



INSTITUTO
SUPERIOR
TÉCNICO

UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

**Development of CMS-based Web Applications
with a
Multi-Language Model-Driven Approach**

João Paulo Pedro Mendes de Sousa Saraiva

Supervisor: Doctor Alberto Manuel Rodrigues da Silva

**Thesis approved in public session to obtain the PhD Degree in
Information Systems and Computer Engineering**

Jury final classification: Pass with merit

Jury

Chairperson: Chairman of the IST Scientific Board

Members of the Committee:

Doctor José Manuel Nunes Salvador Tribolet
Doctor Alberto Manuel Rodrigues da Silva
Doctor Vasco Miguel Moreira do Amaral
Doctor Leonel Domingos Telo Nóbrega
Doctor António Paulo Teles de Menezes de Correia Leitão



INSTITUTO
SUPERIOR
TÉCNICO

UNIVERSIDADE TÉCNICA DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

**Development of CMS-based Web Applications
with a
Multi-Language Model-Driven Approach**

João Paulo Pedro Mendes de Sousa Saraiva

Supervisor: Doctor Alberto Manuel Rodrigues da Silva

**Thesis approved in public session to obtain the PhD Degree in
Information Systems and Computer Engineering**

Jury final classification: Pass with merit

Jury

Chairperson: Chairman of the IST Scientific Board

Members of the Committee:

Doctor José Manuel Nunes Salvador Tribolet, Professor Catedrático do Instituto Superior Técnico, da Universidade Técnica de Lisboa

Doctor Alberto Manuel Rodrigues da Silva, Professor Associado do Instituto Superior Técnico, da Universidade Técnica de Lisboa

Doctor Vasco Miguel Moreira do Amaral, Professor Auxiliar da Faculdade de Ciências e Tecnologia, da Universidade Nova de Lisboa

Doctor Leonel Domingos Telo Nóbrega, Professor Auxiliar do Centro de Ciências Matemáticas, da Universidade da Madeira

Doctor António Paulo Teles de Menezes de Correia Leitão, Professor Auxiliar do Instituto Superior Técnico, da Universidade Técnica de Lisboa

Funding Institutions

Fundação para a Ciência e Tecnologia (FCT)

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA

Abstract

The emerging Model-Driven Engineering (MDE) paradigm advocates the usage of models as the most important artifacts in the software development process, while artifacts such as documentation and source code can quickly be produced from those models by using automated transformations.

On the other hand, we are witnessing the rising popularity of a particular kind of web application, Content Management Systems (CMS), whose importance for organizational websites and intranets is already acknowledged. Although CMS systems are typically unable to address all the business requirements of an organization by themselves, they can however be used as platforms for more complex web applications that can address those requirements. In fact, some CMS systems are evolving to become the next generation of web application platforms.

This dissertation proposes the creation of an MDE approach for the development of web applications based on CMS systems. This approach is based on the creation of two CMS-oriented languages (which are located at different levels of abstraction, and are used to both quickly model a web application and provide a common ground for the creation of additional CMS-oriented languages), and a mechanism for the processing of models specified using those languages.

Keywords

Software Design; Software Development; Model-Driven Engineering; Content Management System; Web Application; Modeling Language; Model Synchronization; CMS-ML; CMS-IL; MYNK.

Resumo

O paradigma da Engenharia Conduzida por Modelos (MDE) advoga a utilização de modelos como os artefactos mais importantes no processo de desenvolvimento de *software*, enquanto artefactos como código ou documentação podem ser rapidamente produzidos a partir desses modelos através de transformações automatizadas.

Por outro lado, estamos a assistir a uma crescente popularidade de um determinado tipo de aplicação *web*, os Sistemas de Gestão de Conteúdos (CMS), cuja importância para *websites* e *intranets* organizacionais já é amplamente reconhecida. Embora tipicamente os CMS não tenham por si só capacidade suficiente para cobrir todos os requisitos de negócio de uma organização, podem ser utilizados como plataformas para aplicações *web* mais complexas que cumpram esses requisitos. De facto, alguns CMS estão a evoluir para se tornarem a próxima geração de plataformas para aplicações *web*.

Esta dissertação propõe uma abordagem MDE para o desenvolvimento de aplicações *web* baseadas em CMS. Esta abordagem dá especial ênfase às necessidades dos *stakeholders* e dos programadores, preconizando a utilização de um conjunto de linguagens de modelação e de um mecanismo de sincronização entre modelos, este último com o objectivo de assegurar a consistência entre modelos dos diferentes pontos de vista do mesmo sistema.

Palavras-chave

Desenho de *Software*; Desenvolvimento de *Software*; Engenharia Conduzida por Modelos; Sistema de Gestão de Conteúdos; Aplicação *Web*; Linguagem de Modelação; Sincronização de Modelos; CMS-ML; CMS-IL; MYNK.

Acknowledgements

I would like to thank Professor Alberto Silva, my PhD supervisor. I would also like to thank everyone at INESC-ID and at SIQuant for their support.

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA

I would also like to thank Fundação para a Ciência e Tecnologia (FCT) for their financial support with scholarship SFRH/BD/28604/2006, which helped me perform my research work.

The outside of the work environment is just as important, and I could not let this opportunity go without thanking all of my closest friends. To them, my gratitude; they were always there for me, even though I wasn't.

Also, many thanks to Eng. David Ferreira, for his patience in working with me all these years, all the while putting up with my difficult personality and still being as great a friend as a man could ever hope to have. We have made of our office a forest that will forever be remembered.

Last (but certainly not least), a very special thanks to my family. I couldn't have reached this stage in my life if it wasn't for my parents, Carlos Alberto de Sousa Saraiva and Maria Isabel Pedro Mendes de Sousa Saraiva, who always supported me and provided me with all they could. No words could even begin to express how much I truly owe them; thus, I can only try to express my gratitude, and my regret for the little availability and time that was devoted to them over these years.

