with `$$` guaranties that no confusion between various simple state template instances will happen.

5.4.2 Predefined DOM Components to Specify User Interactions

SVG is an XML dialect, and an SVG document is an XML tree. DOM is an API that programming languages such as Java use to read and alter XML trees. Thus, a program making use of the DOM API may alter an SVG document. We will further call such kind of program, *DOM components*. We chose an architecture in which user interactions (e.g. mouse moved, mouse clicked, or key hit) trigger execution of some DOM components, which may alter the SVG document that represents the diagram scene. Those DOM components may behave differently depending on the context (e.g. what are the selected elements, what are the elements under the mouse). It is important for the SVG graphical renderer to adapt graphical information according to SVG document as soon as the XML tree is changed. This is the case in our prototype implementation, for which we relied on the Apache Batik toolset [Apa].

5.4.2.1 Architecture

We have used DoPIdom architecture as an inspiration for DOM components organization [Bea06]. We show, in figure 5.14, a conceptual view of the architecture we applied

regarding DOM components. The diagram is an `SVGScene`, whose document is an XML tree (a hierarchy of `XMLNode`s that may own some `XMLAttributes`). The SVG scene, has knowledge of the rendering of each XML node (thanks to Batik), and maintains a list of selected XML elements. Beyond plain SVG, we make it possible for an XML node to declare a set of `Interfaces`. An interface is a `Behavior` that can react to a certain kind of event (`EventKind`), according to a context (e.g. selected component) and persistent `Parameters`. A parameter may be an integer value, a boolean value, a character string value, a reference to an XML node, a collection of values and references, etc. An event may be a `Query` (e.g. to ask for the position of an SVG node - `Location`) or an `Action` (e.g. to set the position an SVG node - `Position`). Beside interfaces, `Interactions` handle user interactions. Examples for user interactions are mouse movement, keystroke, click on the mouse, or action on the mouse wheel. Note that in order to react to a stimulus (event or user interaction), a behavior is likely to send new events to interfaces of XML nodes. In order to be valid, an interface may requires its holding XML node to declare other specific interfaces, e.g. a component with selectable interface should also declare the highlightable interface.

We developed various DOM components (i.e. interfaces and interactors) following the above-described architecture. For instance, one of these interfaces is `Translatable`, which allows the system engineer to move an SVG element. The interface is declared as able to react to a `Translate` event kind. A `Translator` interaction is also defined to react to mouse drag. When the system engineer drags the mouse on the SVG scene, the `Translator` interactor will react by first asking the `SVGScene` to find an interface able to handle a `Translate` event, and held by an XML node at the position of the start of the drag (by mean of the `getInterfaceAtForEvent` method). If such a `Translatable` interface is found, a `Translate` event is instantiated and informed of the translation to be done (by mean of its `dx` and `dy` slots) and sent to the `Translatable` interface. To do so the `Translatable` interface first needs to get the position of element to translate (i.e. the position of its holder): it will check that its holding XML node owns an interface able to answer the `Location` event, and get the result that this interface returns. Finally, the `Translatable` interface will merely change the XML tree of its holder by creating or updating its "transform" `XMLAttribute`, according to the current position of the element and the request mediated by the `Translate` event. Indeed, in SVG, the "transform" attribute permits to change the graphical position and angle of representation.

To declare an interface, the holding XML node needs to add name of the DOM interface as a "component" attribute value. The same remark holds for parameters for at least two reasons. A first reason is that an interface may require some parameters to work, including when the template is instantiated. For instance, a `DirectionAdjustable` interface needs to know what is the vector (as referenced by an SVG path) defining its direction. A second reason for the parameters to appear in the SVG document is that a save/

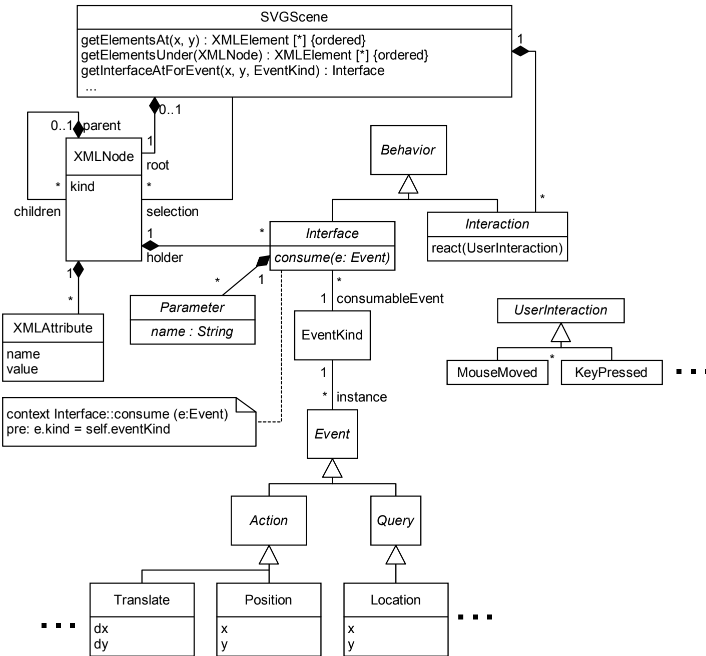


Figure 5.14: Conceptual Architecture of DOM Interfaces

load interaction should not interfere in behavior: context should be saved somewhere; we placed that context in the SVG document itself. Examples will follow in next section.

We implemented an observer pattern that makes it possible to be notified of executed actions. We also built an active query listener mechanism that periodically executes a query and notifies an observer in case the query returns a different result. This observer mechanism helped in designing the framework, and even behaviors. For instance, this mechanism helped in making containable nodes follow movements of their container node.

5.4.2.2 Predefined DOM Interfaces

As the implementation of the prototype evolved and as our experience became broader, we identified the following DOM interfaces necessary to define a graphical language.
Interfaces dedicated to position (only `SVGLocatable` DOM nodes may be handled):

- `OriginGettable`: finds the absolute position of an XML node in the scene (i.e. the absolute position of the point with relative coordinates (0,0)).
- `Locatable`: finds the coordinates of the holding node in the scene in terms of position, width and height.
- `BorderFindable`: finds a list of points in the scene drawing the outline of the holding XML node.

Interfaces dedicated to movement (only `SVGTransformable` DOM nodes may be handled):

- `Positionable`: places the XML node in a given absolute position
- `Translatable`: moves the XML node according to given a vector.
- `Stickable`: relates by mean of a `stickers` parameter to a collection of XML nodes that must be `Translatable`; those `stickers` nodes are moved at same time the holding XML node moves. `Stickable` is an example where the observer pattern described above was helpful regarding implementation.
- `DirectionAdjustable`: changes the orientation of the holding XML node according to a vector (at definition time, basic direction is defined as from left to right).
- `BorderSlidable`: makes the holding XML node able to move only following the outline of a component that is registered in an `attachedComponent` parameter. Note that if the `attachedComponent` XML node moves, the holder XML node must move accordingly.
- `Resizeable`: emphasizes the holding XML node of a given factor.

Interfaces dedicated to text editing (only `Text` XML nodes are handled):

- `CharacterHitable`: places a caret at a given index of the `Text` XML node; the holder XML node must declare the `OriginGettable` interface.
- `CharacterInstertable`: insert a given character at a given position of `Text` XML node; if the contents of the `Text` XML node is a CSVG constraint on an attribute, changes the attribute instead.
- `CharacterDeletable`: removes a given character either at the left or at the right of given position of the child `Text` XML node; if the contents of the `Text` XML node is a CSVG constraint on an attribute, changes the attribute instead.

Interfaces dedicated to scene management:

- `Highlightable`: makes the holding XML node more notable by emphasizing its outline.
- `Selectable`: places the holding XML node as part of the selection of the scene; the holding XML node must declare the `Highlightable` interface.
- `Orderable`: changes the order of the XML nodes in the scene by placing the holder XML node backward, forward, to front or to back.

Interfaces dedicated to connection-based languages:

- `Link`: XML nodes declaring this interface will just be holder for connections and should not be rendered in the scene. Typically, they are empty groups. `Link` interfaces just make the connection between XML nodes participating in the connection by declaring what is the connection XML node (by mean of the `curvedLine` parameter) and what are the elements represented at the ends of the connection (by mean of the `start` and `end` parameters). It may also declare handlers to change the route of the connection, but this should not be part of language designers' concerns.
- `CurvedLine`: such XML node may be placed as a connection for a link. They overload an eventual `Selectable` behavior by creating line handler in order to change its route (i.e. its intermediate points). Only SVG paths are supported for the moment. Original link is known by mean of the `parentLink` parameter.
- `Arrow`: such XML node may be placed at the `start` or the `end` of a link. It has same behavior as `DirectionAdjustable` and `BorderSlidable` interfaces with those differences that a move will also move the corresponding connection end. `position` parameter states whether the XML node is at the beginning or at the end of the connection. Interface changes the `Translatable` and `Positionable` behavior (by mean of an observer) by making the holding XML node part of the `stickers` of an eventual `BorderSlidable` XML node placed under.

Interfaces dedicated to containment:

- `Container`: such XML node may be able to contain XML nodes declaring the `Containable` interface. Contained elements are placed in the `contents` variable. Interface changes the `Translatable` and `Positionable` behavior (by mean of an observer) by making the contained nodes follow the same movement. This notion is independent from the notion of SVG group.
- `Contained`: such XML node may be part of the contents of an XML node declaring the `Container` interface. Container is placed in a `container` parameter. Interface changes the `Translatable` and `Positionable` behavior (by mean of an observer) by attaching or detaching the XML node from its container according to its target position.

We also defined a set of standard interactions to animate those interfaces (e.g. drag and drop raises a `Translate` event on the selection, or click on a selectable element triggers the `Selectable` behavior). Other interactions are able to exit (in case the escape key is pressed), or to save the document by flushing both the XML document and the model into an SVG and an XMI file, respectively.

An example, shown in figure 5.15, is the template definition for `SimpleState`. In this code snippet, the first element is an SVG group that declares the `Contained` and `Translatable` interfaces. The group contains a rectangle that is responsible for being the

```
<svg …>
  <g dpi:component="Translatable, Contained,…" …>
    <rect id="border_$$" dpi:component="OriginGettable,
BorderFindable, Stickable, …"…/>
     <text name="name" dpi:component="CharacterHitable,
CharacterInstertable, CharacterDeletable, …" …/>
     …
  </g>
</svg>
```

Figure 5.15: `SimpleState` SVG Template: Declaring DOM Components

outline of the state in that it declares the `BorderFindable` interface. The group also contains a text that is the placeholder for the name of the state. That is why this text must be editable and declares `CharacterHitable` (thus the `OriginGettable`), `CharacterInsertable` and `CharacterDeletable` interfaces.

Interfaces often need to be used together, following the example above of an editable text that needs to declare specific interfaces. The DoPIdom framework offers the `Component` class that is meant to group interfaces together.

```
<svg …>
  <g id="$$" dpi:component="Link"
     parameters=" Link(endArrow, end_$$),
                  Link(curvedLine, path_$$), …" …/>
  <path id="path_$$" dpi:component="CurvedLine,…"
     parameters="CurvedLine(parentLink, $$)" …/>
  <polygon id="end_$$"
     dpi:component="Translatable,
                    BorderSlidable,
                    DirectionAdjustable,
                    Arrow, …"
     parameters="Arrow(attachedLink, $$), …" …/>
  …
</svg>
```

Figure 5.16: `Transition` SVG Template: Declaring DOM Components

The example for the template for transitions is given in figure 5.16. The template starts with an empty group which is the holder for the `Link` interface. The link group is named `$$`, which will be replaced by a unique identifier at template instantiation time. It declares parameters in the `parameters` XML attribute following the following pattern:

```
parameters ::= parameter (',' parameter)*
```

```
parameter ::=<ParameterInterface> '(' <ParameterName> ','
parameterValue ')'

parameterValue ::= parameterSimpleValue | listValue

parameterSimpleValue ::= <XMLNodeId> | listValue | <IntegerValue>

listValue ::= parameterValue (',' parameterValue)*
```

A path with a `path_$$` identifier and a polygon with an `end_$$` identifier also appear in the template. They also declare parameters according to their respective rules. Note that the `end_$$` arrow end is not connected to any `BorderSlidable` capable XML node, meaning that the arrow end may be moved freely. For example, if the arrow end is moved on top of a simple state template instance, the `border_simplestate1` XML node (in case `$$` was replaced by `simplestate1`) will be automatically updated to integrate the arrow end in its `stickers` parameter.

In order for a display object to be a valid `GraphicalObject` (as introduced in section 5.3.4), template for a main display class must declare the following interfaces in one of its node or sub-node:

- `Locatable` in order to implement the `nearby` method,
- `BorderFindable` in order to implement the `connects` method,
- `BorderSlidable` in order to implement the `connects` method,
- `Container` in order to implement the `contains` method,
- and `Containable` in order to implement the `contains` method.

Moreover, constraint engine or model transformation responsible for abstract/concrete syntaxes consistency may need to force one of those methods to return true or false. To do so, we introduce forcing algorithms that require (1) the `Positionable` interface in addition to the previous ones in order to force the `nearby` method, and (2) the `Resizable` interface to force the `contains` method. In order to force the `connects` method, it is enough to declare interfaces `BorderSlidable`, as the interface integrates a routine to set attached components.

5.4.3 Relation With the Model

Although a language engineer may freely choose his/her own concrete syntax graph, synchronization between concrete syntax graph and representation is not completely automatable: mapping needs to be explicit. We propose here a mechanism to update concrete syntax graph (in model repository) according to changes in the SVG document (as an XML tree).

As explained in previous section, changes that may occur in the SVG scene are made possible by declaring interfaces chosen in a predefined library. The interfaces state what are the behavioral capabilities of their holding XML node, e.g. an SVG rectangle that is able to move because the node declares the `Translatable` interface. While some changes in the

scene don't alter the concrete syntax graph (e.g. changing the route of a path representing a transition), some others need to be reflected (e.g. moving a `Containable` SVG rectangle that represents a `SimpleState` into a `Container` SVG rectangle that represents the contents area of a `CompositeState`).

To solve problem of specifying this representation-to-concrete syntax graph data propagation, we propose an event-based approach in which reactions to events will impact the concrete syntax graph as a model. Such reactions may be written in a specialized language as KerMeta, Xion, EMF or JMI Java. To avoid confusion between the language under specification and the concrete syntax manipulation language, we will further refer to this latter language as the reaction language.

To define a reaction, different information need to be provided. First, reaction needs to be given an access to the model repository in which concrete syntax graph resides. We detail in section 5.4.3.1 how a repository can be handled. A reaction also needs to know what is the display object represented by the template instance. For instance, if a name is changed in a state representation, the corresponding reaction must be sure to alter the name of the correct state. To do so, we introduce in section 5.4.3.2 a mechanism of high-level variables (referring to elements of the concrete syntax) that can be maintained in template instances. We identify in section 5.4.3.3 what are the possible events that may define a reaction according to the DOM components that are defined in section 5.4.2.2. The representation also needs to be updated when the concrete syntax graph is changed, be it by as a side effect of a reaction or by another mean. We detail a mechanism to listen to concrete syntax graph changes in section 5.4.3.4.

5.4.3.1 Access to Repository

To keep our solution as generic as possible, we introduced notion of interpreters: to a given reaction language (name), we associate an interpreter. In the current implementation, we have only a Java interpreter (Koala [Hil02]), but one may imagine many others (e.g. KerMeta, Python, MTL). For a given language, the language engineer may associate three scripts per interpreter that will be executed at initialization time, when document is saved, and when document is loaded. Example for the usage of such script would be instantiating the model repository and creating base objects for the initialization script, saving an XMI file in save script, and loading an XMI file in load script. Variables initialized in the initialization script will be accessible to any further script (load, save, and reactions to events). In the example of JMI model repository, such further accessible variable would be the root `RefPackage` from which any model element may be created or found. Those initialization, load and save scripts are intended to manage access to model repository in a generic fashion: language engineer may freely decide which kind of model repository to use, what is the

exchange format, etc. Example for an initialization script in Java/Koala for the statechart language is given in figure 5.17.

```
metamodelFile = new java.io.File(languageDir,"sc.xml")
                                        .getAbsolutePath();
model = (sc.ScPackage)model.ModelFactory.getInstance()
             .getModel("MDR", new Object [] {metamodelFile});
sc = model.getStateMachine().createStateMachine();
top = model.getCompositeState().createCompositeState();
sc.setTop(top);
```

Figure 5.17: JMI Initialization Script

Script first loads in an MDR repository the statechart metamodel that should be an XMI file `sc.xml` to be found in the SVG templates folder (as given by the predefined `languageDir` variable). As it is a new model, a `StateMachine` instance must be created and a reference to a top `CompositeState` should be realized. Variables `metamodelFile`, `model`, `sc`, and `top` will be accessible in any further script of the same interpreter.

5.4.3.2 Maintaining High-Level Variables

Once an SVG template is instantiated, one or more display objects are supposed to be created in the concrete syntax model. To be able to update the display objects' slots, template instances need to reference them. Do do so, we propose to declare variables in the SVG template, that are filled at template instantiation by a dedicated initialization script, as a reaction to template instantiation. The script is of the same nature as initialization, load and save scripts described above with that difference they are declared within SVG templates.

```
<svg onCreation="{Java|
    s = model.getSimpleState().createSimpleState();
    s.setName(&quot;newState&quot;);
    top.getSubvertex().add(s);}" …>
<g id="$$" var_self="$s" …>
  <rect id="border_$$" var_self="$s"  …/>
  …
  <text id="name_$$" var_self="$s" …>newState</text>
</g>
</svg>
```

Figure 5.18: `SimpleState` SVG Template: Creation Reaction

We provide in figure 5.18 an example for `SimpleState` SVG template. At instantiation time, expression found in the template is executed with the declared interpreter. In the example above, we use the Java (Koala) interpreter, which means that we have access to the variables declared in the Java initialization script (i.e. `metamodelFile`, `model`, `sc`, and `top`). As we make use of a JMI repository, code makes use of the JMI API. `s` is a new variable that receives a new `SimpleState` instance with name `"newState"`, and contained by the top composite state. Note that we should have created here an `SVGSimpleState` and an `SVGText` rather than directly a `SimpleState`, but our prototype tool does not integrate a constraint solver. That is why we work here directly at abstract syntax level, which is bad design since concrete syntax defined here may not be seamlessly applied to another metamodel.

Any element of the template may then declare variables, following the examples of the `$$` group, the `border_$$` rectangle and the `name_$$` text. Variables are XML attributes of the form:

```
variable ::= var_<VariableName> '=' '"' variableValue '"'

variableValue ::= boolean|double|string|integer|object|collection

boolean ::= 'Boolean' '(' ('true'|'false') ')'

double ::= 'Double' '(' <DoubleValue> ')'

string ::= 'String' '(' <StringValue> ')'

integer ::= 'Integer' '(' <IntegerValue> ')'

object ::= 'Object' '(' <ObjectID> ')'

collection ::= ('Set'|'Bag'|'Serquence'|'OrderedSet')
                 '(' variableValue (',' variableValue)* ')'
```

Boolean, double, and integer values have the same form as those of the Java language. Object identifier is a unique identifier referencing an object in the model. Collections may be nested. However, in the example above, variables `var_self` do not follow the above format. Indeed, their value is `$s`. This value is replaced in the template instance by the value found in the exiting context of the creation script. In our example, exiting context actually contains an `s` variable referencing the newly created simple state: any `$s` occurrence will be replaced by `Object(123456)` (assuming that the unique identifier of the new simple state is 123456).

5.4.3.3 Reactions to Events

A change in the model is triggered by a change in the diagram. Possible changes that may occur in the diagram are all described by mean of DOM interfaces. We propose to enhance interfaces with the possibility to define reactions within the SVG templates that are triggered each time a behavior is performed. Such reaction may be described using an action

language following the same approach as before, when it came to defining initialization, load and save scripts, and when we stated what is the action to take at template instantiation time. We call such scripts reaction scripts.

Major difference with initialization, save and load scripts described above is that not all graphical elements define the same reactions following the example of the creation script. Possible reactions depend on interfaces that an XML node declares. Thus, interfaces may trigger a certain number of reactions, and a reaction is related to a given interface. Note that only action interfaces may trigger reactions. In the example of the statechart language, the `border` part of the `SimpleState` template declares the `Stickable` interface. When a `BorderSlidable` XML node (e.g. an end of a transition template instance) actually sticks to that `border`, an `onStick` reaction as found in an `onStick` XML attribute of the border is performed.

For the reaction to be able to perform its task properly, a context needs to be passed. In the previous example, when a `BorderSlidable` XML node sticks to the `border` of a `SimpleState`, the action should update the corresponding `StateVertex::incoming` reference (according to metamodel of figure 4.8 on page 78). To do so, in addition to an access to the repository, the triggered reaction needs to know what are the implied simple state, transition, and whether the stuck end is actually the start or the end of the transition symbol (for the action not to update the `StateVertex::outgoing` reference instead). This problem is solved by constructing a specific context depending on the triggered event, in addition to context constructed by the initialization script. First, each high-level variable maintained by the XML node that declares the interface is added, and may be altered by the scripts. Such alteration requires to update the XML tree once the script is executed. Moreover, in the example of the `Stickable::onStick` event, the context depends not only on the stickable object (the `border`), but also on the sticking object, which is an XML node. As an XML node may not be handled in a straightforward way by an action language, we add to the context each high-level variable (as described in previous section) that is maintained by this latter node. Those variables are presented in the context either in an encapsulated or a prefixed fashion in order to avoid any confusion with high-level variables of the holding node. In our example where the end of a transition sticks to the border of a simple state, the triggered reaction defined by the border of the simple state will be executed in a context that contains `stuckComponent_self : Transition` and `stuckComponent_isSource : Boolean` variables because the end of a transition maintains `self : Transiton` and `isSource : Boolean` high-level variables. This

allows to improve the SimpleState template for the onStick event using the Java/ Koala - JMI language as shown in figure 5.19.

```
<svg …>
<g …>
  <rect id="border_$$" dpi:component="Stickable, …"
        var_self="$s" onStick="{Java|
if ((stuckComponent_isSource).booleanValue()){
      self.getOutgoing().add(stuckComponent_self);
} else {
      self.getIncoming().add(stuckComponent_self);};
}
        }"…/>
…
</g>
</svg>
```

Figure 5.19: SimpleState SVG Template: onStick Reaction

We identified the following possible reactions for action interfaces defined in previous section; specific variables added to context of triggered actions are also given:

- Positionable and Translatable: *onPosition* with posX and posY : Double variables for the new coordinates of the XML node.
- Stickable: *onStick* in case of new stick and *onUnstick* in case of stick removal, with a stuckComponent_/unStuckComponent_ prefixed set of variables for high-level variables maintained by the (un)stuck XML node.
- DirectionAdjustable: *onRotate* with an alpha : Double variable that indicates the new angle in radians.
- Resizeable: *onResize* with sizeX and sizeY : Double variables for the new size of the element.
- CharacterInstertable and CharacterDeletable: *onChange* with a content : String variable stating the new value of the text node.
- Container: *onContained* with a containedComponents : Set(Set(Sequence(Object))) variable that contains, for each contained XML node, each maintained high-level variable as a sequence containing name in first position and value in second position.
- Contained: *onContained* with a containerComponent_ prefixed set of variables for high-level variables maintained by the new container.

5.4.3.4 Reacting to Model Changes

Pulling up information about changes from representation to model is not enough to represent a model: the model may also change either externally (e.g. a refactoring model transformation) or as a side effect (e.g. in case an information of the abstract syntax is represented more that once - a change in one representation should be reflected in the other representations). We introduce here a technique inspired from CSVG to push down model changes into representation.

If constraints of the specification part are enough to state general diagram constraints (see section 5.3.3), template instances need to track slot value of their display object. We propose to copy those values in the SVG document, and to place listeners on the model so that those values are updated automatically in the SVG document. CSVG constraints may then listen to those values and force the template instance to display them in an appropriate way.

To declare such copy-and-listen mechanism, we introduce a new `update` XML node following the example of the CSVG `constraint` XML node: such tag is a child of the XML node owning the XML attribute in which value from the model will be stored. The `attributeName` XML attribute states in which attribute to store the value. Two more XML attributes of the `update` XML node state what is the display object to observe (`var_source` - a high-level variable that is probably an object) and what is the slot to observe (`slot`). Moreover, when the XML attribute storing the value is changed, the model is updated accordingly.

We finalize in figure 5.20 the definition for the simple state template. The `Text` XML node does not contain text to display. Instead, a `tval` CSVG constraint states that the text to display should be found in the attribute `value` of the `Text` XML node by mean of an XPath expression. An `updater` XML node is also defined to maintain the `value` attribute to the actual value of the `name` slot of the model element `s`, as created at template instantiation.

Once the template is instantiated, the `onCreation` reaction is triggered. Among other things, its creates a new `SimpleState` model element in the repository and stores it into the `s` variable. The `name` slot of `s` is also set to `newState`. The `updater` then comes into the play by reading the `name` slot of `s`, and places the result in the `value` attribute of its owning `text` node. The `tval` CSVG constraint, which is listening on that attribute, makes the `text` display the attribute's value. Finally, at then end of template instance, the `text` node displays `newState` even though not directly written in plain SVG.

As text is an XML node declaring the `CharacterInstertable` interface, a user interaction may change the value of the displayed text. As the contents of the text is a `tval` constraint following the `value` attribute, the new text will be placed in that attribute. As the `value` has changed, the `name` slot of the `$s` model element is changed accordingly, thus avoiding to write an `onChange` action event.

```
<svg onCreation="{Java|
          s = model.getSimpleState().createSimpleState();
          s.setName(&quot;newState&quot;);…"…>
  <g …>
   …
   <text id="name_$$" var_self="$s"
       value=""
       onChange="{Java| self.setName(content);}"
       dpi:component="CharacterHitable, CharacterInstertable,
                                       CharacterDeletable, …"…>
     <c:tval value="../@value" />
     <m:updater  attributeName="value"
                 var_source="$s"
                 slot="name" />
     …
   </text>
  </g>
</svg>
```

Figure 5.20: `SimpleState` SVG Template: Updater

Note that this technique is only meant to maintain an attribute value in the SVG document and a slot value in the model synchronized. More complex relations still require a constraint solver at model level and event actions at XML level.

Automatic synchronization between models and a data structure requires to have a specialized knowledge of the model repository technology. So far, our prototype implementation only supports JMI-based repositories, but other technologies may be easily integrated. Adopted technology is selected according to the XML namespace of the `update` tag thus making possible to mix, bridge and move technologies for a same document.

5.4.4 Prototype Implementation Overview

In this section we take a rapid tour of the prototype implementation for the realization step of graphical concrete syntax specifications. If we already had the opportunity to draw its main features all along section 5.4, we describe here how to use of the prototype.

The prototype is an evolution of the DopiDOM framework, and is based on a version of Batik adapted to handle CSVG constraints. We improved DopiDOM by refactoring the interactors system (by applying a composite pattern), and by implementing an event system. The event system listens to actions, or actively executes given queries firing an event each time one of the queries returns a result that is different from last execution.

A language specification is read from a directory that contains at least the SVG templates: the tool finds the templates and creates a toolbar that contains buttons for instantiating each one of the templates. The names of the files (`<name>.svg`) are used in the toolbar to identify the templates. To instantiate a template, the system engineer clicks on the button of the toolbar corresponding to the template to instantiate, and then clicks on the scene at the place the new template instance should appear. The tool then integrates the new copy of the template in the scene. If the instanciated declares either a `Positionable` or a `Translatable` interface, the new template instance is moved to the indicated position. A default background for the scene is available with a default interactor specified. However, one can provide her/his own background by an `empty.svg` file to be found in the language specification directory.

The language specification directory can also contain initialization, load or save scripts. The scripts should be placed in the directory with names `init.<Language>`, `load.<Language>`, and `save.<Language>`, respectively. `Language` corresponds to the name of the language interpreter to be used. Current implementation manages only the Koala dynamic Java interpreter, whose name is either `Java` or `Koala`. The scripts are intended to contain instructions about model repository handling, for instance JMI repositories. To help in writing the scripts, the prototype implementation integrates an helper function to setup an MDR repository by providing as argument the location of an XMI file containing a MOF metamodel. Other helper functions manage saving and loading of models from/into an MDR repository. The helper functions are declared in the `ModelFactory` class of the tool, as shown by figure 5.17. In addition, the tool loads any java library (`jar` file) found in the directory. This notably allows to place a `jar` file containing all the JMI interfaces of the metamodel in the language specification directory to help in manipulating model elements, in inititialization, load and save scripts, and in reactions to events. Without this, the JMI expression of figure 5.18 `s = model.getSimpleState().createSimpleState()` would not be possible, since the `getSimpleState` and `createSimpleState` methods are declared in the JMI interfaces that are specific to the statechart metamodel. Nevertheless, one could use JMI reflective interfaces instead, by writing `model.refClass("SimpleState").refCreateInstance(null)`.

We show here how the prototype behaves when the language directory contains the definition for the statechart language as described above. The files in the language definition directory are the following:

- `empty.svg` that overloads the default background file,
- `init.Java` that is the Java/Koala initialization script,
- `load.Java` and `save.Java` that are the Java/Koala load and save scripts,
- `sc.xml` that is the statechart metamodel in XMI as required by the `init.Java` script,

- `sc-jmi.jar` that is the Java library containing the JMI interfaces for the statechart metamodel,
- `simple_state.svg`, `composite_state.svg`, `initial_state.svg`, `final_state.svg`, and `transition.svg` that are the SVG templates.

When the tool is launched, system engineer must indicate the language definition directory. Tool then loads the `empty.svg` file and creates the buttons responsible for template instantiation. The `init.Java` script is executed, which creates a new model from the `sc.xml` metamodel, and instantiates the main state machine and its top state (according to definition of figure 5.17). Figure 5.21 shows the initial view, in which no SVG template was instantiated yet.

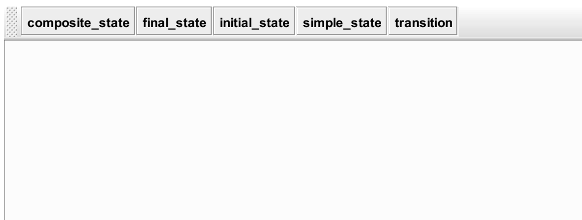


Figure 5.21: Initial View of the Prototye Implementation

In order to instantiate a new template, system engineer first clicks on the button corresponding to the template s/he desires to instantiate, and then, s/he clicks on the scene at the location the template instance should appear. If the button corresponding to the simple state template was clicked, the resulting scene should look like that one of figure 5.22. Note that

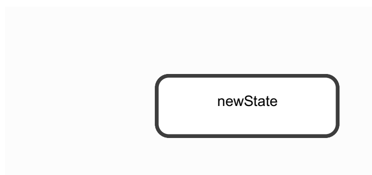


Figure 5.22: Simple State Template Instantiated

the template instance actually appears at the desired place because the template declares the `Translatable` interface. The new template instance is automatically selected because the template declares the `Selectable` interface. The `onCreation` reaction of the template is executed, which creates a new `SimpleState` model element in the model according to

figure 5.18. Moreover, because the background declares the `Container` interface, and because the simple state template declares the `Contained` interface, the `onContained` event is triggered on the background so that the new `SimpleState` model element is placed in the `top` state, as must be declared in the `empty.svg` file (see figure 5.24 ①).

At that moment, a save action would produce the SVG and the XMI files as shown in figure 5.23 and figure 5.24, respectively. The SVG document is a copy of `empty.svg`

```
<?xml version='1.0' standalone='yes'?>
<svg …>
<rect  id="background" dpi:component="Locatable,Container"
       fill="lightyellow" width="800"  height="600"
       parameters="Container(contents, c529)"
       onContained="{Java| ….setContainer(top);…}"/>①

<g id="5c529" parameters="Containable(container, background)"
dpi:component="Contained, Hilightable, Selectable, …"
transform="matrix(1.0 0.0 0.0 1.0 309.0 134.0)"
var_self="Object(a3)">
  <rect id="border_5c529" dpi:component="BorderFindable,
Stickable, …" var_self="Object(a3)" …/>
  …
  <text id="name_5c529" value="newState" var_self="Object(a3)"
…><tval value="../@value"/><updater attributeName="value"
slot="name" var_source="Object(a3)"/></text>
</g>

<g dpi:component="Pointer"/>
</svg>
```

Figure 5.23: SVG File for a Single Simple State Instance

(shown in italic in the figure) to which was inserted the simple state template instance (in standard font in the figure). The template instance is a copy of the template with some differences that are underlined in the figure. First, the `$$` occurrences are replaced by an identifier (`5c529` in the example). Moreover, the high-level variables, which are initialized to `$s` in the template, are holding a model element reference instead (`Object(a3)` in the example). The `Object(a3)` references correspond to the model element that was created and placed in the `s` variable by the `onCreation` reaction script of the simple state template. Finally, a `translation` attribute appeared, since it is the mean used by the `Translatable` interface to move an SVG node in the scene. The generated XMI file contains the abstract model, i.e. a state machine (reference `a1`) and its top state (reference `a2`) as created at language loading by the initialization script. The top state contains a simple state (refer-

```
<?xml version = '1.0' encoding = 'windows-1252' ?>
<XMI xmi.version = '1.2' timestamp = 'Tue Jul 10 12:11:50 CEST
2007'>
  <XMI.header>
    <XMI.documentation>
      <XMI.exporter>Netbeans XMI Writer</XMI.exporter>
      <XMI.exporterVersion>1.0</XMI.exporterVersion>
    </XMI.documentation>
  </XMI.header>
  <XMI.content>
    <sc.StateMachine xmi.id = 'a1'>
      <sc.StateMachine.top>
        <sc.CompositeState xmi.id = 'a2'>
          <sc.CompositeState.subvertex>
            <sc.SimpleState xmi.id = 'a3' name = 'newState'/>
          </sc.CompositeState.subvertex>
        </sc.CompositeState>
      </sc.StateMachine.top>
    </sc.StateMachine>
  </XMI.content>
</XMI>
```

Figure 5.24: XMI File for a Single Simple State Instance

ence a3) that was created at template instantiation and that is referenced in the simple state template instance.

As a final example, we show in figure 5.25 the door state machine of figure 5.3 as can be drawn using the prototype implementation.

5.5 Comparison with Other Approaches

The problem of defining a graphical concrete syntax on the top of a metamodel has been addressed already by numerous authors.

An important remark is that UML Diagram Interchange [ADG+06] is not a mean to define a graphical language. UML-DI is merely a mean to exchange diagrams of a well-agreed language as already said in section 2.1.3.4 page 24.

Some approaches, like XMF [CESW05], argue that the concrete syntax involves a representation language. An example of such languages is SVG, but the XMF framework provides its own graphical language by the mean of a representation metamodel with well defined semantics. Here, semantics correspond to a graphical representation rendering.

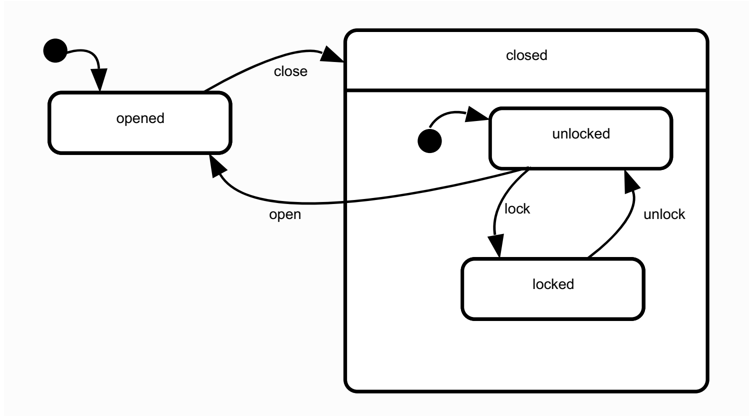


Figure 5.25: The Door State Machine Described in the Tool

Thus, to define the representation of a given language whose abstract syntax is given by a metamodel, it is enough to define a model transformation between the metamodel of the language and the representation metamodel. Here, language specialist needs not only meta-modeling skills but also needs a deep knowledge of that specific metamodel together with model transformation skills. We show in appendix A that manipulating metamodels in model transformation is not that an easy task. In contrary to XMF, we propose here to let language designers specify their own graphical «semantically rich» metamodel, supplying them with a comprehensive way to specify that graphical semantics.

Another kind of approach is taken by most meta-CASE tools, like GME [ESB04], DOME [Hon92], MetaCASE [Poh03], or AToM^3 [dLV02]. The principle is to define a representation template for each metaclass in the abstract syntax. A template includes a set of representation language constructs, as instances of the representation language metamodel, together with some holes that represent the variants in the depiction. Again, each one of these tools impose its own graphical language. When a model element has to be represented, the holes are replaced depending on the information from the model and the representation is placed on a diagram. Unfortunately, while most of these tools provide a constraint language that can be used to impose restrictions on the abstract syntax, they do not provide a programmatic access to the concrete syntax. A notable exception is AToM^3, which accepts to represent the variations in the icons using imperative constructs written in the Python language. However, also in AToM^3 the definition of the concrete syntax is done at a much lower level as our approach which uses OCL as the main language to specify the

concrete syntax. Again, in those approaches, reusability is a problem and representation has to be stated from scratch for each abstract syntax.

Graph-grammar based visual language definitions (as Triple-Graph-Grammar [Sch94], GenGED [Bar98]) are constructive and aim at finding a derivation for a given diagram. In addition to rules, GenGED offers the possibility to attach constraints to the concrete syntax classes (called type graph nodes in the GenGED terminology), but the purpose of the constraints is merely the computation of a possible layout for a diagram. The language definition itself is still based on graph grammar rules (see [BEdLT04] for the GenGED-definition of the same StateChart fragment as we used here for illustration).

In the last year, graphical concrete syntax definition has gained interest in the modeling community and important projects has emerged. GMF [Ecl06], TopCased [VPF+06] and DSL tools [GSCK04] are developed by the IBM/Eclipse community, the CNRT French national research network and Microsoft, respectively. These metamodel-based solutions were developed at the same period the present work was done. The three solutions propose semantically rich metamodels for connection based language representation and metamodels to map abstract syntax, representation. A third metamodel may also provide a mean to define user interactions. Main advantages of these solutions over XMF is that they avoid the need to define complex model transformations. Nevertheless, they do not offer good reusability regarding language definition and choice for model repository. Moreover, philosophy is limited to connection based languages. However, they remain interesting solutions compared to the realization step we propose.