Lisboa, June 2012

João Paulo Pedro Mendes de Sousa Saraiva

Contents

1	Introduction	1
1.1	Background	2
1.2	Research Context	5
1.3	Research Problems	5
1.4	Research Questions and Objective	8
1.5	Thesis Statement	8
1.6	Research Method	9
1.7	Contributions	10
1.8	Publications	12
1.9	Dissertation Structure	15
1.10	Adopted Notations and Conventions	17
2	Modeling Language Design	19
2.1	Modeling and Model-Driven Engineering	20
2.2	The OMG Approach	22
2.2.1	Unified Modeling Language (UML)	22
2.2.2	Meta Object Facility (MOF)	24
2.2.3	Model-Driven Architecture (MDA)	27
2.3	Domain-Specific Modeling (DSM)	28
2.4	Metamodeling	30
2.4.1	Modeling Language Components	31
2.4.2	Metamodeling Issues	33
2.4.2.1	Strict and Loose Metamodeling	33
2.4.2.2	Is-A: Instantiation and Specialization	34
2.4.2.3	Logical Levels and Level Compaction	36
2.4.2.4	Ontological and Linguistic Instantiation	38
2.4.2.5	Accidental and Essential Complexity	39
2.4.3	The MoMM Metamodel	41
2.5	Similarities to Programming Language Design	42

3	Model Transformations	47
3.1	Transformations in Model-Driven Engineering	49
3.1.1	Query/View/Transformation (QVT)	49
3.1.2	MOF Model To Text Transformation Language (MOFM2T)	55
3.1.3	ATL Transformation Language (ATL)	57
3.2	Graph-based Transformations	60
3.3	Transformations in Traditional Software Development	60
3.3.1	SQL Scripts	61
3.3.2	Object-Relational Mapping and Migration Frameworks	63
3.3.3	Source Code Management	65
3.3.4	Code Compilation	68
3.4	XSLT	69
4	Model-Driven Approaches for Web Application Development	73
4.1	Comparative Analysis	74
4.1.1	Analysis Criteria	74
4.1.2	Analysis Results	75
4.2	Additional Related Work	79
5	Content Management Systems	83
5.1	Comparative Analysis	84
5.1.1	Analysis Criteria	84
5.1.2	Analysis Results	86
5.2	Additional Related Work	88
6	The CMS Multi-Language Model-Driven Approach	91
6.1	Problems	92
6.2	Proposed Solution: a Multi-Language Approach	93
6.2.1	CMS-ML	94
6.2.2	CMS-IL	94
6.2.3	MYNK	95
6.2.4	Development Workflow	95
6.2.5	The ReMMM Metamodel	97
6.3	Guidelines For Language Specification	99
7	CMS-ML: CMS Modeling Language	105
7.1	Guidelines	107
7.2	Model Types and Modeling Roles	109
7.3	CMS-ML Architecture	111
7.4	WebSite Template Modeling	112

7.4.1	Structure View	113
7.4.2	Roles View	114
7.4.3	Permissions View	116
7.5	Toolkit Modeling	118
7.5.1	Roles View	119
7.5.2	Tasks View	120
7.5.3	Domain View	124
7.5.4	States View	127
7.5.5	WebComponents View	128
7.5.6	Side Effects View	131
7.5.7	Interaction Access View	133
7.5.8	Interaction Triggers View	134
7.6	WebSite Annotations Modeling	135
7.7	Additional Features	137
7.8	Importing Toolkits	139
7.9	Language Design Considerations	141
8	CMS-IL: CMS Intermediate Language	145
8.1	Guidelines	147
8.2	Model Types and Modeling Roles	149
8.3	CMS-IL Architecture	152
8.4	WebSite Template Modeling	153
8.4.1	Structure View	154
8.4.2	Roles View	156
8.4.3	Permissions View	157
8.4.4	Users View	158
8.4.5	Languages View	160
8.4.6	Artifacts View	161
8.4.7	Contents View	164
8.4.8	Visual Themes View	169
8.5	Toolkit Modeling	171
8.5.1	Roles View	172
8.5.2	Code View	173
8.5.3	Events View	180
8.5.4	Variability View	182
8.5.5	Domain View	186
8.5.6	WebComponents View	190
8.6	WebSite Annotations Modeling	196
8.7	Defining Aliases	197

CONTENTS

8.8	Reminders	198
8.9	Importing Toolkits	200
9	MYNK: Model Synchronization Framework	203
9.1	Current Model Transformation Issues	204
9.2	MYNK Overview	205
9.3	MYNK and the ReMMM Metamodel	207
9.4	Abstract Syntax	208
9.4.1	Model	209
9.4.2	Traceability	210
9.4.3	Changes	212
9.4.4	MYNK	214
9.5	Concrete Syntax	216
9.6	Conflict Resolution	224
9.7	Characteristics of MYNK-Compatible Languages	226
10	Validation	229
10.1	Web Modeling Language Comparison	230
10.2	CMS-ML	233
10.2.1	Validation with Non-Technical Stakeholders	234
10.2.2	WebC-Docs: a CMS-based Document Management System	235
10.3	CMS-IL	244
10.4	MYNK	249
10.4.1	Synchronizing CMS-ML and CMS-IL Changes	250
10.4.2	Synchronizing CMS-ML and UML Changes	255
10.5	Research Work Assessment	259
10.5.1	Research Questions	259
10.5.2	Research Objective and Thesis Statement	262
11	Conclusion	263
11.1	Contributions and Conclusions	264
11.2	Future Work	267
	Glossary	271
	References	279
A	State of the Art: Model-Driven Approaches for Web Application Development	297
B	State of the Art: CMS Systems	313

List of Figures

1.1	The phase cycle of the Action Research method.	9
2.1	The relationship between a model and a metamodel.	21
2.2	The different kinds of UML diagrams.	23
2.3	The dependencies between UML and MOF.	24
2.4	The EMOF model subset of MOF 2.0.	24
2.5	The CMOF model packages of MOF 2.0.	25
2.6	The OMG's four level metamodel architecture.	26
2.7	An overview of the MDA approach.	28
2.8	The difference between CASE and DSM tools regarding the metalevels supported.	29
2.9	Typical DSM process: some experts develop domain-oriented tools for use by less-experienced developers.	29
2.10	Example of the abstract syntax for a simple Social Network metamodel.	32
2.11	Strict and Loose metamodeling.	35
2.12	Is-A: example of instantiation and specialization.	35
2.13	Using Level Compaction to compact two logical levels (User Types and User Instances).	37
2.14	Language and Library metaphors.	38
2.15	Example of accidental complexity.	40
2.16	The MoMM metamodel.	41
2.17	The different stages in the compilation of a program.	42
2.18	Overview of approaches and metamodeling aspects.	45
3.1	QVT metamodels and their relationship to MOF and OCL.	50
3.2	Overview of QVT's metamodel architecture.	50
3.3	Example of the concrete syntax for a QVTRelation relation.	52
3.4	QVT Core patterns and dependencies between them.	54
3.5	ATL overview.	58
3.6	Overview of the operation of an RCS system.	66
3.7	Example of code generation as a model-to-model transformation.	68