Thanks to the DoPIdom architecture, clearly specified user interactions are one of the strengths of our approach: no alteration possibility is left implicit. However, we can compare our approach to that one of Tiger [EEHT05] which uses graph grammars to state any possible change in the diagram, e.g. a move. AToM^3 [BGdL06] offers an architecture dedicated to manage user interactions. Instead of using graph grammars, they propose to use QVT model transformations. They also use two levels of user interactions (user interactions vs. visual actions). However, the approach may not be seamlessly applied to XML trees.

5.6 Conclusion

We propose here a way to specify a graphical concrete syntax. The philosophy of the approach is to let the language designer free to choose the concrete syntax graph s/he believes to best fit his/her needs, using the well-known technique of metamodeling to state display classes. If we made use of MOF for the examples regarding metamodeling facilities, principle may be exported to any other object-oriented metamodeling language.

Defining concrete syntax graph is clearly not enough to define graphical concrete syntax. Problems remaining to solve are the following: how the concrete syntax relates to the

abstract syntax, what are the well-formedness rules of the concrete syntax (e.g. spatial relationships), and what is the actual aspect of each element of the concrete syntax.

Following the example of triple graph grammar, this time adapted to metamodeling, an intermediate metamodel accompanied by constraints, in-between display classes and abstract syntax, can handle relations between abstract and concrete syntax. This approach requires an (incremental multi-way) constraint solver that will support abstract/concrete syntax synchronization at run-time, i.e. when a model is created. Nevertheless, if such kind of constraint solver exist [BLP04, TJ07], one may use a dedicated mapping language (e.g. [LHBJ05]), or define a model transformation that should satisfy the constraints to improve interactive efficiency. In that case, the (bidirectional and incremental) transformation should be verified against the defined constraints. The main advantage of defining such mediator in-between abstract and concrete syntax is reusability. On one hand, one may define many different concrete syntaxes for a same language. On the other hand, one may reuse a given concrete syntax and port it to another language just by rewriting association constraints.

Compared to other approaches, neither we stick to a semantically rich metamodel which already offers a predefined representation meaning, nor we force the language designer to imagine connection based languages. Indeed, we do not make any distinction between nodes and edges. Instead, we propose to use again constraints to define spatial relationships, and which has long proven to be a comprehensive mean to specify graphical layouts of interactive systems [Sut63, BMSX97]. Only requirement is that display objects have knowledge of their relative spatial situations, as expressed by the `GraphicalObject` interface.

To realize concretely all the promises of the concrete syntax specification, and to provide concrete appearance, we also propose a technique based on SVG, the XML based open standard for vector graphics. We thus benefit from the expertise of a specialized technological space, avoiding to develop and maintain a new technology for graphics description. All "main" display classes, i.e. classes directly connected to a display manager class, have representation given by an SVG file which also contains definition for "sub-"display classes. When a display object is created, the SVG file of the display class is copied on the diagram scene, which is an SVG document, with the only change of introducing an unique identifier to make a proper relation to the corresponding display object. In the meantime, display objects and display manager are instantiated in a model repository.

One may also want to make his/her language editable. As SVG is an XML dialect, an SVG document may be controlled by DOM technology. We exploited such aspect to manage user interactions. We propose a set of predefined DOM components, with the help of the DoPIdom framework, to declare what are the interaction possibilities for each SVG fragment. Examples of such SVG components are move, resize, or select components.

For synchronizing the diagram scene with the display objects (and thus the abstract syntax), we also introduced in the SVG fragments the possibility to integrate code to be executed on the (display) model as indirect reactions to user interactions. Code may be given in

KerMeta, Xion, MTL, or JMI java code. Moreover, for the slots of display object to be accessible in the template, we propose to place a synchronized copy of their values in the template instances.

As SVG template instances are made dynamic thanks to predefined DOM components, we propose to use the CSVG technology to define internal layout of elements. CSVG may also be used to adapt value of display objects's slots to representation needs.

However, there is still place for improvement through experimentation. We should develop more languages and improve prototype implementation, which still lacks integration of a constraint solver. Moreover, we believe that this is only a first step in abstracting task of graphical model language engineering, and one may imagine yet more abstract ways, less flexible but more agile, as experience in rapid building of graphical modeling language broadens.

Both specification and realization steps of the approach we propose may be performed using different technologies than we propose. For instance, specification step could be accomplished by (reversible and incremental) model transformations, and realization step by a GMF or a Topcased specification.

Chapter 6:
Conclusion

6.1 Summary

Language Driven Engineering (LDE - e.g. [CESW05]) is an approach to software development that composes characteristics of both Model Driven Engineering (MDE - [Ken02]) and related techniques, which are means to organize abstractions, and Domain Specific Modeling (DSM - [Poh03]), which is a mean to tailor the modeling activity to a specific domain. While MDE promotes the massive use of models that reside at various levels of abstraction, DSM advocates that each one of these models should be expressed in the most appropriate modeling language, depending on the domain of the model and on the abstraction level. When MDE and DSM are used together, one can fear of a proliferation of modeling languages, even for a single software development project: the complexity is moved from software engineering to language engineering.

The use of tailored languages for engineering software was already addressed in the 1980s by CASE tools. However, CASE tools failed to be adopted because of their recurrent lack of quality: it is not economically viable to support comprehensive tooling for DSM languages that are limited in scope and in the size of their user community [Iiv96]. Indeed, to support a language, one needs to develop a development environment (textual IDE or graphical modeling tool), code generators, and bridges to other tools or languages (e.g. transformations or import/export facilities), without forgetting crosscutting concerns such as concurrent development or maintainability. For DSM to overcome this problem, language engineering facilities are required.

To define a language, language engineers need to provide an abstract syntax and semantics. It is now widely accepted that abstract syntax of modeling languages can be provided under the form of a metamodel [AK02], while semantics is still subject to discussions [HR04]. Moreover, to interface abstract syntax with the stakeholders of a development project, language engineers need to develop one or more concrete syntaxes for the models to be created and read in a human-friendly fashion. A recurrent problem with concrete syntax is that it is often specified using natural languages, which is a technique that is inevitably subject to ambiguous interpretations [ABF+06]. If other means exist to define concrete syntax (e.g. graph grammars [EEKR99] or compiler compilers [ALSU06]), they are often inconsistent with a metamodeling approach. We proposed in this document means for clearly specifying concrete syntax for languages whose abstract syntax is already known by mean of a metamodel.

To better introduce LDE, we presented in chapter 3 Netsilon, a web application modeler we developed between 2000 and 2001. Netsilon introduces new languages for modeling web applications. It defines three main modeling languages. The *business model* captures static data using a formalism very close to UML class diagrams, which made it possible to adapt (in an ad-hoc way) a UML modeling tool regarding tool support. The *presentation model* is a set of textual file templates. The *hypertext model* is a completely new modeling language that organizes the composition and links between the template files of the presentation model, according to data modeled by the business model. Netsilon is thus a very flexible template engine capable of representing, in textual form, data organized into a model. We showed that is was possible to generate SVG code [JN05] out of a model (i.e. an instance of business model) using SVG templates in the presentation model, with that price of adding data dedicated to representation. The fact that we used SVG as the generated language made it possible to depict the model graphically. Moreover, by adding code using the DOM API [HHW+04], we made it possible to interact with the diagram.

Inspired by the above results, we drew our solutions to concrete syntax specification both for textual and graphical syntaxes.

The solution we presented in chapter 4 for *textual* concrete syntax specification takes the form of a specification language that we defined by a metamodel and semantics in natural language. The metamodel for specifying textual concrete syntaxes is inspired from both the concepts of Netsilon's hypertext model, and of extended Backus-Naur form [Int01] to text structure specification. Typically, specifications for code generation (to represent a model) and text analysis (to construct a model), are separated from each other, and need to be consistent and maintained in parallel. This is not the case for a unique specification that is sufficient for automatic tools to automate text analysis (i.e. analyzing a text to produce a model) and synthesis (i.e. to produce text from a model). We proposed such kind of specification language that was successfully applied to provide textual concrete syntax for different languages. As examples, we showed specifications for a simple language of data structures, and for a simplified textual version of the statecharts language. The approach can be applied to supply our textual concrete syntax specification language with a concrete syntax.

The solution we presented in chapter 5 to *graphical* concrete syntax specification took the form of a two-step process: specification and realization. As such, abstract syntax is not directly represented by being directly mapped to a predefined representation language, as it is commonly the case using tools like GMF [Ecl06] or XMF [CESW05]. Our approach avoids pollution of concrete syntax by abstract syntax, and pollution of abstract syntax by concrete syntax. In the *specification* step, a language engineer specifies structure of the concrete syntax by defining a metamodel that organizes the representation data. The metamodel for concrete syntax is complemented by OCL constraints to state spatial relationships. For the concrete syntax graph to be synchronized with the abstract syntax graph, we introduced a mapping model complemented with OCL constraints to manage relation-

ships between the two syntaxes. The main interest of this mapping model is that the abstract and concrete syntaxes do not need any dependency between each other. The *realization* step deals with icon representation and possible human interactions (i.e. how to edit a model via its representation). We proposed to define icons using SVG templates following example of chapter 3. We also proposed a predefined collection of DOM components to specify possible user interactions. DOM components are associated to events that can impact the concrete syntax graph according to specific code as embedded in SVG templates. Note that the collection of proposed DOM components may easily be extended in the future.

We proposed two approaches to model textual and graphical concrete syntaxes for modeling languages, respectively. These approaches require means to define mapping between metamodel and text structures, model matching, high-level visual constraint, icon templates, and user interactions able to impact both a model and its graphical representation. We provided solutions for each one of these points, that were validated by prototype tools (except regarding constraint solving). These solutions look likely to evolve while our experience in building modeling language improves, and may be replaced individually by comparable solutions without breaking the overall process (e.g. TCS [JBK06], triple graph grammars [Sch94], Cassowary [BMSX97], GMF [Ecl06], and model transformations [EEHT05], respectively). Further activities may also include improvements of prototype implementations, for instance by providing support for OCL constraint solving. Another possible field of research is to find solutions to improve homogeneity between textual and graphical concrete syntaxes. For instance, we could determine whether it is helpful and possible to apply a two-step process (specification and realization) for textual concrete syntax specification. We could also develop automatic model recognition from legacy diagrams, according to concrete syntax.

6.2 «A language that is used will be changed» [Leh80]

The solutions and approaches we presented in this document are only small (and perfectible) steps towards agile language driven engineering. To define a language, one can provide his/her abstract syntax under the form of a metamodel, semantics possibly under the form proposed in [MB06], and concrete syntax according to this document. Nevertheless defining a language is not enough. As underlined in chapter 1, languages are likely to become the cornerstone of software development projects, and developing languages and methods from scratch to help engineering a small set of software projects is not an approach that can scale.

For LDE to be successful, we believe that a next step should be language composition, following the example of Component Based Software Development (CBSE) [Szy02]. Indeed, in CBSE, software systems typically use software components implemented and sold by third-party vendors (so called “off the shelf” components). Regarding language

engineering, this would mean to be able to reuse other languages, including tool support, to build a new language. An example of such composition of language could be Xion, as introduced in chapter 3. Indeed, instead of building Xion from scratch, it would have been more agile to compose the OCL and Java languages, and to reuse an existing OCL to SQL compiler and a Java parser. To do so, we need facilities to select the parts of OCL and Java that are of interest in the context of defining an action language (i.e. Xion) for already customized UML class diagrams (i.e. the Netsilon's business model). Composition of languages looks harder to achieve than composition of software, since it introduces additional problems. On one hand, languages should be developed for reuse, that is in a modular way, but interfaces may not be of same nature as software interfaces since they need to deal with abstract syntax, semantics, and concrete syntaxes. On the other hand, it must be possible to customize reusable languages. For instance, a reusable specification for OCL [ABF+06] would ease the task of porting OCL to various versions of UML, MOF (avoiding constructs loosing their meaning such as pre or post conditions), or even relational databases. It should also be possible to add concepts to OCL at abstract syntax, concrete syntax and semantics levels (e.g. extensions proposed in [ZG03]). Solutions to complex reusability and variants specification can be found in the product family engineering community [Bos00].

Customizing a metamodel can be achieved by higher-order hierarchies [Ern03], for instance using constructs such as package merge as introduced by UML 2.0 [AAB+07]. To a smaller extent, metamodels can also be customized by the profile tag (or decoration) mechanism. We propose in appendix an example of adapting an LDE process using a tag mechanism, which is assumed to be available in the modeling languages that play a role in the process (at abstract and concrete syntax levels). Principle is to add new concepts in an intermediate abstraction layer, and to define corresponding interactive model transformations. A problem is that in-place automatic improvements below (i.e. after) the intermediate abstraction level need to be enhanced to cope with the inserted concepts. To do so, we show in a second part how to tailor a model transformation (which represents an improvement) by adding the aspect mechanism [KLM+97] to the MTL transformation language. Unfortunately, the appendix only proposes specific examples to illustrate the problem of language adaptability. Further work and experiments are necessary to provide a more broad in scope solution.

Appendix A:
Applying and Customizing LDE

LDE can be seen as a software development process definition facility (see section 2.1.1 on page 11). This appendix presents an example for the definition and the application of an LDE process when it comes to reuse or extend existing languages or improvements. In a first part, after introducing key technologies, we present an example that follows the LDE principles. We also illustrate reusability of a modeling language by using tag mechanisms such as UML profiles. In a second part, we will improve an LDE methodology through model transformation customization. The technique we used follows principles of aspects oriented software development, that we adapt to the MTL model transformation language. For MTL to integrate an aspect-oriented programming facility, we defined an extension using a tag mechanism very similar to UML profiles. However, extensions we use here alter only abstract syntax, semantics being left to an interpretation engine and concrete syntax being already handled by an existing tag or profile mechanism.

The following were partly published in the Model Driven Architecture: Foundations and Applications (MDAFA 2004) workshop [SFS04b] and in the 8th international IEEE Enterprise Distributed Object Computing (EDOC) conference held in 2004 [SFS04a].

A.1 Technologies

We first introduce here concepts of UML profiles and the MTL model transformation language that are necessary for the rest of the appendix.

A.1.1 UML Profiles

UML (see section 2.1.2.5 on page 17) provides a standard set of extension mechanisms, including stereotypes, tag definitions, tagged values, and constraints. These mechanisms are used to specify how UML model elements can be customized and extended with new semantics. A coherent set of such extensions, defined for a specific domain or purpose, constitutes a UML *profile*. The *stereotype* concept provides a way of branding (classifying) model elements. A stereotype creates a “virtual” UML metaclass by extending an existing UML metaclass with new meta-attributes and additional semantics. Meta-attributes are specified as *tag definitions*, which introduce new kinds of properties that may be attached to model elements. The actual properties of individual model elements are specified using *tagged values*. Simplifying, we could also say that a tag definition specifies the tagged val-

ues that can be attached to a kind of model element. The *constraint* concept allows new semantics to be specified linguistically for a model element. The language used can be a special constraint language (such as OCL [ABF+06]), a programming language, mathematical notation, or even natural language.

A certain number of UML profiles have already been defined, either for generic purposes, such as the UML profile for SPEM, or to deal with specific middleware technologies, such as the CORBA profile [DDG+02], and the list could very well continue. A consensus appears to say that profile is a comprehensive mechanism to adapt UML to domain specificities when gap is not important. Otherwise, one should seriously consider using another language.

A.1.2 MTL

Among all model transformation languages presented in section 2.1.3.2 on page 22, we made an extensive use of the Model Transformation Language (MTL) [fRiCSI05, VJ04]. MTL, is an imperative model transformation language with a textual concrete syntax we developed at INRIA in 2003. MTL takes its roots in the Xion action language (presented in chapter 3) and is source of inspiration for the KerMeta project [MFV+05, Fle06]. The definition of the MTL language is an example of traditional language engineering: abstract syntax is formalized by a metamodel, and concrete syntax by a compiler compiler specification (from which was generated a parser - see section 2.2.1) complemented with a hand-written visitor constructing the transformation model from concrete syntax tree.

MTL was developed at the time QVT standard was under early study, when many radically different solutions were proposed. Still, even though QVT is now an almost published standard, many different solutions exist together and adaptability to changes is still an issue while new paradigms continue to emerge. The idea of MTL is to provide *all* model transformation facilities, including the possibility to transform MTL transformations. This makes it possible for any future evolution of transformation languages to be mapped to an MTL transformation by means of an MTL transformation. This *pivot* approach has already been validated. The MTL itself is developed according to a bootstrapped approach: a simple language, called BasicMTL [Voj04], provides the most important facilities, such as classes or attributes, and new facilities are added by extending the abstract syntax and by making a transformation from the extended to the initial syntax, always relying in this way on the small “kernel” of BasicMTL. As an example, the n-ary navigable associations between classes have been added following such an approach.

As said before, MTL is an object-oriented imperative language for model transformations. Therefore, MTL transformations are defined as programs in terms of classes, methods, attributes, etc. In order not to confuse these MTL constructs with the ones that the manipulated models may contain, we will further on refer to them as MTL classes, MTL methods, MTL attributes, and so on. A special entry point, the `main` method, has to be

defined for each MTL transformation. Pieces of MTL transformations are organized in MTL *libraries*, each library being in addition responsible for holding models. Each such model can either be a collection of instances of MTL classes from an MTL library, or a collection of model elements inside a repository.

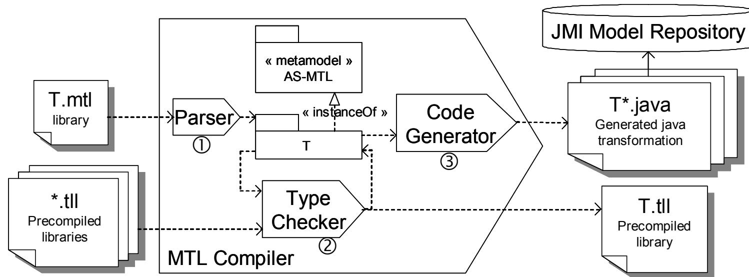


Figure A.1: The MTL Compilation Process

MTL is a compiled language, figure A.1 presenting the compilation process. In order to compile an MTL transformation `T` described in an `mtl` file, the first step is to parse it. A parser (①) reads the transformation as text and transforms it into an internal model that is compliant with the abstract syntax of MTL. In the next step, a type checker (②) refines this model by adding information about types. For instance, in order to deal with polymorphism, it is the type checker that will perform the analysis of MTL methods in order to reference, for each of them, other MTL methods that they are overriding. If necessary, the types used by the transformation `T` might need to be referred from already compiled MTL libraries. For example, the MTL standard library, which defines the MTL predefined types and operations, is typically used by all MTL transformations, and thus, it participates in such library-usage dependencies. In order for the MTL transformation `T` to be reused by other MTL transformations, its internal model, decorated with type information, is stored in a binary file (`T.tll`). In the end, a code generation step is performed (③). Java source files that implement the behavior described by the internal (refined) model of the MTL transformation `T` are generated, and they will make use of the model repositories on which the implemented transformation was defined to act. We used two `*` signs in figure A.1 in order to show that many precompiled libraries (`*.tll`) may be needed, on one hand, and several Java source files (`*.java`) may be generated, on the other hand, for one MTL transformation. If transformation `T` relies on other libraries, the generated Java source files for `T` will require the Java source files resulted from the compilation of those libraries.