LIST OF FIGURES

3.8	Overview of aspects and problems regarding model transformations.	71
4.1	Overview of aspects and problems regarding web application modeling languages.	81
5.1	Overview of aspects and problems regarding CMS systems.	89
6.1	Overview of the proposed MDE-oriented development approach.	93
6.2	The ReMMM metamodel.	98
7.1	Relationship between the different CMS-ML models.	109
7.2	Modeling roles and artifacts considered by CMS-ML.	110
7.3	Metalevels considered by CMS-ML.	111
7.4	Views involved in the definition of a WebSite Template.	113
7.5	Abstract syntax for the WebSite Template's Structure view.	114
7.6	Concrete syntax for the WebSite Template's Structure view.	115
7.7	Abstract syntax for the WebSite Template's Roles view.	115
7.8	Concrete syntax for the WebSite Template's Roles view.	115
7.9	Abstract syntax for the WebSite Template's Permissions view.	116
7.10	Concrete syntax for the WebSite Template's Permissions view.	118
7.11	Views involved in the definition of a Toolkit.	118
7.12	Abstract syntax for the Toolkit's Roles view.	120
7.13	Concrete syntax for the Toolkit's Roles view.	121
7.14	Abstract syntax for the Toolkit's Tasks view.	121
7.15	Concrete syntax for the Toolkit's Tasks view.	123
7.16	Abstract syntax for the Toolkit's Domain view.	125
7.17	Concrete syntax for the Toolkit's Domain view.	126
7.18	Abstract syntax for the Toolkit's States view.	127
7.19	Concrete syntax for the Toolkit's States view.	128
7.20	Abstract syntax for the Toolkit's WebComponents view.	129
7.21	Concrete syntax for the Toolkit's WebComponents view.	131
7.22	Abstract syntax for the Toolkit's Side Effects view.	132
7.23	Concrete syntax for the Toolkit's Side Effects view.	133
7.24	Abstract syntax for the Toolkit's Interaction Access view.	133
7.25	Concrete syntax for the Toolkit's Interaction Access view.	134
7.26	Abstract syntax for the Toolkit's Interaction Triggers view.	134
7.27	Concrete syntax for the Toolkit's Interaction Triggers view.	135
7.28	Abstract syntax for the WebSite Annotations model.	136
7.29	Concrete syntax for Annotations.	137

7.30	Abstract syntax for CMS-ML Additional Features.	138
7.31	Concrete syntax for CMS-ML Additional Features.	139
7.32	Abstract syntax for the CMS-ML Toolkit Import mechanism.	140
7.33	Concrete syntax for Toolkit Import elements.	141
7.34	Concrete syntax for imported Toolkit elements.	142
8.1	Relationship between the different CMS-IL models.	150
8.2	Modeling roles and artifacts considered by CMS-IL.	151
8.3	Metalevels considered by CMS-IL.	152
8.4	Views involved in the definition of a WebSite Template.	154
8.5	Abstract syntax for the WebSite Template's Structure view.	155
8.6	Abstract syntax for the WebSite Template's Roles view.	156
8.7	Abstract syntax for the WebSite Template's Permissions view.	157
8.8	Abstract syntax for the WebSite Template's Users view.	159
8.9	Abstract syntax for the WebSite Template's Languages view.	160
8.10	Abstract syntax for the WebSite Template's Artifacts view.	162
8.11	Abstract syntax for the WebSite Template's Contents view.	165
8.12	Abstract syntax for the WebSite Template's Visual Themes view.	169
8.13	Modeling views involved in the definition of a Toolkit.	171
8.14	Abstract syntax for the Toolkit's Roles view.	172
8.15	Abstract syntax for the Toolkit's Code view.	173
8.16	Abstract syntax for the Toolkit's Events view.	181
8.17	Abstract syntax for the Toolkit's Variability view.	183
8.18	A WebSite Template contains implicit Variability Points.	185
8.19	Abstract syntax for the Toolkit's Domain view.	187
8.20	Abstract syntax for the Toolkit's WebComponents view.	191
8.21	Abstract syntax for the WebSite Annotations model.	196
8.22	Abstract syntax for CMS-IL Reminders.	199
8.23	Abstract syntax for the CMS-IL Toolkit Import mechanism.	200
9.1	Typical model transformations may lose changes.	205
9.2	Possible synchronization scenarios.	206
9.3	How the same model is viewed by MYNK and UML.	208
9.4	Overview of the MYNK language.	209
9.5	Abstract syntax for the Model module.	210
9.6	Abstract syntax for the Traceability module.	210
9.7	Abstract syntax for the Changes module.	212
9.8	Abstract syntax for the MYNK module.	214

LIST OF FIGURES

9.9	CMS-ML and CMS-IL models after applying a Sync Rule.	223
10.1	CMS-ML WebSite Template for the WebC-Docs web application: Macro Structure view.	236
10.2	CMS-ML WebSite Template for WebC-Docs: Micro Structure view.	237
10.3	CMS-ML WebSite Template for WebC-Docs: Roles view.	237
10.4	CMS-ML WebSite Template for WebC-Docs: Permissions view.	238
10.5	CMS-ML Toolkit for WebC-Docs: Roles view.	239
10.6	CMS-ML Toolkit for WebC-Docs: Tasks view.	239
10.7	CMS-ML Toolkit for WebC-Docs: Domain view.	240
10.8	CMS-ML Toolkit for WebC-Docs: States view.	240
10.9	CMS-ML Toolkit for WebC-Docs: WebComponents view.	241
10.10	CMS-ML Toolkit for WebC-Docs: Side Effects view.	242
10.11	CMS-ML Toolkit for WebC-Docs: Interaction Access view.	242
10.12	CMS-ML Toolkit for WebC-Docs: Interaction Triggers view.	243
11.1	Using a CMS-IL model to deploy a web application in a variety of CMS systems.	268
11.2	Proposed development approach, extended with additional languages.	269
A.1	Examples of WebML models.	298
A.2	UWE metamodel's structure.	302
A.3	Examples of UWE models.	302
A.4	Examples of XIS2 models.	305
A.5	The XIS2 views and design approaches.	306
A.6	Examples of models in the OutSystems Agile Platform.	308
A.7	Examples of Microsoft Sketchflow models.	311

List of Tables

4.1	Relevant characteristics of the analyzed MDE approaches and languages. .	76
5.1	Relevant characteristics of the analyzed CMS systems.	86
10.1	Comparison between CMS-ML, CMS-IL, and other web modeling languages.	231

Listings

2.1	Example of a scanner specification.	43
2.2	Example of a parser specification.	44
3.1	Example of a QVTRelation transformation.	51
3.2	Example of a QVTOperational transformation.	53
3.3	Example of a QVTCore transformation.	56
3.4	Example of MOFM2T templates, with and without composition.	57
3.5	Example of a MOFM2T template that redirects output to different streams.	57
3.6	Example of a MOFM2T macro.	58
3.7	Example of an ATL transformation.	59
3.8	Example of a SQL script.	62
3.9	Example of a SQL trigger.	62
3.10	Example of a Ruby on Rails Migration.	65
3.11	Example of an XSLT transformation.	70
8.1	Concrete syntax for the WebSite Template's Structure view.	156
8.2	Concrete syntax for the WebSite Template's Roles view.	157
8.3	Concrete syntax for the WebSite Template's Permissions view.	158
8.4	Concrete syntax for the WebSite Template's Users view.	159
8.5	Concrete syntax for the WebSite Template's Languages view.	161
8.6	Concrete syntax for the WebSite Template's Artifacts view.	163
8.7	Concrete syntax for the WebSite Template's Contents view.	167
8.8	Concrete syntax for the WebSite Template's Visual Themes view.	170
8.9	Concrete syntax for the Toolkit's Roles view.	173
8.10	Concrete syntax for the Toolkit's Code view.	177
8.11	Concrete syntax for the Toolkit's Events view.	182
8.12	Concrete syntax for the Toolkit's Variability view.	184
8.13	Concrete syntax for the Toolkit's Domain view.	188
8.14	Concrete syntax for the Toolkit's WebComponents view.	195
8.15	Concrete syntax for Annotations.	197
8.16	Defining CMS-IL aliases.	198

8.17	Concrete syntax for CMS-IL Reminders.	200
8.18	Concrete syntax for Toolkit Import elements.	201
8.19	Concrete syntax for defining WebSite Template elements as instances of Toolkit elements.	201
8.20	Concrete syntax for using or refining elements from another Toolkit.	202
9.1	Concrete syntax for MYNK.	217
9.2	Example of Simple Sync Rule.	221
10.1	CMS-IL WebSite Template of WebC-Docs.	244
10.2	CMS-IL Toolkit for the WebC-Docs web application.	245
10.3	Synchronization Spec: CMS-ML and CMS-IL.	250
10.4	Synchronization Spec: CMS-ML and UML.	255

Relevant Acronyms

CASE	Computer-Aided Software Engineering
CMS	Content Management System
CMS-IL	CMS Intermediate Language
CMS-ML	CMS Modeling Language
CRUD	Create, Read, Update, and Delete
CSS	Cascading Style Sheet
DBMS	Database Management System
DSL	Domain-Specific Language
DSM	Domain-Specific Modeling
HTML	HyperText Markup Language
IEEE	Institute of Electrical and Electronics Engineers
INESC-ID	Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento em Lisboa
MDE	Model-Driven Engineering
MOF	Meta Object Facility
MOFM2T	MOF Model To Text Transformation Language
MoMM	Metamodel for Multiple Metalevels
MYNK	Model sYNchronization frameworK
OCL	Object Constraint Language

RELEVANT ACRONYMS

OMG	Object Management Group
OOP	Object-Oriented Programming
UI	User Interface
UML	Unified Modeling Language
QVT	Query/View/Transformation
RCS	Revision Control System
ReMMM	Revised Metamodel for Multiple Metalevels
RSS	Really Simple Syndication
SQL	Standard Query Language
URL	Universal Resource Locator
WYSIWYG	What You See Is What You Get
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSL	eXtensible Stylesheet Language
XSLT	XSL Transformations

Chapter 1

Introduction

I thought I'd begin by reading a poem by Shakespeare, but then I thought, why should I? He never reads any of mine.

SPIKE MILLIGAN

With the worldwide expansion of the Internet and the World Wide Web in the last years, we have witnessed a booming rise in popularity of *web applications*. Although some literature defines a web application as an executable program that makes extensive usage of web concepts and technology (such as *request* or *HTTP [HyperText Transfer Protocol]¹ connection*), in this dissertation we opt for a similar definition that is gaining popularity: a **web application** consists of a program that makes use of web-based technologies and is accessed (and used) via a web browser. A particular segment of technologies for web applications, *Content Management Systems* (CMS), have recently gained particular relevance, as they facilitate the distribution of wide varieties of content by non-technical users [SS 08_b, SS 09_a]. Although some of these CMS systems also provide support for the development of more complex web applications to be based on them, that development is still done by using traditional (and error-prone) source code-oriented techniques.

On the other hand, just as software development paradigms changed over the last decades, the current development paradigm is changing from current third-generation programming languages to *Model-Driven Engineering (MDE)* [Sch 06, SV 05_b]. This increasingly-popular paradigm, which aims to achieve a higher level of abstraction than what is common ground today, advocates the usage of models as the most important artifacts in the software development process. Other artifacts, such as source code and

¹“Hypertext Transfer Protocol – HTTP/1.1”, <<http://www.w3.org/Protocols/rfc2616/rfc2616.html>> (accessed on March 15th, 2012)

documentation, can be automatically produced from those models by means of model transformation mechanisms.