A.2 Integrating Distribution Concern in an UML Model

With the rapid growth of the Internet and the associated web services revolution, distributed systems become more and more pervasive setting up new standards for modern industries. Current enterprise applications consist of heterogeneous components written in different programming languages and distributed in heterogeneous environments that comprise different hardware platforms, operating systems, data bases, and network protocols. The only way of masking these differences within an enterprise, or between enterprises, is by relying on middleware infrastructures, which can integrate diverse software components and allow them to interoperate effectively.

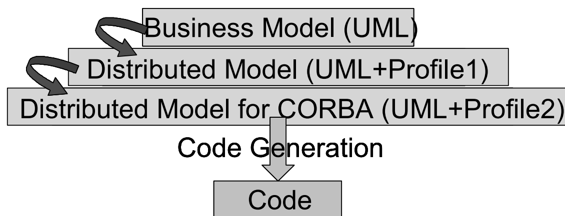


Figure A.2: Refinement Steps to Integrate a Distribution Concern with UML

This section shows how an improvement step may be accomplished in an LDE development process as depicted by figure A.2 when the modeling language of the more concrete layer is an extension of the more abstract layer. In a first step, we will consider the abstract system to be the description in pure UML class diagrams of the business model. The goal of the step is to integrate the distribution concern using a UML profile. The outcome of that first step will become the abstract system of a second refinement step that will be in charge of adapting the distributed system to the CORBA [ABB+04] middleware platform, using a second profile. To use an MDA terminology, we break here the strict separation between platform-independent models (PIM) and platform-specific models (PSM) by attributing different PIM or PSM roles to an abstraction layer. One may find a similar architecture in [ADvSP04]. As discussed in section A.2.3.1, the second profile inherits the first one using an higher-order hierarchy mechanism [Ern03]. If we believe that it is good LDE design when working with UML (or any profile-enabled language that is used in the two layers of a refinement step), this is not at all a mandatory requirement for applying LDE. Nor it is mandatory to use languages like UML or profile mechanism to practice good LDE.

This example is part of the enterprise fondue method [SS03, Sil06] and is further described in [SFS04a], though an overview is given in section A.2.2, especially regarding the distribution concern.

A.2.1 From Object-Oriented Designs to Distributed Systems

From a rather pragmatic point of view, we argue in this section that UML diagrams lack precision when it comes to providing specific distribution information that is typically required for generating distribution code out of UML models. We also provide a simple example for business model expressed in UML.

Let's consider the object-oriented design of a simple Bank system, like the one illustrated in figure A.3. We consider the example to be enough self-explaining for not entering into more details. It is important to mention nevertheless that not all designs follow the design by contract principles [JM97, BJPW99]. Typically, the interfaces in figure A.3 do not exist and clients act directly on the `Bank`, or on the `Account`, respectively. In that case, an intermediate "extract interface refactoring" [FBBO99, MB05] step is required in order to refactor the design and enforce good design principles.

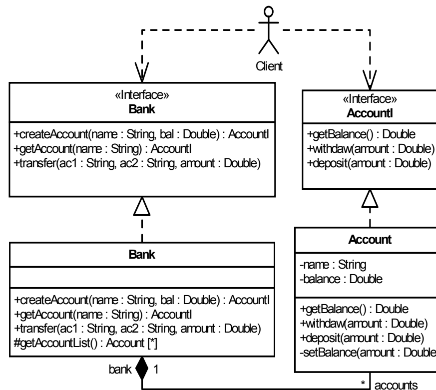


Figure A.3: The Bank Example

Once we have a "good" design, based on interfaces, we would like to automate the distribution of such object-oriented designs on different middleware infrastructures, and we would like to achieve this as transparently as possible for the developer.

One key aspect of distributed systems is their location transparency. Typically, `registries` are used to store the location of distributed objects. Clients `find` and use services (i.e., the interfaces of distributed objects that were already `bound` into registries) and do not care where they are located. Flexibility is very much increased, since distributed objects can be moved around and run on different machines, without any impact on the client side. It is only the information published in registries, and the registries themselves, that clients and distributed objects must agree upon.

Besides binding distributed objects into registries, there is also the problem of distributed interfaces. If we have a closer look at the `Bank::createAccount(...)` or `Bank::getAccount(...)` methods, we notice they both return `AccountI` interfaces back to the client. If we consider, for example, the CORBA technology for implementing the distributed system, code generators must generate a CORBA IDL for the `AccountI` interface as well, otherwise the `AccountI` interface cannot be passed around in a CORBA distributed setting. As a consequence, this interface must have been previously marked as distributed inside the design model, so that code generators know to treat it properly.

As a conclusion, in order to be able to generate distribution code for a specific middleware infrastructure, design models must be refined and enhanced with distribution related information. But the question is how to model such distribution-related information in UML? How to specify that an interface should be distributed? How to specify that an object instance of class `Bank` should be the entry point of the distributed system? How to differentiate that object instance from other objects in the model in order to be able to bind it into a naming registry? How to infer that, because the `BankI` interface should be distributed, the `AccountI` interface should be distributed as well?

Of course, we are aware that component-oriented models address some of these issues up to a certain level through component and deployment diagrams. However, from the implementation point of view, when it comes to generating distribution code, we still have to rely on object-oriented programming constructs and specific middleware support. Therefore, the work presented in this paper focuses only on the distribution of object-oriented designs, presenting new UML modeling elements for addressing distribution, and stressing the amount of distribution code that can be automatically generated from the enhanced object-oriented models. Moreover, we believe that it is a very good preliminary step before analyzing the actual support for generating a similar amount of distribution code out of component-oriented designs based on information that can be retrieved or inferred from component and deployment diagrams.

A.2.2 Enterprise Fondue and the Distribution Concern

The *Enterprise Fondue* software development method brings together four important paradigms in software engineering, namely Component-Based Software Engineering (CBSE), Separation of Concerns (SoC), Model Driven Architecture (MDA), and Aspect-Oriented Programming (AOP), and shows how they can complement each other at different stages in the development life-cycle of enterprise, middleware-mediated applications. The method identifies five layers corresponding to different levels of abstraction, each layer addressing specific concerns that pertain to enterprise applications in general. Model transformations are used to refine design models inside the same layer, or between different layers, along specific concern-dimensions.

For consistency reasons, we tend to use the terms middleware-specific *concern-dimensions* in relation with the *refining* activity ("refining along a dimension"), and middleware-specific *concerns* in all other contexts. Nevertheless, both terminologies refer to the same concepts, i.e., distribution, concurrency, transactions, security, and so on.

Figure A.4 presents how the refinement process in Enterprise Fondue evolves from one abstraction level to the next one by incremental refinements along different concern-dimensions. We use a mixed notation for representing the process flow, on one hand, and for representing UML 2.0 dependencies or relationships between modules, on the other hand.

The first refinement step is performed by the `MTL1` transformation. In the Enterprise Fondue terminology, we say that `MTL1` refines along a middleware-specific *concern*-dimension (`Cx`) according to an associated UML profile for that concern. This transformation is performed inside the Concern-Driven Object-Oriented Models Layer (`L2`) as defined in Enterprise Fondue. `MTL1` will refine here along the distribution concern-dimension. However, several `MTL1s` can be applied at this layer, addressing different middleware-specific concerns.

The second refinement step is performed by the `MTL2` transformation, that is, in the Enterprise Fondue terminology, along the *technology*-dimension (`Ty`). Actually, in the Enterprise Fondue, `MTL2` is a sequence of model transformations (not necessarily two), but for sake of clarity, we will avoid that aspect. Globally, the transformation correspond to refinement along middleware-specific concern-dimensions according to UML profiles for those concerns on the specific technology (`Cx` on `Ty`). All these transformations are performed inside the Technology-Dependent Layer (`L1`) as defined in Enterprise Fondue. For example, by refining along the CORBA technology-dimension, we will first apply the UML profile addressing the distribution concern on the CORBA technology (the `CORBADistributionRealizationProfile`, as it will be introduced in section A.2.3.3).

Finally, the code generation step is handled by the Parallax [SS05, Sil06] tool configured to handle the concern with the technology. Parallax also covers both the Platform-Dependent (`Pz`) and Language-Dependent (`Lw`) layers of Enterprise Fondue (`L0`).

A.2.3 UML Profiles to Address the Distribution Concern

The hierarchy of UML profiles presented in this section addresses the distribution concern in an MDA-oriented fashion at three different levels of abstraction: at a *platform-independent* level, at an *abstract realization* level, and at a *concrete realization* level for the CORBA technology (see figure A.5). We rely on specialization relationships between distribution profiles: each specialization introduces new modeling elements or simply refines the already existing ones.

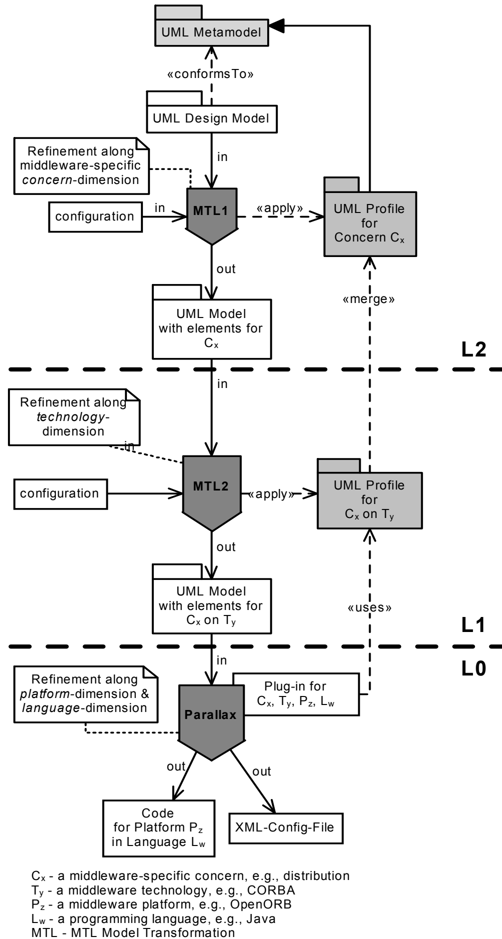


Figure A.4: Refinement Process in Enterprise Fondue

A.2.3.1 UML Distribution Profile

The purpose of distribution is to logically, or even physically separate communicating elements, typically a "core" system from its users. In UML, what these elements know about each other is described in terms of interfaces. UML interfaces are defined as sets of coher-

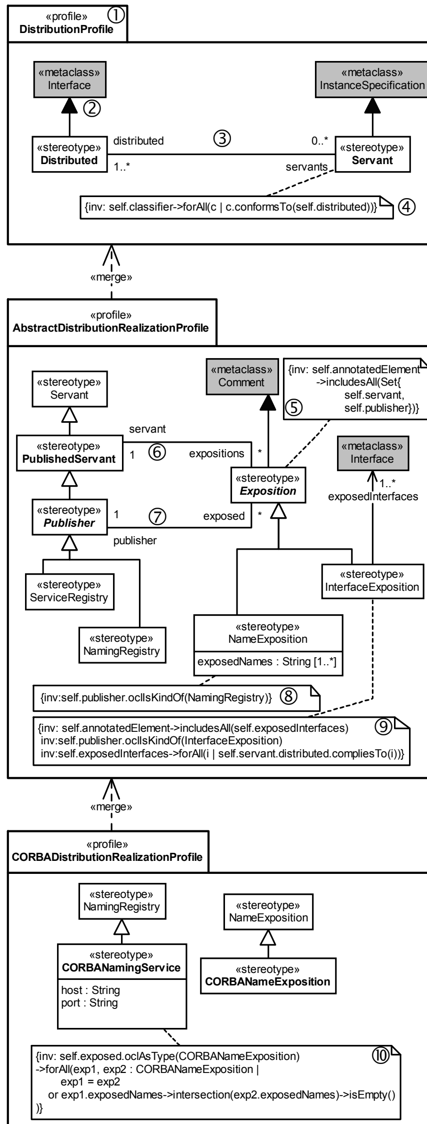


Figure A.5: MDA-Oriented Hierarchy of UML-D Profiles

ent publicly available features and obligations, fulfilled at runtime by instances of classes realizing them. The distribution process should end-up in subsystems communicating through known interfaces. We qualify these interfaces as distributed. This additional information can be added to the model of the system by applying the *`DistributionProfile`* presented in figure A.5 ①.

To make a difference between *what* interfaces are used for communication within a system and between systems, the profile defines the *«`Distributed`»* stereotype as an extension of the `Interface` metaclass as shown in figure A.5 ②. All features defined within a «`Distributed`» interface are remotely available. In the case a «`Distributed`» interface extends other interfaces, available remote features are only those defined within a «`Distributed`» interface, allowing in this way to separate between distributed and not distributed interfaces even within the same hierarchy of interfaces. For instance, the following OCL query may be defined within the scope of the «`Distributed`» stereotype to find all its remotely available operations:

```
context Distributed
def: allRemoteOperations : Set(Operation) =
  self.ownedOperation->union(
    self.allParents()
     ->select(oclIsKindOf(Distributed))
     .ownedOperation
  )
```

We have assumed here that the predefined OCL `oclIsKindOf` operation holds true if a model element has the stereotype passed as parameter. A similar query may also be defined to know about all remotely available attributes.

Nevertheless, it is also necessary to indicate key instances that the environment needs to know in order to start-up an interaction with the system as an entity. They are identified by applying the *«`Servant`»* stereotype. As these instances receive invocations from the environment, their class must realize a "well-known" «`Distributed`» interface, that is why «`Servant`» and «`Distributed`» stereotypes are associated as shown in figure A.5 ③. This allows one, once these stereotypes are applied on corresponding instances and interfaces, to indicate at the model level what are the «`Servant`» instances of a «`Distributed`» interface, and what are the «`Distributed`» interfaces of a «`Servant`» instance. As indicated by multiplicities of the association ③, a «`Servant`» instance must realize at least one «`Distributed`» interface, otherwise a client application does not know how to communicate with it. On the other side, a «`Distributed`» interface may be connected to any number of «`Servant`» objects. If none, it means the given interface does not participate in any interaction set-up. If a «`Distributed`» interface defines many «`Servant`» objects, this means the system has many entry points with this interface.

For the profiled model to be consistent, it is important to add a constraint stating that one of the classes of each «`Servant`» object must realize the «`Distributed`» interface it

is related to through the `distributed` association end ③. This constraint is given in figure A.5 ④.

Thought very comprehensive and understandable, one may remark that associations between stereotypes are not allowed in UML. Nevertheless, a stereotype may declare a tag definition whose type is a metaclass. We define thus the association from stereotype `S1`, which extends metaclass `C1`, to stereotype `S2`, which extends metaclass `C2`, by a tag definition in `S1` whose type is `C2` with the additional constraint that the corresponding tagged values are stereotyped with `S2`. Tag definition are crossed in case of bidirectional association.

A.2.3.2 UML Abstract Distribution Realization Profile

The purpose of the *`AbstractDistributionRealizationProfile`* is to provide a framework to describe *which* «`Servant`» instances are made available to the outside world, *how*, and *where*. We enter here the technology-dependent layer (`L1`) described in figure A.4. As this profile specializes entities described in the `DistributionProfile`, it merges this latter profile and as a consequence integrates into it all its stereotypes and tag definitions.

The *«`PublishedServant`»* stereotype is a specialization of «`Servant`», in order to show that «Servant» instances may be exposed to the outside world. Such «Servant» instances will be further referred as *«`PublishedServant`» instances* (`PSI`). Due to the specialization, the «`PublishedServant`» stereotype inherits the `InstanceSpecification` base class, the `distributed` tag definition, and the constraint ④, which must now hold for all elements conforming to this stereotype as well.

A *«`Publisher`»* is a «`PublishedServant`» instance specialized in making available a given «`PublishedServant`» to the outside world. This means that in order to interact with a `PSI`, an actor from the environment should first locate it by sending a request to a «`Publisher`» instance. How an external actor localizes a «`Publisher`» is voluntarily left unresolved and should be defined by specializing the «`Publisher`» stereotype according to the concrete technology to be used. This is the reason why the «`Publisher`» stereotype is abstract. For instance, if the Bank example of figure A.3 is made distributed, a `Client` should first retrieve the right «`Servant`» Bank instance by sending a request to a «`Publisher`» instance. Because the «`Publisher`» stereotype inherits the «`PublishedServant`» stereotype, an instance stereotyped «`Publisher`» is also a `PSI`, and may be published by another «`Publisher`» instance.

The relationship between a «`PublishedServant`» and a «`Publisher`» is expressed by the *«`Exposition`»* stereotype. Unfortunately, there is not really an ideal relationship between instances in the UML metamodel that this stereotype could extend. Therefore, we decided to make the `Comment` metaclass the base class of this stereotype. Information of *what* is published and *who* is the publisher is gathered through the associations ⑥ and ⑦ respectively. The constraint ⑤ requires all the «`Exposition`» com-

ments to be attached to both the «Servant» and the «Publisher» instances. A given PSI may be published several times within several «Publisher» instances, and conversely, a «Publisher» may expose several PSIs, all these relationships being modeled through «Exposition» comments.

«Publisher» and «Exposition» are abstract stereotypes. Therefore, they cannot be applied to a model element as such, because some *registration information* needs to be provided. We include therefore in the profile a kind of "reference implementation", although it would be possible to define an additional independent profile that extends the AbstractDistributionRealizationProfile and describes additional information required by another implementation mechanism.

As a first reference implementation mechanism, we propose to register a PSI by *names*, which are character strings, within a «Publisher» instance. We therefore define *«NamingRegistry»* as an extension of the «Publisher» stereotype, together with the *«NameExposition»* as an extension of the «Exposition» stereotype. The registration names are stored in the exposedNames tag definition, that is defined in the «NameExposition» stereotype. This tag definition has a 1..* multiplicity meaning that the PSI can be exposed by at least one name. A «NamingRegistry» may only publish PSIs through a «NameExposition», and a «NameExposition» may only refer to a «NamingRegistry». This is enforced by the constraint ⑧. This kind of exposition mechanism looks like the one of the phone directory of residents ("white pages").

As a second mechanism, we propose to register a PSI with the *services* it can offer. For instance, one can ask the environment for a printer, provided that there is a printer that knows about how to *print* PostScript documents. This mechanism is described by the couple of stereotypes *«ServiceRegistry»* for the publishing part and *«InterfaceExposition»* for the exposition part. This time the registration is performed by means of interfaces, referenced by the tag definition exposedInterfaces within the «InterfaceExposition» stereotype. Again, a constraint (⑨) states that they should be used together and only together. This kind of exposition mechanism looks like the one of the business phone directory ("yellow pages").

A.2.3.3 UML CORBA Distribution Realization Profile

The *CORBADistributionRealizationProfile* addresses the realization of the distribution concern when the implementation is supposed to use the CORBA technology. It takes advantage of the AbstractDistributionRealizationProfile by adapting its abstract concepts to the CORBA technology, so that a code generator has enough information to generate the necessary distribution code.

The *«CORBANameExposition»* stereotype represents a «NameExposition» using the CORBA technology. It extends the «NameExposition» but does not add particular information to it. The idea here is to state clearly that a PSI is registered by name using

the CORBA technology. The same holds for the *«CORBANamingService»* stereotype that extends the «NamingRegistry». In order for an actor of the environment to find the «CORBANamingService», we add the host and port tag definitions. As previously, an OCL constraint (⑩) enforces that these two stereotypes work together and only together. This constraint also enforces that exposition names are all different within the context of a «Publisher».

We chose to describe here only the CORBA technology, but the same principle may be applied to any kind of middleware technology, possibly with additional intermediate profile specialization steps.