The web application development processes that are available today (CMS-based or otherwise) still present problems, namely their inadequacy to deal with the different perspectives of the system’s various stakeholders: it is typically the developer’s responsibility to consolidate those perspectives into a single coherent system. This, in turn, leads to the need for further changes or refinements to the system, because of the unlikeliness that the first iteration of the developed system will meet the expectations of all its stakeholders. Additionally, most current MDE approaches for web application development still consider source code, not models, as the “ultimate” artifact to be deployed or compiled; this is because model design is typically seen as an intermediate step in a process meant to obtain source code, which will be customized at a later stage in the development process.

This dissertation presents our proposal for an MDE approach that addresses the development of CMS-based web applications. Unlike other development approaches – which typically use a single language, such as C# [NEG⁺ 10], Java [Sch 11], or PHP² [LTM 06], aimed at an audience of technical developers –, this approach focuses on addressing the needs of non-technical stakeholders as well as developers. To do so, it proposes (1) a set of modeling languages at different levels of abstraction, and (2) a model synchronization mechanism to ensure consistency between the models for the different stakeholders perspectives of the system.

The remainder of this chapter introduces the research work that was performed in the context of this dissertation. It begins by presenting the background and motivation that led to this research. Afterward, it defines the central point of this dissertation, the *thesis statement*. After this, it identifies the research goals to achieve. Finally, the structure of this document is presented.

1.1 Background

The Internet has become a powerful platform for the deployment of a variety of artifacts and systems, and has also changed the way in which applications are developed. This development shift – from desktop-based rich applications, to web-based applications accessible by regular web browsers – has led to the appearance of a myriad of *web-oriented frameworks and libraries*, which allow developers to harness the power of Internet-based technologies in order to accomplish objectives of various kinds. The main difference between a *framework* (also typically known as *platform*, in the context of web applications) and a *library* is that the latter is just code that the developer can (re)use in the application,

²<http://www.php.net> (accessed on March 15th, 2012)

while a framework typically provides (1) a number of hooks to which a developer can (or must) provide functionality, and (2) generic functionality that the developer can choose to override or customize. Examples of well-known web-oriented frameworks include Microsoft's ASP.NET³ [MF 10], JavaEE⁴ and Java Server Pages [RLJ⁺ 03] (JSP)⁵, or Ruby on Rails (RoR)⁶.

An increasingly popular result of this shift are the many web-oriented CMS (Content Management System) [SATE 03, Boi 01] and ECM (Enterprise Content Management) [Roc 02, Jen 05] systems that are currently available, which have the primary objective of facilitating the management and publication of digital contents on an intranet, or even on the Internet.

Due to the different objectives of each framework (which, in turn, influences the nature of the hooks and functionality that it provides), they are often accompanied by approaches to the development of simple web applications based on that framework. However, the Internet has changed *where* we deploy applications, but not *how* we develop those applications, as their development is still done in a manual fashion. On the other hand, the emerging MDE (Model-Driven Engineering) paradigm [Sch 06, SV 05_b] aims to remove the importance that current software development processes still place on source code, and focus instead on *models* and concepts from the problem- and target-domains.

Model-driven approaches. These approaches are becoming increasingly popular, due to the emergence of the Model-Driven Engineering (MDE) paradigm. As was previously mentioned, MDE advocates the use of *models* as the main artifacts in the software development process, while artifacts such as documentation and source code can be produced from those models by using automated model transformations. This orientation toward models allows developers to focus on relevant concepts (instead of focusing on source code and implementation details that, from the end user's perspective, can be considered irrelevant), which in turn enables a more efficient management of the ever-growing complexity of software. Besides leaving most of the repetitive and error-prone tasks to model transformations, MDE approaches also present additional advantages [Sch 06], such as: (1) relieving developers from issues like underlying platform complexity or inability of programming languages to express domain concepts; or (2) targeting multiple deployment platforms without requiring different code bases.

Although some examples of MDE approaches exist – the most famous of which is perhaps the Object Management Group's Model-Driven Architecture (MDA) [KWB 03] –,

³<http://www.asp.net> (accessed on March 15th, 2012)

⁴<http://www.oracle.com/javaee> (accessed on March 15th, 2012)

⁵<http://www.oracle.com/technetwork/java/javaee/jsp> (accessed on March 15th, 2012)

⁶<http://rubyonrails.org> (accessed on March 15th, 2012)

most developers still use MDE in the context of Domain-Specific Modeling (DSM) [KT 08], in which (1) a visual language is defined and oriented toward a specific (and typically simple) domain, and (2) the models developed will be used to automatically generate the corresponding source code. Thus, the software development process itself is still source code-oriented, although some parts of it are simplified by the DSLs.

CMS-based approaches. Although it is usually not their main goal, Content Management Systems (CMS) can not only be used for creating websites, but also as support platforms for the deployment of specialized web applications. Before proceeding, it is important to establish the difference between the concepts of *web application* and *website*, in the context of this dissertation: a web application is a program that uses web technologies (as was mentioned at the beginning of this chapter), while a website consists of a *concrete usage* of a web application. In other words, this dissertation considers a website to be a *specific deployment* of a web application (e.g., the Google search website⁷ is a deployment of the web application that uses the search engine software developed by Google’s creators).

These CMS systems (which are themselves web applications) can effectively support the dynamic management of websites and their contents [SS 08_b], and typically present aspects such as (1) extensibility and modularity, (2) independence between content and presentation, (3) support for several types of contents, (4) support for access management and user control, or (5) support for workflow definition and execution. On the other hand, Enterprise Content Management (ECM) systems are typically web applications oriented toward using Internet-based technologies and workflows to capture, manage, store, preserve, and deliver contents and documents in the context of organizational processes [Roc 02].

Nevertheless, these two areas are not disjoint, and it is not unusual to find a regular CMS system acting as a repository for an organization’s documents and contents, albeit at a somewhat primitive level (e.g., with no checking for duplicate information, no logical grouping of documents, and no support for providing metadata for each document). Furthermore, a CMS platform can be used to perform the same tasks as an ECM platform, as long as the CMS provides the developer with adequate functionality and hooks [SS 08_b]. In fact, some CMS systems are evolving to become true frameworks themselves, by endowing developers with a set of hooks and features that are adequate for the creation of web applications with purposes other than simple content management. In addition to these features, CMS systems also provide a set of high-level concepts – such as user, role, or module – that facilitate the development of complex web applications.

⁷ Google, <<http://www.google.com>> (accessed on May 24th, 2012)

However, development and deployment of CMS-based web applications is still typically done in a traditional manner. This development involves (1) manually writing source code that uses an API (Application Programming Interface) made available by the CMS, in a programming language supported by the CMS's underlying web server technology (such as C#, Java, or PHP), (2) compiling that source code, and (3) deploying the final output artifacts to the CMS, either by using a mechanism built into the CMS or by manually copying those artifacts to the corresponding web server.

1.2 Research Context

The research work presented in this dissertation was supervised by Prof. Dr. Alberto Manuel Rodrigues da Silva, and was developed in the context of (1) the Information Systems Group (GSI) of INESC-ID (Instituto de Engenharia de Sistemas e Computadores Investigação e Desenvolvimento em Lisboa), and (2) the company SIQuant.

The Information Systems Group is a research group of INESC-ID⁸, a nonprofit organization dedicated to research and development. The GSI is dedicated to a variety of research topics, some of which are of relevance for this work: (1) social software engineering (also known as Computer-Supported Cooperative Work, CSCW); (2) CMS-based systems and applications; (3) model-driven engineering; and (4) methodologies, processes, and tools to design and build information systems.

On the other hand, SIQuant⁹ is a company that specializes in information systems and geographical information. SIQuant's relevance for this dissertation lies in its experience in the design and development of CMS-based web applications, namely by using the WebComfort CMS framework¹⁰ [SS 08_b] (which is analyzed in Appendix A). In fact, the WebC-Docs document management system (the case study used to validate our proposal, depicted in Chapter 10) was initially implemented in the context of WebComfort [SS 09_b, SS 11_a].

1.3 Research Problems

The main research problems that this dissertation (and the proposed solution) aim to address can be summarized in the following list:

P1: Most current model-driven development approaches are based on a *single* modeling language;

⁸<http://www.inesc-id.pt> (accessed on March 15th, 2012)

⁹<http://www.siquant.pt> (accessed on March 15th, 2012)

¹⁰<http://www.siquant.pt/WebComfort> (accessed on March 15th, 2012)

P2: Current modeling languages try to either (1) abstract the designer from low-level implementation details, or (2) sacrifice the aforementioned abstraction, and be expressive enough to define a fully-functional application; and

P3: The primary artifact that results from current model-driven web application development approaches is still source code, instead of models.

A more in-depth explanation of these problems is provided in the following paragraphs.

P1: MDE approaches based on a single language. The first research problem is that current model-driven software development approaches (web application-oriented or otherwise) are centered around a single modeling language: either OMG’s standard UML [OMG 11_e] or another approach-specific language. This would not present a problem if the developer was the only participant in the software development process, as the modeling language would simply be oriented toward facilitating the developer’s tasks.

However, the software development process involves other stakeholders (namely the application’s end-users), each of which has a particular perspective regarding what the application should do. Thus, what typically happens is that those other stakeholders provide the developer with a set of requirements, such as intended features or business rules, and it becomes the developer’s responsibility to map those requirements to the implementation language used (i.e., to manually establish a correspondence between the problem-domain and the solution-domain). This, in turn, is often an error-prone task, as it is highly dependent on the developer’s interpretation (and, occasionally, *mis*interpretation) of those requirements.

The scenario mentioned in the previous paragraph is just a simplified view of the typical development process, of course. In practice, this process actually involves not only developers but also other specialized kinds of stakeholder (technical or otherwise), such as the System Architect or the Project Manager. However, in this dissertation, and to keep the problem description as simple as possible, we will consider that the process’ participants can be divided into technical and non-technical stakeholders, and that the term “developer” means “someone who participates in the low-level implementation of the application by using a programming language”, unless explicitly stated otherwise (these kinds of stakeholder will be further analyzed in Chapter 6).

P2: Modeling language is either too simple or too complex. The second research problem is that most modeling languages which try to abstract the designer from low-level implementation details – such as WebML or XIS2, which are analyzed in Appendix A – are typically not expressive enough to define a fully-functional application, and vice versa. Although this is understandable (because a model is supposed to be an abstraction over

reality), this eventually leads to developers manually editing source code, a practice which MDE is also meant to avoid. On the other hand, approaches that provide a very expressive modeling language – such as the OutSystems Agile Platform language, also analyzed in Appendix A – are likely to not provide a real abstraction over implementation details (which is one of the cornerstones of MDE), but only a different concrete syntax for the same programming language concepts.

This problem is related to the previous one, because those languages are typically the result of trying to concentrate too many details of various abstraction levels – such as high-level models and implementation details – in a single language. This compromise, in turn, makes the language either very complex but able to model most intended applications, or relatively simple but unable to model real-world applications. We have also observed this compromise in our own experience with the development of the XIS2 language [SSSM 07].

P3: Source code is still the main artifact. The third research problem is that the primary artifact resulting from current development approaches is still source code specified in a certain programming language (because that is what is compiled), instead of *models*. Although programming languages could themselves be considered as modeling languages (this is further discussed in Chapter 2), they are nevertheless still oriented toward *how the computer should solve a problem*, instead of *what is the problem to solve*. Furthermore, even though most approaches consider that developers can directly edit the generated low-level artifacts, one of the main objectives of MDE is precisely to avoid this kind of editing, as it would lead to developers having to deal with low-level issues anyway. Furthermore, if the source code generation process was run again, then it is likely that the manual changes made to the source code would be lost.

Again, this problem is related to the previous one, because the lack of expressiveness in most modeling languages is likely derived from the fact that their creators already assume that modeling will only take place as an *intermediate* activity in the application development process, instead of being an activity to produce the main artifact that will be deployed in a production environment.

This is another issue where OutSystems’ approach distinguishes itself. Although it does consider source code generation and compilation, those steps are done “behind the scenes”, and the developer cannot intervene in the process. This, in turn, makes it mandatory that the modeling language be sufficiently expressive to accommodate the low-level needs of developers regarding the development of most kinds of web applications.