A.2.4 MTL Model Transformations for Applying the UML-D Profiles

We present in this section the model transformations that incrementally refine existing design models according to the UML-D Profiles introduced in the previous section using the MTL language (see section 2.1.3.2). The Bank example (figure A.3) is used for illustrating the refinement process.

A.2.4.1 Refining Along the Distribution Concern-Dimension

The distribution transformation (MTL1-D) is refining centralized design models along the distribution concern-dimension according to the DistributionProfile presented in section A.2.3.1. Its name indicates, on one hand, that it belongs to the MTL1 family of transformations (as shown in figure A.4), and on the other hand, that it is related to the distribution concern.

In order to achieve the distribution, the transformation requires the developer to provide information about the interfaces that the «Servant» object should realize. These special interfaces will be further referred as *«Servant» interfaces* (SIs). If several «Servant» objects are needed, then the transformation may be called several times. As already mentioned in section A.2.1, we rely on the premise that all interactions with the environment occur through well-defined interfaces.

In the first step, the MTL1-D transformation "imports" the DistributionProfile into the model, making available the UML extensions it defines. The second step is to find the right classifier for the «Servant» object. Note that if more than one classifier realizes all the interfaces the developer has specified, then the MTL1-D transformation may choose an arbitrary one, or ask the developer to choose among the possible realizations; if no class is found, then the transformation ends in error, without modifying the model. Once the right classifier is found, the corresponding SIs are marked with the «Distributed» stereotype and an object instance of the found classifier is created and marked with the «Servant» stereotype. As these stereotypes are associated, it is still necessary to provide crossed tagged values as specified in section A.2.3.1. This means that the «Servant» object references its SIs by means of the distributed tagged value, and each SI references its

«Servant» objects by means of the servants tagged value, according to the DistributionProfile.

The last step of the MTL1-D transformation is to infer all interfaces that participate in interactions with the environment and to stereotype them «Distributed» as well. To this end, the transformation explores, starting from the provided SIs, the types of parameters of all operations, and the types of all attributes of each interface. While exploring, all encountered interface types are stereotyped «Distributed» (if not yet the case), and recursive explorations are started for each such interface. The MTL code for this exploration is shown in figure A.6.

```
//Within the MTL class Distributor
treatOperationDependencies() {
  i1 : Standard::Iterator;
  i2 : Standard::Iterator;
  pa : m::Core::Parameter;

  i1 := toDistribute.feature.getNewIterator();
  while i1.isOn() {
  if i1.item().oclIsKindOf(!m::Core::Operation!) {
    i2 := i1.item()
           .oclAsType(!m::Core::Operation!)
           .parameter.getNewIterator();
    while i2.isOn() {
       pa := i2.item();
       new Distributor()
        .init(self,pa.type).run();
       i2.next();
    }
  }
  i1.next();
}}
```

Figure A.6: The MTL1-D Transformation: Exploring Operations Part

As an example, we show in figure A.7 the outcome of the MTL1-D transformation on the Bank example when BankI is the only SI requested by the developer. As you can see, BankI gets stereotyped as «Distributed» (①). An object of class Bank, the only classifier realizing the BankI interface, is created and stereotyped «Servant» (②). The tagged values are shown in grey colored notes. Due to the association between «Distributed» and «Servant» stereotypes (as defined in the profile), the «Distributed» BankI interface references its SIs by means of the servants tagged value (③), and the «Servant» b references the «Distributed» BankI interface by means of the dis-

tributed tagged value (④). As the «Distributed» BankI interface contains operations, such as createAccount and getAccount, involving the AccountI interface, this last interface is stereotyped «Distributed» as well, but with an empty servants tagged value (⑤).

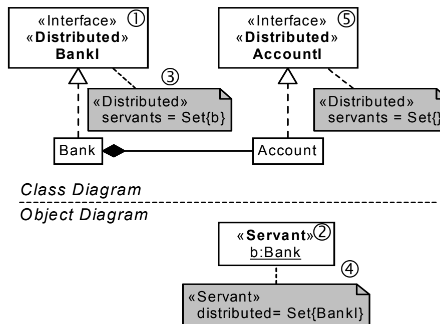


Figure A.7: The MTL1-D Outcome for the Bank Example

A.2.4.2 Refining Distribution Along the CORBA Technology-Dimension

The CORBA distribution realization transformation (MTL2-D) is refining distributed models along the CORBA technology-dimension. More precisely, it refines models, to which the DistributionProfile was already applied, to ones that are more specific about how the distribution concern is actually implemented on the CORBA technology. As previously, the name indicates that it belongs to the MTL2 family of transformations (as shown in figure A.4), on one hand, and that it is related to the distribution concern, on the other hand.

In order to be able to apply the CORBADistributionRealizationProfile, the MTL2-D transformation requires the developer to provide all the new information that this profile adds with respect to the DistributionProfile. Indeed, MTL2-D needs to be able to provide each CORBA publisher (i.e., CORBA Naming Service [BDII02], «CORBA-NamingService») with its name, host and port, and each PSI with its expositions («CORBANameExposition») containing the names to be exposed (exposedNames) and the publisher to be used (publisher). The transformation does not integrate any «InterfaceExposition», which is more related to the Trading Service [ABH+00] of CORBA that is not discussed in this paper.

Like for MTL1-D, the first step of the MTL2-D transformation is to “import” the CORBADistributionRealizationProfile into the model. The MTL2-D also “imports” standard CORBA libraries, like the interface of the CORBA publisher, which is

org::omg::CosNaming::NamingServiceExt. Then, the transformation creates «CORBANamingService» publisher objects and sets their host and port tagged values according to the provided parameters. Since «CORBANamingService» inherits from the «Servant» stereotype the distributed tag definition, and because of its multiplicity 1..*, it is necessary to provide it with a value. Other inherited tagged values are not mandatory as their lower bound multiplicity is zero and because they would have no meaning for the «CORBANamingService» publisher. Moreover, as the «CORBANamingService» publisher instance provides behaviors described in the NamingServiceExt interface, this interface must be stereotyped as «Distributed». The distributed and servants tagged values are used to relate the instance and the interface together.

At the end, the MTL2-D transformation creates the expositions as specified by the developer. As defined in the profile, each «CORBANameExposition» is a Comment on the PSI and the «Publisher» instance, which are referenced through the servant and publisher tagged values respectively. Both the PSI and the «Publisher» («CORBANamingService» in our case) know about the «Exposition» comment by means of the expositions and exposed tagged values respectively. The exposed names of a «CORBANameExposition» are stored in the exposedNames tagged value. figure A.8 shows the MTL code for this last step.

The outcome of the MTL2-D transformation for the Bank example is shown in figure A.9. The source model for the transformation is the one shown in figure A.7. The MTL2-D provides one «CORBANamingService» publisher object cns (①) and sets its distributed tagged value to reference the NamingServiceExt interface. The «Distributed» stereotype was applied to this last interface and its servants tagged value was set to reference the cns, but this is not depicted here. The diagram also shows the b object, which was applied the «PublishedServant» stereotype (②). A new comment, named BankExposition, with stereotype «CORBANameExposition», represents the expositions (③), and it is referenced by the expositions and exposed tagged values in the b and cns instances respectively. This comment annotates these instances and references them with the servant and publisher tagged values respectively. For the sake of

```
//The exposition
//- publisher is the publisher object
//- servant is the published servant object
//- profile is an MTL proxy for the
//          CORBADistributionRealizationProfile
//- expositionName is the provided name
//                 of the exposition
//- publishedNames are the names the servant
//                 is registered with
ex := new m::Core::Comment();
ex.name := expositionName;
associate (
  comment := ex : m::Core::Comment,
  annotatedElement := servant
          : m::Core::ModelElement);
associate (
  comment := ex : m::Core::Comment,
  annotatedElement := publisher
          : m::Core::ModelElement);

profile.applyStereotype(ex,
  profile.cORBANameExposition);

profile.setTaggedValueData(servant,
  profile.publishedServantExpositionsTag,ex);

profile.setTaggedValueData(publisher,
  profile.publisherExposedTag, ex);

profile.setTaggedValueData(ex,
  profile.expositionServantTag, servant);

profile.setTaggedValueData(ex,
  profile.expositionPublisherTag, publisher);

profile.setTaggedValueData(ex,
  profile.nameExpositionExposedNamesTag,
  publishedNames);
```

Figure A.8: The MTL2-D Transformation: Creating Exposition Part

clarity, all tagged values have been provided, even if empty, and classified according to the stereotype in which they where declared.

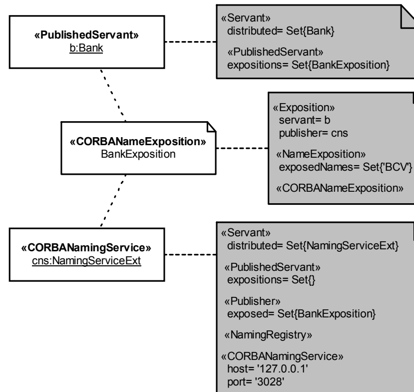


Figure A.9: The `MTL2-D` Outcome for the Bank Example

A.2.5 Conclusions

The Enterprise Fondue method proposes a systematic approach for addressing pervasive services in an MDA-compliant manner, at different levels of abstraction, through incremental refinement steps along middleware-specific concern-dimensions. In this section, we introduced the key elements that support the Enterprise Fondue method when refining along the *distribution* concern-dimension, namely: (1) the UML Profiles for Distribution (*UML-D Profiles*) that address the distribution concern in an MDA-oriented fashion at three different levels of abstraction (*platform-independent* level, *abstract realization* level, and *concrete realization* level), (2) the model transformations that incrementally refine existing design models (within the same or between different MDA-levels) along distribution-related concern-dimensions and in conformance to the proposed UML profiles, and (3) the Parallax support for generating code towards specific middleware infrastructures. The CORBA technology was used to illustrate how the refinement process evolves on a concrete example.

This example follows the prescriptions of LDE. It applies a stepwise refinement process taking advantage of model transformation. This section actually introduced Enterprise Fondue as an LDE methodology by stating, as prescribed in section 2.1.1 on page 11, (1) what are the levels of abstraction and their platforms (Business Model, Distributed Model, Distributed Model for CORBA), (2) what are the languages to use (UML, UML + DistribuitonProfile, UML + CORBADistributionRealizationProfile), (3) the refinement process

(MTL1 and MTL2 transformations), (4) the code generation process (Parallax), even though (5) verification and testing were left aside.

The example presented in this section also shows how an LDE methodology can be tailored to support new concepts introduced in intermediate steps. Indeed, the genuine LDE methodology states that one should develop the system under study using the UML language. Then, code is generated out of the UML model. We introduced here new concepts relative to distribution in a first step, and distribution using the CORBA platform in a second step. For the LDE methodology to be properly tailored, we enhanced the modeling language (i.e. UML) using a tag mechanism, and we introduced semi-automatic improvements. A problem is that code generation needed to be tailored as well. In our example, we used the aspect-oriented customization mechanism offered by the Parallax code generator [SS05].

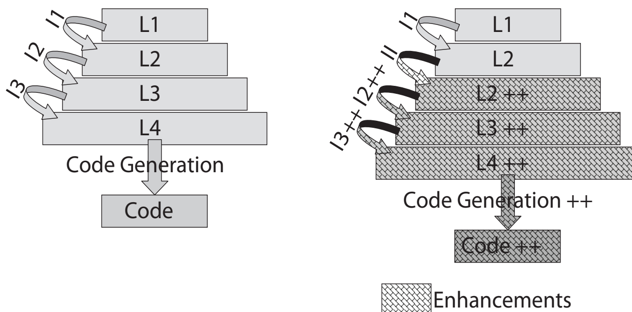


Figure A.10: Customizing an LDE Process

Figure A.10 shows a generalization of LDE methodology customization. At the left, the genuine LDE process states that the system under study must be modeled first using the L1 language. The model is then improved by I1, I2 and I3, producing specifications using the L2, L3, and L4 languages, respectively. A final code generation step is performed to obtained the final implementation. The right part of the figure shows the same LDE process, slightly enhanced, to integrate a new concern, following the example of the distribution concern that introduces new concepts to the UML language, as presented in this section. The new concern is introduced by an II improvement (following the example of the `MTL1-D` and `MTL2-D` transformations), that integrates the concern using an enhanced version of the L2 language (L2++). For those enhancements to be lost neither by further improvements (I2, I3, and code generation), nor by languages of the next steps (L3, L4, and target code), improvements and languages needs to be customized either (I2++, L3++, etc.), unless the new concern is handled by a library (see section 1.1 on page 1). If languages can be custom-

ized using higher-order hierarchies or tag mechanisms (as shown in this section), and code generation can be customized using the approach proposed by Parallax [Sil06], one needs means to customize improvements. We propose in the next section a mean to customize improvements, when they are defined as MTL model transformation.

A.3 AspectMTL: Customizing Improvements

Model transformations are undoubtedly one of the key technology in the realization of the LDE vision, and thus one need transformation languages to apply an LDE process [SK03]. Among other usages, model transformations are the ones responsible for refining models for example to map them to concrete middleware-based implementations, providing thus an elegant approach to adapt system specifications to the peculiarities of the new middleware infrastructures that do not cease to appear as an example in previous section.

LDE also promotes reuse and adaptation of existing languages to particular needs. Model transformation languages thus may be adapted to the needs of a particular LDE methodology (i.e. an LDE instance). We propose in this part to improve reusability of the transformations in use in the LDE process presented in previous section, thus reusability of the complete methodology. This will require to change the Model Transformation Language (MTL - see section A.1.2) that is employed to implement the transformations.

Model transformations should be able to act on *any kind of model of any kind of meta-model*. Since model transformations are at the same time models compliant with the meta-model of the transformation language, model transformations should be able to *transform* other model transformations independently of their metamodels. As a consequence, all existing model transformation languages (to our knowledge) implement such a “reflective” behavior. However, the "reflective" use of model transformations is not trivial.

Typically, writing model transformations for driving the development process of domain-specific applications requires the transformation developer to be familiar with the metamodel of that specific domain and with the syntax of the model transformation language used – and no more than that. As a consequence, many transformation developers are not at all familiar with the metamodel of the transformation language itself, and thus they are not capable of writing "reflective" model transformations, i.e., model transformations that transform already existing model transformations.

We present here a solution inspired by Aspect-Oriented Programming (AOP) [KLM+97] approaches. We have designed and implemented an MTL *weaver* that modifies MTL transformations according to some *weaving behavior* that is specified as a special kind of MTL transformations, called *MTL-aspects*. The MTL transformation produced by the MTL weaver can be immediately used for refining application models.

As in the case of AspectJ [KHH+01], which is an aspect-oriented extension to Java, the syntax defining the weaving behavior in MTL-aspects is a small AOP-like extension to

the MTL language itself. In this way, relying on a few high-level AOP-like but MTL-based constructs for defining the weaving behavior, average MTL transformation developers should not have any problems using this MTL extension straightforwardly for defining their "reflective" model transformations.

The entire compilation process of the MTL language as presented in figure A.1 of section A.1.2 relies on the model of the MTL transformation `T` itself, which complies with the well-defined MTL metamodel. Therefore, steps ①, ②, and ③ of figure A.1 can be viewed as special transformations acting on the MTL model of the transformation `T` itself. Besides these three steps, it is at this MTL model level of the MTL transformations that new special transformations may be defined in order to change the very behavior of those MTL transformations. Following this idea, our MTL weaver is indeed implemented as such a special transformation, acting on the MTL models of the MTL transformations and transforming them according to the weaving behavior defined in MTL-aspects, as we will see in section A.3.2.

A.3.1 Motivations

We present in this section how currently applied MTL transformations benefit from the weaving support provided by the MTL weaver, promoting the separation of concerns paradigm even at level of model transformations.

Separation of concerns [Par72] and modularity are fundamental techniques of software engineering. Decomposing software into smaller, more manageable and comprehensible parts, each of which encapsulating and addressing a particular area of interest, called a *concern*, is a well-proven method towards developing applications that are easy to configure, adapt, or extend according to changes in the requirements specification.

Middleware is an essential element in large distributed systems such as those that support enterprise applications, requiring multiple heterogeneous components to interoperate. Previous section has shown that middleware, like software in general, is subject to concerns. Several concern-dimensions about middleware can be grouped into a category called Middleware Services, as the middleware addresses specific concerns of a system, such as distribution, concurrency, security, or transactions. An extended list of categories that group several middleware-specific concern-dimensions can be found in [Sil06]. In the context of Enterprise Fondue we defined several MDA-oriented UML profiles that address middleware-specific concerns at different levels of abstraction. MTL transformations were used to incrementally refine existing design models (within the same or between different MDA-levels) along middleware-specific concern-dimensions and according to the UML profiles defined. A complete example of applying the Enterprise Fondue method for addressing the distribution concern in the concrete case of the CORBA technology was also presented. The proposed *UML-D Profiles* address the distribution concern at three different MDA-levels of abstraction: *platform-independent* level (the `DistributionProfile`), *abstract realiza-*

tion level (the AbstractDistributionRealizationProfile), and *concrete realization* level (the CORBADistributionRealizationProfile).

Based on the support provided by the MTL weaver, we refactored the MTL transformation that refined application designs in the context of the Enterprise Fondue method along the distribution concern-dimension and according to the DistributionProfile. Out of one big model transformation that performed the entire refinement, we have now one standard MTL transformation that performs the copy of an input model to an output model, both models being compliant with the same UML metamodel, and a very small MTL-aspect that defines the weaving behavior according to the DistributionProfile that has to be applied. Both the MTL-Copy transformation and the MTL1-D-Aspect are now fully separated as they should be, since they address totally different concerns. Figure A.11 a sketches the refinement process in the presence of the MTL1-D-Aspect, or more general in the presence of MTL-aspects. Its name, MTL1-D-Aspect, was chosen in accordance with the MTL1-D transformation. The MTL-Distribution-Copy transformation is the result produced by the weaver when modifying the MTL-Copy transformation according to the weaving directives defined in the MTL1-D-Aspect.

A more complex example is shown in figure A.11 b, where the metamodel of the input and output models changes. In this example we move from a UML model to a Java model ready to be mapped to concrete Java implementation. Considering as input the output model of the previous refinement process, we refine this time along the RMI-technology and Java-language concern-dimensions as defined in the context of the Enterprise Fondue method. While the MTL-UML2Java deals with transforming any UML model to its correspondent Java model (relying on their respective metamodels), the MTL2-D-Aspect addresses how distribution specific elements in the UML model are transformed into their Java model counterparts when employing RMI as their implementation technology. For instance, interfaces marked as «Distributed» in the UML model will extend java.rmi.Remote in the Java model; similarly, the class of the object marked as «Servant» will extend java.rmi.UnicastRemoteObject in the Java model, and so on. Once again, the name, MTL2-D-Aspect, was chosen in accordance with the MTL2-D transformation even though we considered this time another technology, i.e., we have chosen RMI instead of CORBA. The MTL-RMI-UML2Java transformation is the result produced by the weaver when modifying the MTL-UML2Java transformation according to the weaving directives defined in the MTL2-D-Aspect.

As can be seen in figure A.11, the support provided by the MTL weaver has enabled us to make modular the different concerns in stand-alone units of encapsulation represented by MTL-aspects. In this way, we give transformation developers not only the possibility, but also the means to rely on the well-proven power of separation of concerns even at model transformation level. Moreover, the size of such MTL-aspects is very much reduced, compared to their corresponding implementation in the initial MTL transformations, since they rely on the MTL weaver which is now the one carrying all the burden of the weaving. The

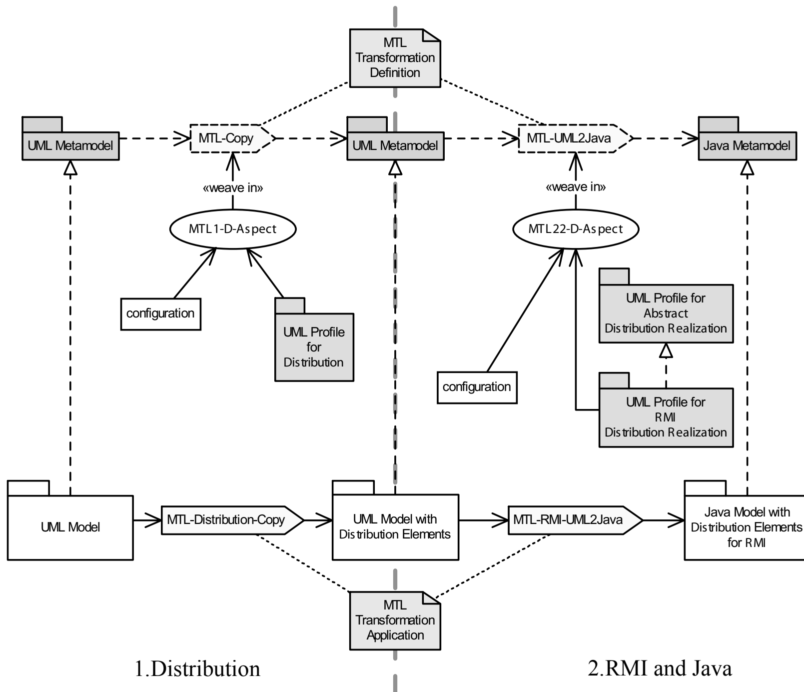


Figure A.11: Refining along the Distribution, RMI-Technology, and Java-Language Concern-Dimensions

example presented in figure A.11 a is reconsidered further on in section A.3.2 where we discuss in more details its complete implementation.

Besides encapsulating middleware-specific concerns into MTL-aspects as presented in this section, the number of possible usages of such MTL-aspects is unlimited since the support provided by the MTL language enables us to implement almost anything in the MTL weaver, and thus, the expressiveness power that could be provided to transformation developers through the MTL extension syntax may be very broad, covering all possible and impossible needs that developers may think of.

A.3.2 The MTL Weaver

Reusability has always been an important concern in the software development industry due to its potential to reduce the cost of software development. During the last decade, different

levels of reuse have been proliferated, such as functions, procedures, classes, components, aspects, or even entire models. But how can we achieve the reuse of model transformations? How to adapt existing model transformations that already accomplish most of our needs?

The reuse of MTL transformations is currently promoted at the level of MTL libraries, which are some kind of light model transformation components. In this section, we present some implementation details and the provided facilities of an aspect-oriented support that allows transformation developers to reuse existing MTL transformations and to easily adapt them in order to address new needs, or concerns, that the application under development has to incorporate. The main concepts of the MTL weaver are introduced along with the AOP-like extension to MTL for defining the weaving behavior in MTL-aspects. We also present an example showing both the input and the output of a concrete weaving.

The standard MTL language already provides support for transformation developers to define MTL transformations that *transform* other MTL transformations. However, writing such "reflective" MTL transformations still requires transformation developers to be familiar with the metamodel of the MTL language itself, a requirement that significantly reduces the number of such developers. In order to overcome this impediment for the MTL language, we propose a solution inspired by AOP approaches. We have designed and implemented an MTL *weaver* that modifies MTL transformations according to some *weaving behavior* that is specified in terms of *weaving directives* modularized in special stand-alone MTL transformation encapsulation units, called *MTL-aspects*. As in the case of AspectJ, which is an aspect-oriented extension to Java, the syntax defining the weaving behavior in MTL-aspects is a small AOP-like extension to the MTL language itself. In this way, relying on a few high-level AOP-like but MTL-based constructs for defining the weaving behavior, average MTL transformation developers should not have any problems using this MTL extension straightforwardly for defining their "reflective" model transformations.

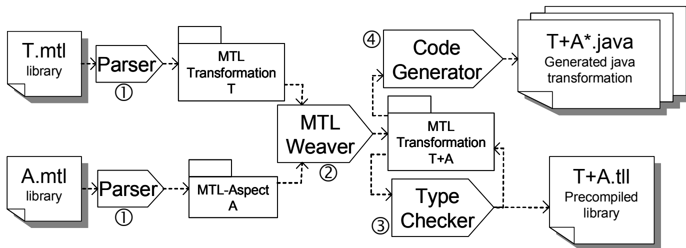


Figure A.12: The MTL Weaving Process

The place of the MTL weaver in the MTL compilation process and the evolution of the MTL weaving process are presented in figure A.12, where the MTL transformation `T` is

refined according to the weaving directives defined in the MTL-aspect `A`. The weaving process is very similar to the compilation process presented in figure A.1. First, both `T` and `A` are parsed (①) in order to transform the two text files into internal MTL models compliant with the MTL metamodel. The important change comes next, when the *MTL Weaver* (②) reads the two internal models of `T` and `A`, and produces a new model instance (of the MTL metamodel) for the new MTL transformation `T+A`, which represents the result of modifying `T` according to the weaving directives defined in `A`. Even though it is not explicitly shown in figure A.12, the MTL weaver itself is implemented as an MTL transformation as well. Once this weaving step is finished, the normal compilation process can continue with the type checking step (③), which produces a reusable precompiled MTL library, and the code generation step (④), which produces Java source files. Please notice that the weaving process results in a completely new MTL transformation, without making any changes to the original MTL transformation `T`. In this way, both transformations can independently be reused later on in order to transform application models. Moreover, the MTL-aspect `A` may be reused as well for refining other MTL transformations according to the same weaving directives. Another remark is that we reused as is elements of the MTL "official" compiler of section A.1.2 so that only the MTL weaver had to be implemented.

A.3.3 MTL-Based Syntax for Describing the Weaving Behavior

There are two major requirements that an MTL-aspect must fulfill. First, it must clearly identify *where* the modifications have to be performed, and second, it must clearly define *what* are those modifications. In AOP terminology, a *join point* is a well-defined point in the execution of a program where additional functionality may be "injected". To identify such points in our weaving process, a *pattern matching* mechanism is used with respect to the names of the MTL libraries, MTL classes, MTL methods, etc. Both requirements can be expressed using the MTL syntax as well, relying on small extensions that are detailed in this section.

One of the extension mechanisms proposed by the MTL language is the tagging facility. *Tags* are key/value pairs associated either with an MTL library, an MTL class, or an MTL method. Since tags are part of the MTL metamodel, once they are analyzed by the MTL parser, they populate the internal MTL model representing the MTL transformation. This makes it possible for the MTL weaver presented in figure A.12 ② to access these tags and to use them for very different purposes. Since MTL-aspects only rely on the *tag* extension mechanism to define additional weaving directives, it is possible to use the same parser for reading both MTL-aspects and MTL transformations, as shown in figure A.12 ①.

In order to give an example of an MTL-aspect that could play the role of `A` in figure A.12, we show in figure A.13 some snippets of the `MTL1-D-Aspect`. For the sake of readability, we will further on refer to as *input library* the MTL library taken as input for the weaving process, i.e., the library that plays the role of `T` in figure A.12, and its elements

input classes, *input methods*, etc. The MTL library produced as a result of the weaving process, `T+A` in figure A.12, will further on be referred to as *output library*, and its elements *output classes*, *output methods*, etc.

```
  library Copy;
  tag rename := specialtag [Distribution];

① class Copier {
    servantIterfaceName : Standard::String;

    initDI(sin : Standard::String) : Copier {
      self.servantIterfaceName := sin;
      return self;
    }
  }

② class [{Copier$}] {
    [{^getTarget(.*)}](theSource : Standard::ModelElement)
    tag merge := specialtag [Append];
    tag refactorParameters := booleantag true; {
      theSource.toOut();
    }
  }
```

Figure A.13: Snippets of the `MTL1-D-Aspect`

Each line in figure A.13 may be considered as a weaving directive for the MTL weaver. For instance, the first line defines the name of the input library that the `MTL1-D-Aspect` will have to be weaved in, i.e., `Copy`. In order not to alter the `Copy` input library during the weaving process and to avoid name clashes between input and output libraries, the name of the output library has to be provided. This can be achieved by defining a tag on the MTL library of the MTL-aspect. We have named this tag `rename`, and its value represents the name of the MTL library produced as a result of the weaving process, e.g., `Distribution` in this particular case.

By default, elements of the input library will be simply reproduced in the output library. However, this simple reproduction can be tuned by the rest of the MTL-aspect. For instance, in figure A.13 ①, the MTL class `Copier` is defined. This weaving directive indicates to the MTL weaver that if a class with the same name exists in the input library, then the reproduced class in the output library contains both the members in the input class and the ones defined in the MTL-aspect class. This process is called *class merge*. On the other hand, if this class does not exist in the input library, then it will simply be added to the output library exactly as it is defined in the MTL-aspect, i.e., it will include all member defini-

tions defined by the MTL-aspect, e.g., the `servantIterfaceName` MTL attribute and the `initDI` MTL method.

A *conflict* may appear during a class merge if some members in the matching input classes and in the MTL-aspect class have the same name. If the member in the MTL-aspect is an attribute, it will be added as it is, without worrying whether the name of the attribute already exists in the input MTL library, since the rest of the compilation process will detect such a duplicate attribute, if any, and an error will be thrown. For methods, the detected conflict is registered to be solved later.

MTL-aspect developers may refer many MTL classes or MTL methods in a single pattern by relying on "wildcard" facilities, such as "_", which matches any name, or the more sophisticated regular expressions delimited by curly brackets. For instance, in figure A.13 ②, the class named `{Copier$}`, matches all input classes whose name ends (denoted by `$`) with "Copier", and its method `{^getTarget(.*)}` matches all input methods, defined on the matched input classes, whose name starts (denoted by ^) with "getTarget". As a rule, MTL-aspect developers should not abuse of such constructs in order to add new classes or methods to the output library.

The class merge process, as it is implemented in the MTL weaver, is shown in figure A.14. The `libClass` represents the input class, and the `behaviorClass` represents the MTL-aspect class. Please note that the name of the `behaviorClass` matches the name of the `libClass` as a precondition for the `mergeClass` method.

A method conflict may be solved according to some predefined rules. We have identified three kinds of possible rules that prescribe the MTL weaver how to manage the instructions defined by the conflicting method of the MTL-aspect:

- run MTL-aspect instructions at the very beginning of the output method,
- run MTL-aspect instructions just before returning from the output method, or
- replace input instructions with MTL-aspect instructions in the output method.

It is the responsibility of the MTL-aspect developer to indicate which alternative s/he desires to be chosen for a given method conflict. For this purpose, we defined the `merge` tag that has to be added on each conflicting method in the MTL-aspect. The three possible values corresponding to the previously described rules are `Prepend`, `Append`, and `Replace` respectively. If a conflict cannot be solved, the weaving process ends in failure.

The instructions in the MTL-aspect method may need to refer to some parameters of the matched input methods. The presence of the boolean tag `refactorParameters` set to `true` makes the parameters of the input methods accessible inside the MTL-aspect according to the names provided in the MTL-aspect method. Moreover, this tag makes the method matching take care of the number of parameters in the input methods rather than just matching the names of the methods.

As an example, figure A.13 ② states that for all input methods whose names start with "getTarget" inside classes whose names end with "Copier", the first parameter, named

```
mergeClass(libClass : BasicMtlASTView::UserClass;
           behaviorClass : BasicMtlASTView::UserClass) {
  lo : Standard::Set;
  // adding attributes
  if (isNull(behaviorClass.definedAttributes).not()) {
    foreach (at : BasicMtlASTView::Attribute)
      in (behaviorClass.definedAttributes) {
      libClass.appendDefinedAttributes(at);
    }
  }
  // merging operations
  foreach (bo : BasicMtlASTView::Operation)
    in (behaviorClass.definedMethods) {
    lo := matchingOperations(libClass, bo);
    if (lo.size().[=](0)) { // to be added
      if (self.canAdd(bo)) {
        libClass.appendDefinedMethods(bo);
      } else {
        bo.name.concat(
          'seems to be a pattern; no correspondance found.'
          ).toOut();
       'ignoring addition to class '.concat(lib-
Class.name).toOut();
      }
    } else { // conflict, to be treated later
      self.operationConflicts :=
         operationConflicts.including(
           new OperationConflict().init(libClass, lo, bo));
    }
  }
}
```

Figure A.14: MTL Weaver Snippets for Class Merge (`mergeClass`)

in the MTL-aspect `theSource`, must be sent to the console by means of the MTL predefined operation `toOut`. This output must be performed before returning from the modified MTL methods, as stated by the value `Append` of the `merge` tag defined for the MTL-aspect method.

As a summary, the list of possible tags that may appear in the definition of an MTL-aspect is provided in figure A.15 The first column gives the name of the tag as it must appear in the MTL-aspect. The second column indicates on which MTL element this tag may be defined. The third column indicates whether the presence of the tag is mandatory or

optional; default values are indicated for optional tags. The fourth column gives a brief description of the semantics of the possible associated values.

Tag Name	Base MTL Element	Presence	Description
`rename`	Library	mandatory	The name of the output library.
`merge`	Method	mandatory if conflict	`Prepend` to add instructions at the very beginning of the method. `Append` to add instructions just before returning from the method. `Replace` to replace initial instructions with MTL-aspect instructions.
`refactor-Parameters`	Method	optional; default value is `false`	Indicates if the number of parameters has to be considered in the pattern matching, and if parameters have to be intercepted for further use inside MTL-aspect instructions.

Figure A.15: Predefined MTL-Aspect Tags

As we showed on some concrete examples, the MTL-aspect developer does not need to have a deep knowledge of the MTL metamodel and its semantics in order to transform an MTL transformation. All s/he needs to know is the MTL syntax and some predefined tags. Moreover, with the current implementation of the MTL weaver, an MTL-aspect is about 10 times smaller (in lines of code) and about 50 times faster to develop than a standard MTL transformation that would achieve the same weaving behavior on another MTL transformation.

Please notice, however, that the MTL weaver and the aspect-oriented support provided are prototypes, so there is still room for refinement and improvement. New constructs could be added in order to address MTL-aspect developer needs and to facilitate as much as possible the development of "reflective" MTL transformations. For instance, it would be very helpful to have a pattern matching for instructions or expressions, e.g., matching all *calls* to a given method. The pattern we adopted for extending the MTL language with AOP-like constructs would remain nevertheless the same, i.e., extending the language by providing new tags that change the semantics of their base element, just like UML profiles extend the UML.

A.3.4 Running Example

In this part, we consider the weaving of the `MTL1-D-Aspect` in the simple MTL `Copy` transformation in order to modify its behavior and make a system distributed by applying

the stereotypes defined in the `DistributionProfile` according to some configuration information. Since the goal is to illustrate the most important principles of the weaving process, we focus on very small parts of the example.

The input MTL `Copy` transformation is specialized in copying an input UML 1.4 model to an output UML 1.4 model. Snippets of the transformation are presented in figure A.16. The transformation is located in the MTL library `Copy`, having two variables, `in` and `out`, for referring to the input, and output models respectively. One of the MTL classes of this library is `Copier`, which defines the `getTarget` method. This method takes as parameter a UML element `srcElt` from the `in` model, and retrieves and returns the corresponding UML element inside the `out` model. Another MTL class, extending `Copier`, is `UML14CreatorCopier`, which defines the `getTargetClass` method. This method takes a UML class `src` in the `in` model as parameter, and is responsible for creating and returning a UML class in the `out` model.

```
library Copy;
model in  : RepositoryModel;// should be a UML1.4 MetaModel
model out : RepositoryModel;// should be a UML1.4 MetaModel
class Copier {
  getTarget(srcElt : in::Core::Element) : out::Core::Element
    {
    r : out::Core::Element;
    ... // compute r
    return r;
  }
}
class UML14CreatorCopier extends Copier {
  getTargetClass(src : in::Core::Class) : out::Core::Class {
    r : out::Core::Class;
    r := new out::Core::Class();
    trace(src, r);
    return r;
  }
}
```

Figure A.16: Snippets of the `Copy` Input Library

We present now two of the modifications that have to be performed in order for the MTL `Copy` transformation to make a system distributed. The first one is to make an interface remotely available, but before doing this we still need to identify the right interface. The solution we considered is to add an attribute, `servantIterfaceName`, to the MTL `Copier` class as a placeholder for the name of the interface to be distributed. This attribute is transmitted to the MTL `Copier` class by means of the new method `initDI` defined in the `MTL1-D-Aspect`. The second modification is to display on the console UML elements

from the `in` model for which a correspondence in the `out` model has been requested. A thorough analysis of the complete MTL `Copy` transformation would clarify that such correspondences are only requested when invoking methods whose names start with "getTarget", and which belong to a class whose name ends with "Copier". These modifications are prescribed in the `MTL1-D-Aspect` that was partly presented in figure A.13, where part ① corresponded to the first modification, and part ② to the second one.

The result of weaving the `MTL1-D-Aspect` in the MTL `Copy` transformation is shown in figure A.17. Even though we have clearly stated in section A.3.2 that the results of the MTL weaving process are just MTL binaries and Java source files, figure A.17 represents what a pretty printer would produce for the MTL binary. Changes introduced by the MTL-aspect are highlighted by change bars. Since the output MTL library is different from the original MTL `Copy` library, a renaming has occurred according to the `rename` tag that was specified on the library definition inside the `MTL1-D-Aspect`, as shown in figure A.17.

Part ① of the `MTL1-D-Aspect` in figure A.13 states that an MTL class named `Copier` must appear with a `servantIterfaceName` attribute and an `initDI` operation in the output library. Even though such an MTL `Copier` class already exists in the input library, no name conflicts have been found, and therefore member definitions from both the MTL-aspect and the input class are directly added to the MTL `Copier` output class, as shown by figure A.17 ①.

The MTL-aspect method defined in part ② of the `MTL1-D-Aspect` in figure A.13 matches the input methods `Copier::getTarget` and `UML14CreatorCopier::getTargetClass`. Please note that the presence of the `refactorParameters` tag set to `true` in the MTL-aspect has made the method matching check that only one parameter is defined for these input methods, parameter that will further on be used as the variable `theSource` inside the body of the MTL-aspect method. The tag `merge` set to `Append` defined on the MTL-aspect method indicates how possible conflicts should be solved. Since conflicts have indeed been found, the instructions defined in the MTL-aspect have to be inserted in the output class just before returning from the corresponding reproductions of the input methods in the output class, as part of the output library. To achieve this, we rely on the MTL `try-catch-finally` statement: instructions of the input method are reproduced in the `try` part, and instructions from the MTL-aspect method are reproduced in the `finally` part, as shown in figure A.17 ②. In this way, we enforce that instructions from the MTL-aspect method are executed just before returning from the output method, wherever an MTL `return` instruction may appear in the input method. The `true` value for the `refactorParameters` tag also instructs the MTL weaver to produce new variables in the output methods according to the parameters defined in the MTL-aspect method that are supposed to match parameters from the input methods. These new variables represent placeholders for the values of the parameters of the input methods that were intercepted by the corresponding MTL-aspect method. Applying this rule for the two input methods matching

```
    library Distribution;
    model in  : RepositoryModel;// should be a UML1.4 MetaModel
    model out : RepositoryModel;// should be a UML1.4 MetaModel
    class Copier {
      servantIterfaceName : Standard::String;
      initDI(sin : Standard::String) : Copier {
①       self.servantIterfaceName := sin;
        return self;
      }
      getTarget(srcElt : in::Core::Element) : out::Core::Element
    {
        r : out::Core::Element;
        theSource : Standard::ModelElement;
②       theSource := srcElt;  // [*]
        try {
          ... // compute r
          return r;
        } finally {
②         theSource.toOut();  // [*]
        }
      }
    }

    class UML14CreatorCopier extends Copier {
      getTargetClass(src : in::Core::Class) : out::Core::Class {
        theSource : Standard::ModelElement;
②       theSource := src;     // [*]
        try {
          r : out::Core::Class;
          r := new out::Core::Class();
          trace(src, r);
          return r;
        } finally {
②         theSource.toOut();  // [*]
        }
      }
    }
```

Figure A.17: Snippets of the `Distribution` Output Library

the MTL-aspect method `{^getTarget(.*)}`, new theSource variables will be added in the corresponding output methods for storing the very input parameters that were previously matched (see figure A.17 [*]).