1.4 Research Questions and Objective

Considering the analysis of the State of the Art (presented in Chapters 2–5 of this dissertation) and on the research problems mentioned in Section 1.3, we have defined a set of research questions that we intend to address:

- How can an MDE approach effectively address the perspective that *each* kind of stakeholder will have regarding the web application to be developed?
- How can a modeling language (or an MDE approach) address the *various abstraction levels* that are necessary to specify a web application in practice (e.g., the website’s layout configuration, and the components that can be used in a website)?
 - Is it really necessary to have a single language adopt a compromise between the support for low-level details and the ease of its learning and use?
- Would an approach that supports multiple abstraction levels be more productive than a traditional (i.e., manual) development approach?
- Is it feasible to have *models* as the project’s *de facto* main artifact, instead of source code (the more traditional artifact)?
- Considering the aforementioned issues, is it possible for an MDE approach to adequately support the development of CMS-based web applications?

Derived from these questions, the objective of this research work is to *improve the manner in which CMS-based web applications are developed*. This improvement should be achieved by: (1) supporting the perspectives of *various kinds of stakeholders*; and (2) making *stakeholder-produced models* the main artifacts of the development process (and avoiding the traditional need to later generate source code that will then be manipulated by developers). This research objective, in turn, leads to the Thesis Statement that is presented in the next section.

1.5 Thesis Statement

The thesis statement that this research work aims to prove is the following:

An MDE approach for the development of CMS-based web applications does not need to be constrained by the usage of a single modeling language. Instead, it is possible for such an approach to provide a set of languages, each of which addressing the needs of a specific kind of stakeholder. Furthermore, the usage of multiple languages does not require that corresponding models be continuously updated in a manual fashion.

In other words, we intend to prove that it is possible for an MDE approach (aimed at developing CMS-based web applications, in the context of this dissertation) to support the

usage of *multiple languages* – one for each of its kinds of stakeholder – in such a manner that it can effectively be used in the development of relatively complex web applications. Such an approach would contrast with typical MDE approaches that provide a single language (oriented toward either an abstract high-level view of the application, or a low-level view of its implementation). Of course, in addition to the modeling languages themselves, the MDE approach must also address the automatic synchronization between models of different languages; otherwise, it would become necessary to perform such synchronization manually, which would easily lead to human-generated errors.

The rationale behind this thesis statement is twofold: (1) each stakeholder has a specific perspective (or view) of what she intends the web application to be; and (2) the web application should be an explicit result from the gathering of *all* those views, as opposed to it being what the *developer* understands the stakeholders' views to be.

1.6 Research Method

This research project began with our intent to: (1) gain further expertise on the domain of model-driven engineering; and possibly (2) change and improve the way in which CMS-based web applications are developed, by applying adequate MDE-oriented techniques to this domain.

Because of the nature of this project, and considering the aforementioned intent, we opted to perform our research work by using the method of *Action Research*¹¹ [Lew 46, SE 78]. **Action Research** is a research method that focuses on making questions, gathering data, and obtaining the corresponding results, in a continuous and cyclic manner. This cycle – depicted in Figure 1.1 – is composed of the following phases [SE 78], which are performed in the specified order:

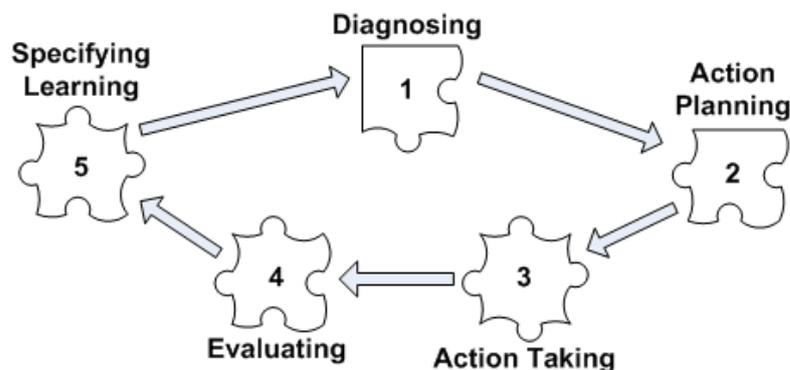


Figure 1.1: The phase cycle of the Action Research method (adapted from [SE 78]).

¹¹Greg Waddell, “What is Action Research?”, <<http://www.slideshare.net/Gregwad/action-research>> (accessed on May 1st, 2012)

- **Diagnosing:** This phase consists of explicitly identifying and defining the *problem* at hand;
- **Action Planning:** In this phase, the researcher selects a potential course of action that is believed to adequately solve the identified problem, and *designs* a corresponding *solution*;
- **Action Taking:** This phase sees the researcher, in collaboration with participants (e.g., clients), applying the solution in an *experiment*;
- **Evaluating:** In this phase, the researcher gathers the data that results from performing the experiment, and *analyzes* it; and
- **Specifying Learning:** It is in this phase that the researcher identifies the *lessons* learned during the experiment. This can lead to one of the following outcomes: (1) the research is considered to be completed; or (2) further research is required in order to solve the problem. In the case of the latter, the cycle is repeated, but now with the additional knowledge that was gained during this phase.

We have also used the following research techniques while performing this work:

- (1) analysis of the State of the Art (sometimes referred to as *disciplined literature review* or *systematic review*¹²), to identify features that were relevant to our research; and
- (2) engineering-based techniques, namely the technique of designing and constructing a prototype of a system to test a hypothesis.

Finally, the obtained research results were validated by performing: (1) a set of case studies; and (2) a feature-based comparison with the related State of the Art. This work was also validated, whenever possible, by the submission of papers to international and national scientific conferences and journals on model-driven software engineering and other related topics.

1.7 Contributions

The research work presented in this dissertation has resulted in a number of scientific contributions that should be mentioned, and which are listed below:

- C1:** A model-driven development approach for CMS-based web applications that supports the usage of multiple modeling languages;
- C2:** ReMMM, a simple metamodel for the definition of modeling languages;
- C3:** The CMS-ML and CMS-IL modeling languages, which also have some features that are noteworthy contributions to the State of the Art regarding modeling language design;

¹²<http://www.rogerclarke.com/SOS/DLR.html> (accessed on May 1st, 2012)

C4: A small set of guidelines that can be used to endow the aforementioned approach with additional modeling languages; and

C5: The MYNK model synchronization language.