A.4 Conclusion

An LDE step is mainly composed of an abstraction layer describing the system under study that is improved to reach a more concrete abstraction layer by integrating more information about platform(s) specificities. Regarding current technologies (see chapter 2), an abstraction layer may be a set of models, each one of them being described using a modeling language defined by an abstract syntax, one or more concrete syntaxes, and semantics. Improvement may be realized by a model transformation expressed in a model transformation language (with abstract syntax, semantics and concrete syntax).

This thesis proposed techniques to describe concrete syntaxes of languages whose abstract syntax is given in the form of a metamodel. Beside this, this appendix underlined the interest for an existing LDE methodology to be "tailorable". We also explored a mean to define such customizations regarding abstract syntaxes and improvements (through transformations) when modeling languages and transformation languages proposed an extension mechanism related to profiles. UML and MTL are such languages, so does Java when used with annotations [Jav04].

We explored here such a mean for languages that already include an extension mechanism both in their abstract syntax and concrete syntax. However, this is only a first step. Indeed, one may need to tailor languages at abstract syntax level (e.g. using higher-order hierarchies), at semantics level, but also at concrete syntax level.

Bibliography

[AAB+07] Adaptive Ltd., Alcatel, Borland Software Corporation, Computer Associates International, Inc., Telefonaktiebolaget LM Ericsson, Fujitsu, Hewlett-Packard Company, I-Logix Inc., International Business Machines Corporation, IONA Technologies, Kabira Technologies, Inc., MEGA International, Motorola, Inc., Object Management Group., Oracle Corporation, SOFTEAM, Telelogic AB, Unisys, and X-Change Technologies Group, LLC, Unified Modeling Language (UML), version 2.1.1, OMG Document formal/07-02-05, February 2007.

[AAR05] Uwe Aßmann, Mehmet Aksit, and Arend Rensink (eds.), Model Driven Architecture, European MDA Workshops: Foundations and Applications, MDAFA 2003 and MDAFA 2004, Twente, The Netherlands, June 26-27, 2003 and Linköping, Sweden, June 10-11, 2004, Revised Selected Papers, Lecture Notes in Computer Science, vol. 3599, Springer, 2005.

[ABB+04] Alcatel, BEA Systems, Inc., BNR Europe Ltd., Borland International, DSTC Pty Ltd, Concept Five Technologies, Digital Equipment Corporation, Eternal Systems, Inc., Expersoft Corporation, FUJITSU LIMITED, Genesis Development Corporation, Hewlett-Packard Company, HighComm, Highlander Communications, L.C., HyperDesk Corporation, Inprise Corporation, International Business Machines Corporation, ICL plc, Inprise Corporation, International Computers, Ltd., IONA Technologies, Ltd., Lockheed Martin Federal Systems, Inc., Lucent Technologies, Inc., Micro Focus Limited, NCR Corporation, NEC Corporation, Netscape Communications Corporation, Nortel Networks, Northern Telecom Corporation, Novell USG, Object Design, Inc., Object Management Group, Inc., Objective Interface Systems, Inc., Object-Oriented Concepts, Inc., Oracle Corporation, PeerLogic, Inc., Siemens Nixdorf Informationssysteme AG, Sun Microsystems, Inc., SunSoft, Inc., Sybase, Inc., Telefnica Investigacin y Desarrollo S.A. Unipersonal, TIBCO, Inc., Tri-Pacific Software, Inc., and Visual Edge Software, Ltd., Common Object Request Broker Architecture: Core specification, v.3.0.3, OMG Document formal/2004-03-12, March 2004.

[ABF+06] Adaptive Ltd., Boldsoft, France Telecom, International Business Machines Corporation, IONA Technologies, and Object Management Group, Object Constraint Language specification, v2.0, OMG Document formal/06-05-01, May 2006.

[ABH+00] AT&T, BNR, Hewlett-Packard, Object Management Group, Inc., and Sun Soft, Trading Object Service specification, v1.0, OMG Document formal/2004-03-12, May 2000.

[Abr96] J.-R. Abrial, The B-book: assigning programs to meanings, Cambridge University Press, New York, NY, USA, 1996.

[ACC+06] Adaptive, Ceira Technologies, Inc., Compuware Corporation, Data Access Technologies, Inc., DSTC, Gentleware, Hewlett-Packard, International Business Machines, IONA, Object Management Group, MetaMatrix, Softeam, SUN, Telelogic AB, and Unisys, Meta-Object Facility (MOF) core, v2.0, OMG Document formal/2006-01-01, January 2006.

[ACD+05] Adaptive, Compuware Corporation, DSTC, Hewlett-Packard, International Business Machines, IONA, Object Management Group, SUN, and Unisys, XML Metadata Interchange (XMI v2.1), OMG Document formal/05-09-01, September 2005.

[ADF+05] Alcatel, DMR Consulting, Fujitsu Limited, International Business Machines Corporation, Q-Labs, Rational Software Corporation, SOFTEAM, and Unisys Corporation, Software Process Engineering Metamodel, v1.1, OMG Document formal/05-01-06, January 2005.

[ADG+06] Adaptive, DaimlerChrysler AG, Gentleware AG, IBM Rational Software, Sun Microsystems, and Telelogic AB, Diagram Interchange specification, v1.0, OMG Document formal/06-04-04, April 2006.

[ADvSP04] João Paulo A. Almeida, Remco M. Dijkman, Marten van Sinderen, and Luis Ferreira Pires, On the Notion of Abstract Platform in MDA Development, In Akehurst et al. [AvSB04], pp. 253–263.

[AK02] Colin Atkinson and Thomas Kühne, The role of meta-modeling in MDA, Workshop in Software Model Engineering (WISME@UML) (Dresden, Germany), October 2002.

[ALPT03] Marcus Alanen, Johan Lilius, Ivan Porres, and Dragos Truscan, Realizing a Model Driven Engineering Process, Tech. Report 565, TUCS - Turku Centre for Computer Science, Turku, Finland, November 2003.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman, Compilers: Principles, Techniques, and Tools (2nd Edition), Addison Wesley, August 2006.

[And96] Marc Andries, Graph Rewrite Systems and Visual Database Languages, Ph.D. thesis, Universität Bremen, 1996.

[AP04] Marcus Alanen and Ivan Porres, A Relation Between Context-Free Grammars and Meta Object Facility Metamodels, Tech. Report 606, TUCS - Turku Centre for Computer Science, Turku, Finland, March 2004.

[Apa] Apache Foundation - XML Graphics Project, Batik SVG toolkit, http://xml-graphics.apache.org/batik/.

[AvSB04] David H. Akehurst, Marten van Sinderen, and Barrett R. Bryant (eds.), 8th international enterprise distributed object computing conference (EDOC 2004), 20-24 september 2004, Monterey, California, USA, proceedings, IEEE Computer Society, 2004.

[BA04] Kent Beck and Cynthia Andres, Extreme Programming Explained: Embrace Change (2nd Edition), Addison-Wesley Professional, 2004.

[Baa06] Thomas Baar, Correctly Defined Concrete Syntax for Visual Modeling Languages, In Nierstrasz et al. [NWHR06], pp. 111–125.

[Bar98] Roswitha Bardohl, GENGED - A Generic Graphical Editor for Visual Languages Based on Algebraic Graph Grammars, VL, 1998, pp. 48–55.

[BBG+60] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger, Report on the algorithmic language ALGOL 60, Commun. ACM 3 (1960), no. 5, 299–314.

[BD99] Adolf-Peter Bröhl and Wolfgang Dröschel, Das V-Modell. Der Standard in der Softwareentwicklung mit Praxisleitfaden (Auflage: 2), Oldenbourg Verlag, 1999.

[BDII02] BEA Systems, DSTC, Iona Technologies Ltd., and Inprise, Naming Service specification, v1.3, OMG Document formal/2004-10-03, September 2002.

[BDJ+03] Jean Bézivin, Grégoire Dupé, Frédéric Jouault, Gilles Pitette, and Jamal Eddine Rougui, First experiments with the ATL model transformation language: Transforming XSLT into XQuery, OOPSLA 2003 Workshop (Anaheim, California), 2003.

[Bea06] Olivier Beaudoux, DoPIdom: une approche de l'interaction et de la collaboration centrée sur les documents, IHM '06: Proceedings of the 18th international conference on Association Francophone d'Interaction Homme-Machine (New York, NY, USA), ACM Press, 2006, pp. 19–26.

[BEdLT04] Roswitha Bardohl, Hartmut Ehrig, Juan de Lara, and Gabriele Taentzer, Integrating Meta-modelling Aspects with Graph Transformation for Efficient

Visual Language Definition and Model Manipulation, FASE (Michel Wermelinger and Tiziana Margaria, eds.), Lecture Notes in Computer Science, vol. 2984, Springer, 2004, pp. 214–228.

[Ben86] Jon Bentley, Programming pearls: little languages, Commun. ACM 29 (1986), no. 8, 711–721.

[BGdL06] Paolo Bottoni, Esther Guerra, and Juan de Lara, Metamodel-based definition of interaction with visual environments, MoDELS'06 Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI'06 at MoDELS'06), CEUR Workshop Proceedings, vol. 214, 2006.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins, Making Components Contract Aware, Computer 32 (1999), no. 7, 38–45.

[BLP04] Fabrice Bouquet, Bruno Legeard, and Fabien Peureux, CLPS-B - A constraint solver to animate a B specification, STTT 6 (2004), no. 2, 143–157.

[BM03] Peter Braun and Frank Marschall, Transforming Object Oriented Models with BOTL, Electr. Notes Theor. Comput. Sci. 72 (2003), no. 3, 103–117.

[BMSX97] Alan Borning, Kim Marriott, Peter J. Stuckey, and Yi Xiao, Solving Linear Arithmetic Constraints for User Interface Applications, ACM Symposium on User Interface Software and Technology, 1997, pp. 87–96.

[Boh07] Matthias Bohlen, AndroMDA, http://www.andromda.org/, 2007.

[Bos00] Jan Bosch, Design and use of software architectures: adopting and evolving a product-line approach, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[BPSM+06] Tim Bray, Jean Paoli, C.-M. Sperberg-McQueen, Eve Maler, Francois Yergeau, and John Cowan, Extensible Markup Language (XML 1.1 - second edition), World Wide Web Consortium, August 2006.

[BRTK] Marko Boger, Jason Robbins, Linus Tolke, and Markus Klink, ArgoUML), http://argouml.tigris.org/.

[CBR03] Adrian Colyer, Gordon Blair, and Awais Rashid, Managing Complexity in Middleware, The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS) (Yvonne Coady, Eric Eide, and David H. Lorenz, eds.), March 2003.

[CCD+06] Codagen Technologies Corp, Compuware, DSTC, France Telecom, IBM, INRIA, Interactive Objects, Kings College London, Object Management

Group, Softeam, Sun Microsystems, Tata Consultancy Services, Thales, TNI-Valiosys, University of Paris VI, and University of York, Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, OMG Document ptc/05-11-01, October 2006, MOF QVT Final Adopted Specification.

[CDP04] Gennaro Costagliola, Vincenzo Deufemia, and Giuseppe Polese, A framework for modeling and implementing visual notations with applications to software engineering, ACM Trans. Softw. Eng. Methodol. 13 (2004), no. 4, 431–487.

[CESW05] Tony Clark, Andy Evans, Paul Sammut, and James Willans, Applied Meta--modelling: A Foundation for Language-Driven Development, 2005.

[CH06] K. Czarnecki and S. Helsen, Feature-based survey of model transformation approaches, IBM Syst. J. 45 (2006), no. 3, 621–645.

[Cho56] Noam Chomsky, Three models for the description of language, IEEE Transactions on Information Theory 2 (1956), no. 3, 113–124.

[CIM+06] Compuware Corporation, Interactive Objects Software GmbH, Mentor Graphics Corporation, Object Management Group, Pathfinder Solutions, SINTEF, Softeam, and Tata Consultancy Services, MOF Models to Text Transformation Language final adopted specification, OMG Document ptc/06-11-01, November 2006.

[CLOP02] Gennaro Costagliola, Andrea De Lucia, Sergio Orefice, and Giuseppe Polese, A Classification Framework to Support the Design of Visual Languages, J. Vis. Lang. Comput. 13 (2002), no. 6, 573–600.

[CM03] Sitt Sen Chok and Kim Marriott, Automatic generation of intelligent diagram editors, ACM Trans. Comput.-Hum. Interact. 10 (2003), no. 3, 244–276.

[Dal05] Christopher J Daly, AST framework generation with Gymnast, Tech Exchange Panel: Language Toolkits. EclipseCON, 2005.

[Dav03] James Davis, GME: the generic modeling environment., OOPSLA Companion (Ron Crocker and Guy L. Steele Jr., eds.), ACM, 2003, pp. 82–83.

[DDF+04] Data Access Technologies, DSTC Pty Ltd, France Telecom, IBM, IONA Technologies, Object Management Group, Inc., Open-IT, and Unisys, Human-Usable Textual Notation, v1.0, OMG Document formal/04-08-01, August 2004.

[DDG+02] Data Access Corporation, DSTC Pty Ltd, Genesis Development Corporation, Telelogic AB, and UBS AG, UML profile for CORBA specification, v.1.0, OMG Document formal/02-04-01, April 2002.

[dLV02] Juan de Lara and Hans Vangheluwe, Using AToM3 as a Meta-Case Tool, Proceedings of the 4th International Conference on Enterprise Information Systems (ICEIS), 2002, pp. 642–649.

[DMNS97] Serge Demeyer, Theo Dirk Meijler, Oscar Nierstrasz, and Patrick Steyaert, Design Guidelines for 'Tailorable Frameworks', Commun. ACM 40 (1997), no. 10, 60–64.

[DV02] Péter Domokos and Dániel Varró, An Open Visualization Framework for Metamodel-Based Modeling Languages, Electr. Notes Theor. Comput. Sci. 72 (2002), no. 2, 69–78.

[Ecl] Eclipse Consortium, Eclipse Graphical Editing Framework (GEF), http://www.eclipse.org/gef.

[Ecl05] Eclipse, Eclipse Modeling Framework (EMF), December 2005.

[Ecl06] Eclipse, Graphical Modeling Framework (GMF), June 2006.

[EEHT05] Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer, Generation of visual editors as eclipse plug-ins, ASE (David F. Redmiles, Thomas Ellman, and Andrea Zisman, eds.), ACM, 2005, pp. 134–143.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, Handbook of graph grammars and computing by graph transformation, World Scientific, 1999.

[Ern03] Erik Ernst, Higher-Order Hierarchies, ECOOP 2003 - Object-Oriented Programming, 17th European Conference on (Luca Cardelli, ed.), LNCS, vol. 2743, Springer, July 2003, pp. 303–328.

[ESW+05] Andy Evans, Paul Sammut, James S. Willans, Alan Moore, and Girish Maskeri, A Unified Superstructure for UML, Journal of Object Technology 4 (2005), no. 1, 165–182.

[FB05] Frédéric Fondement and Thomas Baar, Making Metamodels Aware of Concrete Syntax, European Conference on Model Driven Architecture (ECMDA), Lecture Notes in Computer Science, vol. 3748, 2005, pp. 190 – 204.

[FBBO99] M. Fowler, K. Beck, J. Brant, and W.F. Opdyke, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[Fle06] Franck Fleurey, Langage et méthode pour une ingénierie des modèles fiable, Ph.D. thesis, Université de Rennes 1, October 2006.

[Fon06] Choong Koon Fong, Quick Start Guide to MDA, A Primer to Model-Driven Architecture Using Borland Together Technologies, Borland White Paper, September 2006.

[fRiCSI05] French National Institute for Research in Computer Science and Control (INRIA), Model transformation language (MTL), http://modelware.inria.fr/, 2005.

[FS04] Frédéric Fondement and Raul Silaghi, Defining Model Driven Engineering Processes, Third International Workshop in Software Model Engineering (WiSME), held at the 7th International Conference on the Unified Modeling Language (UML), 2004.

[FSGM06] Frédéric Fondement, Rémi Schnekenburger, Sébastien Gérard, and Pierre-Alain Muller, Metamodel-Aware Textual Concrete Syntax Specification, Tech. Report LGL-REPORT-2006-005, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[GBNT01] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck, Handling crosscutting constraints in domain-specific modeling, Communications of the ACM 44 (2001), no. 10, 87–93.

[Göt82] Herbert Göttler, Attributed graph grammars for graphics, Graph-Grammars and Their Application to Computer Science (Hartmut Ehrig, Manfred Nagl, and Grzegorz Rozenberg, eds.), Lecture Notes in Computer Science, vol. 153, Springer, 1982, pp. 130–142.

[GRS06] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra, Deriving a textual notation from a metamodel: an experience on bridging modelware and grammarware, European Workshop on Milestones, Models and Mappings for Model-Driven Architecture - European Conference on Model Driven Architecture (3M4MDA), July 2006.

[GSCK04] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent, Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools, Wiley, August 2004.

[GSR05] Leif Geiger, Christian Schneider, and Carsten Reckord, Template and model-based code generation for MDA-Tools, 3rd International Fujaba Days (IFD05), 2005.

[Har87] David Harel, Statecharts: A Visual Formulation for Complex Systems, Science of Computer Programming 8 (1987), no. 3, 231–274.

[HHW+04] Arnaud Le Hors, Philippe Le Hégaret, Lauren Wood, Gavin Nicol, Jonathan Robie, Mike Champion, and Steve Byrne, Document Object Model (DOM) level 3 core specification, World Wide Web Consortium, April 2004.

[Hil02] Stéphane Hillion, Koala DynamicJava, http://koala.ilog.fr/djava/index.html, June 2002.

[Hof79] Douglas R. Hofstadter, Godel, Escher, Bach: An Eternal Golden Braid, Basic Books, Inc., New York, NY, USA, 1979.

[Hon92] Honeywell, DOME guide, http://www.htc.honeywell.com/dome/, 1992.

[Hon05] Fabrice Hong, Provide behaviour to XML-SVG, Bachelor Semester Project, École Polytechnique Fédérale de Lausanne (EPFL), 2005.

[HR04] David Harel and Bernhard Rumpe, Meaningful Modeling: What's the Semantics of "Semantics"?, Computer 37 (2004), no. 10, 64–72.

[HRS02] David Hearnden, Kerry Raymond, and Jim Steel, Anti-Yacc: MOF-to-Text, Proceedings of the Sixth International Enterprise Distributed Onject Computing Conference (EDOC'02) (Washington, DC, USA), IEEE Computer Society, 2002, p. 200.

[Iiv96] Juhani Iivari, Why Are Case Tools Not Used?, Commun. ACM 39 (1996), no. 10, 94–103.

[Int01] International Organization for Standardization, Information Technology - Syntactic Metalanguage - Extended BNF, ISO/IEC 14977, August 2001.

[ISO05] ISO/IEC, Meta-Object Facility (MOF) v1.4.1, OMG Document formal/05-05-05, ISO PAS 19502 standard, July 2005.

[J03] Jean-Marc Jézéquel, Model-Driven Engineering: Basic Principles and Challenges, Invited Presentation at Formal Methods for Components and Objects (FMCO'03), November 2003.

[Jac02] Daniel Jackson, Alloy: a lightweight object modelling notation, Software Engineering and Methodology 11 (2002), no. 2, 256–290.

[Jac04] Ivar Jacobson, Object-Oriented Software Engineering: A Use Case Driven Approach, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[Jav02] Java Community Process, Java(TM) Metadata Interface API specification 1.0 final release, JSR-000040, June 2002.

[Jav04] Java Community Process, A metadata facility for the Java(TM) programming language, JSR-000175, September 2004.

[JBK06] Frédéric Jouault, Jean Bézivin, and Ivan Kurtev, TCS: a DSL for the specification of textual concrete syntaxes in model engineering, GPCE (Stan Jarzabek, Douglas C. Schmidt, and Todd L. Veldhuizen, eds.), ACM, 2006, pp. 249–254.

[JM97] Jean-Marc Jézéquel and Bertrand Meyer, Design by Contract: The Lessons of Ariane, IEEE Computer 30 (1997), no. 1, 129–130.

[JN05] Dean Jackson and Craig Northway, Scalable Vector Graphics (SVG) Full 1.2 specification, World Wide Web Consortium, Working Draft WD-SVG12-20050413, April 2005.

[Joh79] Steven C. Johnson, Yacc: Yet Another Compiler Compiler, UNIX Programmer's Manual (New York, NY, USA), vol. 2, Holt, Rinehart, and Winston, New York, NY, USA, 1979, pp. 353–387.

[Jon90] Chris Jones, An example-based introduction to graph grammars for modeling, Proceedings of the Twenty-Third Annual Hawaii International Conference on System Sciences, vol. iii, 1990, pp. 433–442.

[KBA02] Ivan Kurtev, Jean Bézivin, and Mehmet Aksit, Technical spaces: An initial appraisal, CoopIS, DOA 2002 Federated Conferences, Industrial track, 2002.

[KBC04] Audris Kalnins, Janis Barzdins, and Edgars Celms, Model Transformation Language MOLA, In Aßmann et al. [AAR05], pp. 62–76.

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and Spencer A. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, U.S.A., November 1990.

[Ken02] Stuart Kent, Model Driven Engineering, IFM (Michael J. Butler, Luigia Petre, and Kaisa Sere, eds.), Lecture Notes in Computer Science, vol. 2335, Springer, 2002, pp. 286–298.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold, Getting started with ASPECTJ, Commun. ACM 44 (2001), no. 10, 59–65.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, Aspect-Oriented Programming, ECOOP, 1997, pp. 220–242.

[KMB+96] Richard B. Kieburtz, Laura McKinney, Jeffrey M. Bell, James Hook, Alex Kotov, Jeffrey Lewis, Dino Oliva, Tim Sheard, Ira Smith, and Lisa Walton, A Software Engineering Experiment in Software Component Generation, ICSE, 1996, pp. 542–552.

[Knu68] Donald E. Knuth, Semantics of Context-Free Languages, Mathematical Systems Theory 2 (1968), no. 2, 127–145.

[KP02] Steven Kelly and Risto Pohjonen, Domain-Specific Modelling for Cross-Platform Product Families, ER (Workshops) (Stefano Spaccapietra, Salvatore T. March, and Yahiko Kambayashi, eds.), Lecture Notes in Computer Science, vol. 2503, Springer, 2002, pp. 182–194.

[Kru03] Philippe Kruchten, The Rational Unified Process: An Introduction, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[KSLB03] Gabor Karsai, Janos Sztipanovits, Ákos Lédeczi, and Ted Bapty, Model-integrated development of embedded software, Proceedings of the IEEE 91 (2003), no. 1, 145–164.

[KW03] Anneke Kleppe and Jos Warmer, Do MDA Transformations Preserve Meaning? An investigation into preserving semantics, MDA Workshop (Andy Evans, Paul Sammut, and James S Willans, eds.), November 2003, pp. 13–22.

[LBM+01] Ákos Lédeczi, Arpad Bakay, Miklos Maroti, Péter Völgyesi, Greg Nordstrom, Jonathan Sprinkle, and Gabor Karsai, Composing Domain-Specific Design Environments, IEEE Computer 34 (2001), no. 11, 44–51.

[Leh80] Meir M. Lehman, On understanding laws, evolution, and conservation in the large-program life cycle, Journal of Systems and Software 1 (1980), 213–221.

[LHBJ05] Denivaldo Lopes, Slimane Hammoudi, Jean Bézivin, and Frédéric Jouault, Generating Transformation Definition from Mapping Specification: Application to Web Service Platform, CAiSE (Oscar Pastor and João Falcão e Cunha, eds.), Lecture Notes in Computer Science, vol. 3520, Springer, 2005, pp. 309–325.

[MB05] Slavisa Markovic and Thomas Baar, Refactoring OCL annotated UML class diagrams, MoDELS (Lionel C. Briand and Clay Williams, eds.), Lecture Notes in Computer Science, vol. 3713, Springer, 2005, pp. 280–294.

[MB06] Slavisa Markovic and Thomas Baar, An OCL Semantics Specified with QVT, In Nierstrasz et al. [NWHR06], pp. 661–675.

[MCF03] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami, Guest Editors' Introduction: Model-Driven Development, IEEE Software 20 (2003), no. 5, 14–18.

[MFF+06] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel, Model Driven Analysis and Synthesis of Concrete Syntax, Models/UML 2006, LNCS, vol. 4199, 2006, pp. 98–110.

[MFJ05] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel, Weaving Executability into Object-Oriented Meta-languages, International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS, vol. 3713, Springer, October 2005, pp. 264–278.

[MFV+05] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel, On Executable Meta-Languages applied to Model Transformations, Model Transformations In Practice Workshop, October 2005.

[MH05] Pierre-Alain Muller and Michel Hassenforder, HUTN as a bridge between modelware and grammarware, WISME Workshop, MODELS / UML'2005, 2005.

[Min02] Mark Minas, Concepts and realization of a diagram editor generator based on hypergraph transformation, Sci. Comput. Program. 44 (2002), no. 2, 157–180.

[MM03] Jishnu Mukerji and Joaquin Miller, MDA guide, v1.0.1, OMG Document omg/03-06-01, June 2003.

[MMM04] Cameron L. McCormack, Kim Marriott, and Bernd Meyer, Constraint SVG, WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (New York, NY, USA), ACM Press, 2004, pp. 310–311.

[MSFB05] Pierre-Alain Muller, Philippe Studer, Frédéric Fondement, and Jean Bézivin, Platform independent Web application modeling and development with Netsilon, Software and System Modeling 4 (2005), no. 4, 424–442.

[MTAL98] Stephen J. Mellor, Stephen R. Tockey, Rodolphe Arthaud, and Philippe Leblanc, An Action Language for UML: Proposal for a Precise Execution Semantics, UML (Jean Bézivin and Pierre-Alain Muller, eds.), Lecture Notes in Computer Science, vol. 1618, Springer, 1998, pp. 307–318.

[MVC03] Esperanza Marcos, Belén Vela, and José Maria Cavero, A Methodological Approach for Object-Relational Database Design using UML, Software and System Modeling 2 (2003), no. 1, 59–75.

[Nag76] M. Nagl, Formal Languages of Labelled Graphs, Computing 16 (1976), no. 1–2, 113–137.

[Net05] Java metamodel, http://java.netbeans.org/, November 2005.

[NFTJ06] Clémentine Nebut, Franck Fleurey, Yves Le Traon, and Jean-Marc Jézéquel, Automatic Test Generation: A Use Case Driven Approach, IEEE Transactions on Software Engineering 32 (2006), no. 3, 140–155.

[NWHR06] Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio (eds.), Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings, Lecture Notes in Computer Science, vol. 4199, Springer, 2006.

[Obe07] Obeo, Acceleo, http://www.acceleo.org/, 2007.

[Par72] David Lorge Parnas, On the Criteria To Be Used in Decomposing Systems into Modules, Commun. ACM 15 (1972), no. 12, 1053–1058.

[Par05] Terence Parr, ANother Tool for Language Recognition (ANTLR), 2005.

[Poh03] Risto Pohjonen, Boosting Embedded Systems Development with Domain-Specific Modeling, RTC Magazine (2003), 57–61.

[Por03] Ivan Porres, A Toolkit for Model Manipulation, Springer International Journal on Software and Systems Modeling 2 (2003), no. 4, 370–389.

[PQ95] Terence J. Parr and Russell W. Quong, ANTLR: A predicated-LL(k) parser generator, Software - Practice and Experience 25 (1995), no. 7, 789–810.

[RH06] Fabien Rohrer and Francois Helg, Synchronization between display objects and representation templates in graphical language construction, Bachelor Semester Project, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[RM07] Romain Rouvoy and Philippe Merle, Un langage de description et de vérification de motifs d’architecture : Fractal ADL, LMO (Isabelle Borne, Xavier

Crégut, Sophie Ebersold, and Frédéric Migeon, eds.), Hermès Lavoisier, 2007, pp. 49–64.

[Sch94] Andy Schürr, Specification of Graph Translators with Triple Graph Grammars, WG (Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, eds.), Lecture Notes in Computer Science, vol. 903, Springer, 1994, pp. 151–163.

[Sch06] Douglas C. Schmidt, Guest Editor's Introduction: Model-Driven Engineering, IEEE Computer 39 (2006), no. 2, 25–31.

[Sch07] Markus Scheidgen, Textual Editing Framework, http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/, August 2007.

[SFS04a] Raul Silaghi, Frédéric Fondement, and Alfred Strohmeier, Towards an MDA-Oriented UML Profile for Distribution, In Akehurst et al. [AvSB04], pp. 227–239.

[SFS04b] Raul Silaghi, Frédéric Fondement, and Alfred Strohmeier, "Weaving" MTL Model Transformations, In Aßmann et al. [AAR05], pp. 123–138.

[Sil06] Raul Silaghi, Aspect-Oriented Model-Driven Engineering of Middleware-Mediated Distributed Systems, Ph.D. thesis, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[SK97] Janos Sztipanovits and Gabor Karsai, Model-Integrated Computing, IEEE Computer 30 (1997), no. 4, 110–111.

[SK03] Shane Sendall and Wojtek Kozaczynski, Model Transformation: The Heart and Soul of Model-Driven Software Development, IEEE Software 20 (2003), no. 5, 42–45.

[SKB+95] Janos Sztipanovits, Gabor Karsai, Csaba Biegl, Ted Bapty, Ákos Lédeczi, and Amit Misra, MULTIGRAPH: an architecture for model-integrated computing, ICECCS, IEEE Computer Society, 1995, pp. 361–368.

[SS99] Shane Sendall and Alfred Strohmeier, UML Based Fusion Analysis Applied to a Bank Case Study, UML (Robert B. France and Bernhard Rumpe, eds.), Lecture Notes in Computer Science, vol. 1723, Springer, 1999, pp. 278–291.

[SS03] Raul Silaghi and Alfred Strohmeier, Integrating CBSE, SoC, MDA, and AOP in a Software Development Method, EDOC, IEEE Computer Society, 2003, pp. 136–146.

[SS05] Raul Silaghi and Alfred Strohmeier, Parallax, or Viewing Designs Through a Prism of Middleware Platforms, HICSS, IEEE Computer Society, 2005.

[Sun05] Sun Microsystems, Metadata repository (MDR), December 2005.

[Sut63] Ivan E. Sutherland, Sketchpad, A Man-Machine Graphical Communication System, Outstanding Dissertations in the Computer Sciences, Garland Publishing, New York, 1963.

[Szy02] Clemens Szyperski, Component Software: Beyond Object-Oriented Programming, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[TJ07] Emina Torlak and Daniel Jackson, Kodkod: A Relational Model Finder, TACAS (Orna Grumberg and Michael Huth, eds.), Lecture Notes in Computer Science, vol. 4424, Springer, 2007.

[VBBJ06] Eric Vépa, Jean Bézivin, Hugo Brunelière, and Frédéric Jouault, Measuring Model Repositories, Proceedings of the Model Size Metrics Workshop at the MoDELS/UML 2006 conference, Genova, Italy, 2006.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser, Domain-Specific Languages: An Annotated Bibliography, SIGPLAN Notices 35 (2000), no. 6, 26–36.

[VJ04] Didier Vojtisek and Jean-Marc Jézéquel, MTL and umlaut NG - engine and framework for model transformation, ERCIM News 58 (2004), 46–47.

[Voe06a] Markus Voelter, oAW 4 introduction and overview, http://www.eclipse.org/gmt/oaw/, August 2006.

[Voe06b] Markus Voelter, oAW 4 xText - a framework for textual dsls, http://www.eclipse.org/gmt/oaw/, October 2006.

[Voj04] Didier Vojtisek, BasicMTL realization guide, Tech. report, Institut National de Recherche en Informatique et en Automatique (INRIA), February 2004.

[VPF+06] F. Vernadat, Christian Percebois, Patrick Farail, R. Vingerhoeds, Alain Rossignol, Jean-Pierre Talpin, and David Chemouil, The TOPCASED Project - A Toolkit in OPen-source for Critical Applications and SystEm Development, Data Systems In Aerospace (DASIA), Berlin, Germany, 22/05/2006-25/05/2006 (http://www.esa.int/publications), European Space Agency (ESA Publications), May 2006.

[VS06] Markus Völter and Thomas Stahl, Model-Driven Software Development, Wiley, 2006.

[ZG03] Paul Ziemann and Martin Gogolla, An OCL Extension for Formulating Temporal Constraints, Tech. Report 1/03, Universität Bremen, 2003.

Curriculum Vitae

Frédéric Fondement

Education

1997	University degree of technology (DUT) in electrical engineering and industrial data processing (GEII) with emphasis in electrical engineering - Belfort / France.
2000	Engineering degree in control engineering and industrial data processing from the École Supérieure des Sciences Appliquées pour l'Ingénieur de Mulhouse (ESSAIM - now ENSISA) / France.
2001	Diploma in technological research (DRT) GEII from the Upper Alsace University entitled "Development of Components for an UML Virtual Machine" - Mulhouse / France.

Research and Professional Experience

2001	Engineering internship for ObjeXion Software - Development of an OCL interpreter for an UML virtual machine (ObjeXion Facsimile).
2001 - 2002	Software engineer for ObjeXion Software - Development and support for a model-driven web application generator (Netsilon [MSFB05]) - Vieux-Thann / France.
2003	Research and development engineer at the French National Institute of Research in Computer Science and Control (INRIA / IRISA) - Model Transformation Language (MTL [VJ04]) - Rennes / France.
2004 - 2007	Ph.D. candidate at the Software Engineering Laboratory (LGL) of the École Polytechnique Fédérale de Lausanne (EPFL) in the field of model driven engineering with special interest in concrete syntax modeling - Lausanne / Switzerland.
2007 - 2008	Teaching and research assistant at the Upper Alsace University - Mulhouse / France
PC member	Ingénierie Dirigée par les Modèles (IDM) 2005 and 2006
Reviewer	Model Driven Software Development Software and Systems Modeling (SoSyM), IET Software, WWW Journal Models/UML 2005, VL-HCC 2005, UML 2003, UML 2001, UML 2000

Teaching

2004 - 2005 Assistant of the Software Engineering Project for 3rd year computer science students.

2005 - 2006 Supervision of 3rd year semester projects entitled "Provide Behaviour to XML / SVG" [Hon05] and "Synchronization between display objects and representation templates in graphical language construction" [RH06].

Publications

[BMFS06] Thomas Baar, Slavisa Markovic, Frédéric Fondement, and Alfred Strohmeier, Definition and correct refinement of operation specifications., Research Results of the DICS Program (Jürg Kohlas, Bertrand Meyer, and André Schiper, eds.), Lecture Notes in Computer Science, vol. 4028, Springer, 2006, pp. 127–144.

[FB05] Frédéric Fondement and Thomas Baar, Making metamodels aware of concrete syntax., ECMDA-FA (Alan Hartman and David Kreische, eds.), Lecture Notes in Computer Science, vol. 3748, Springer, 2005, pp. 190–204.

[FS04] Frédéric Fondement and Raul Silaghi, Defining Model Driven Engineering Processes, Third International Workshop in Software Model Engineering (WiSME), held at the 7th International Conference on the Unified Modeling Language (UML), 2004.

[FSGM06] Frédéric Fondement, Rémi Schnekenburger, Sébastien Gérard, and Pierre-Alain Muller, Metamodel-Aware Textual Concrete Syntax Specification, Tech. report LGL-REPORT-2006-005, École Polytechnique Fédérale de Lausanne (EPFL), 2006.

[MDFH04] Pierre-Alain Muller, Cédric Dumoulin, Frédéric Fondement, and Michel Hassenforder, The topmodl initiative., UML Satellite Activities (Nuno Jardim Nunes, Bran Selic, Alberto Rodrigues da Silva, and José Ambrosio Toval Álvarez, eds.), Lecture Notes in Computer Science, vol. 3297, Springer, 2004, pp. 242–245.

[MFF+06] Pierre-Alain Muller, Franck Fleurey, Frédéric Fondement, Michel Hassenforder, Rémi Schneckenburger, Sébastien Gérard, and Jean-Marc Jézéquel, Model-driven analysis and synthesis of concrete syntax., MoDELS (Oscar Nierstrasz, Jon Whittle, David Harel, and Gianna Reggio, eds.), Lecture Notes in Computer Science, vol. 4199, Springer, 2006, pp. 98–110.

[MFV+05] Pierre-Alain Muller, Franck Fleurey, Didier Vojtisek, Zoé Drey, Damien Pollet, Frédéric Fondement, Philippe Studer, and Jean-Marc Jézéquel, On executable meta-languages applied to model transformations, Model Transformations In Practice Workshop, oct 2005.

[MSFB05] Pierre-Alain Muller, Philippe Studer, Frédéric Fondement, and Jean Bézivin, Platform independent web application modeling and development with netsilon., Software and System Modeling 4 (2005), no. 4, 424–442.

[SFS04a] Raul Silaghi, Frédéric Fondement, and Alfred Strohmeier, Towards an mda-oriented uml profile for distribution., EDOC, IEEE Computer Society, 2004, pp. 227–239.

[SFS04b] Raul Silaghi, Frédéric Fondement, and Alfred Strohmeier, "weaving" mtl model transformations., MDAFA (Uwe Aßmann, Mehmet Aksit, and Arend Rensink, eds.), Lecture Notes in Computer Science, vol. 3599, Springer, 2004, pp. 123–138.

[ZJF03] Tewfik Ziadi, Jean Marc Jézéquel, and Frédéric Fondement, Product Line Derivation with UML, Software Variability Management Workshop, February 2003, pp. 94–102.