These contributions are presented in greater detail in the remainder of this dissertation. Nevertheless, the following paragraphs provide a brief description of these contributions.

C1: A multi-language approach for the development of CMS-based web applications. The main contribution of this dissertation is the definition of a model-driven approach to develop web applications based on CMS platforms. This approach is meant to address the needs of multiple kinds of stakeholders in a collaborative manner, and avoid the traditional workflow of (1) non-technical stakeholders specifying requirements, (2) handing them over to developers, and afterward (3) restarting the cycle when problems are detected in the implementation. This way, a non-technical stakeholder can intervene in the web application’s development at any time by adjusting its models, and the developer can also immediately obtain the necessary feedback that typically would only be available at the end of the aforementioned development cycle.

We expect that this contribution brings some added-value to the software development process, by providing the following advantages:

- *Less errors*, because each stakeholder can express their own views of the desired web application in a language to which she can relate, without the developer having to manually make a complex translation between the stakeholder’s problem-domain and the developer’s solution-domain; and
- *Less development time*, because changes to any such view should be reflected over other related views in an automatic manner, by means of a synchronization mechanism (contribution C5, explained further down this section).

C2: The ReMMM metamodel. ReMMM is a simple metamodel that provides designers with the basic concepts for developing new modeling languages. The most significant difference between ReMMM and other (meta)metamodels (such as MOF [OMG 11_b]) is that ReMMM is meant to enable the definition, representation, and manipulation of not only models but also of *metamodels*.

C3: The CMS-ML and CMS-IL modeling languages. The CMS-ML modeling language allows *non-technical stakeholders* (i.e., not involved in the system’s implementation) to participate in the design and development of the CMS-based web application, by defining a set of concepts that is adequate for this audience. On the other hand, CMS-IL is a relatively low-level language for *developers* to perform their development tasks.

The manner in which CMS-ML and CMS-IL are defined, in conjunction with ReMMM, can itself be considered as another scientific contribution, because it makes these languages (1) not only extensible, but also (2) compliant with the *strict metamodeling* principle (which is explained in Chapter 2). Future ReMMM-based modeling languages can benefit from this solution’s rationale, to easily add further extensibility mechanisms and/or metalevels.

C4: Guidelines for modeling language design. Still regarding the proposed multi-language development approach, this dissertation provides not only the ReMMM meta-model to define new modeling languages, but also a small set of guidelines that can help the language designer in determining the scope and extensibility requirements for a new modeling language. The CMS-ML and CMS-IL languages were both obtained by following these guidelines which, although simple in nature, were essential to (1) quickly ensure that each language would have the necessary elements, and (2) avoid gold plating the languages at the same time.

C5: The MYNK model synchronization language. The MYNK model synchronization language is the cornerstone of the proposed development approach; without MYNK, this approach would just be impractical and error-prone. This language is based on the usage of model changes (instead of the model itself) as the driver for the transformation and evolution of models, which presents some advantages over other existing model transformation techniques (presented in Chapter 3). Furthermore, MYNK only requires the usage of the ReMMM metamodel, and is not in any way dependent on CMS-IL and CMS-ML. Thus, it can be used with other ReMMM-based modeling languages.

1.8 Publications

In the context of this doctoral research work, the following articles have been published in international journals and books, with peer-reviewing:

1. **Saraiva, João** and Silva, Alberto. Design Issues for an Extensible CMS-Based Document Management System. In Ana Fred, Jan L. G. Dietz, Kecheng Liu, and Joaquim Filipe, editors, *Knowledge Discovery, Knowledge Engineering and Knowledge Management: First International Joint Conference, IC3K 2009, Funchal, Madeira, Portugal, October 6–8, 2009, Revised Selected Papers*, volume 128 of *Communications in Computer and Information Science*. Springer Berlin/Heidelberg, 2011. doi:10.1007/978-3-642-19032-2_24. ISBN 978-3642190322.
2. **Saraiva, João** and Silva, Alberto. A Reference Model for the Analysis and Comparison of MDE Approaches for Web-Application Development. *Journal of Software*

- Engineering and Applications*, 3(5):419–425, May 2010. doi:10.4236/jsea.2010.35047. ISSN 1945-3116.
3. **Saraiva, João** and Silva, Alberto. Evaluation of MDE Tools from a Metamodeling Perspective. In Keng Siau and John Erickson, editors, *Principle Advancements in Database Management Technologies: New Applications and Frameworks*, pages 105–131. Information Science Publishing, 2010. doi:10.4018/978-1-60566-904-5.ch005. ISBN 978-1605669045.
 4. **Saraiva, João** and Silva, Alberto. Evaluation of MDE Tools from a Metamodeling Perspective. *Journal of Database Management*, 19(4):21–46, October/December 2008. doi:10.4018/jdm.2008100102. ISSN 1063-8016.
 5. Silva, Alberto, **Saraiva, João**, Ferreira, David, Silva, Rui, and Videira, Carlos. Integration of RE and MDE Paradigms: The ProjectIT Approach and Tools. *IET Software Journal – Special Issue: On the Interplay of .NET and Contemporary Development Techniques*, 1(6):294–314, December 2007. doi:10.1049/iet-sen:20070012. ISSN 1751-8806.

Furthermore, the following articles have also been published in international conferences with peer-reviewing:

1. **Saraiva, João** and Silva, Alberto. WebC-Docs: A CMS-based Document Management System. In Kecheng Liu, editor, *Proceedings of the International Conference on Knowledge Management and Information Sharing (KMIS 2009)*, pages 21–28. INSTICC Press, October 2009. ISBN 978-9896740139.
2. **Saraiva, João** and Silva, Alberto. CMS-based Web-Application Development Using Model-Driven Languages. In Kenneth Boness, João M. Fernandes, Jon G. Hall, Ricardo Jorge Machado, and Roy Oberhauser, editors, *Proceedings of the Fourth International Conference on Software Engineering Advances (ICSEA 2009)*, pages 21–26. IEEE Computer Society, September 2009. doi:10.1109/ICSEA.2009.12. ISBN 978-0769537771.
3. **Saraiva, João** and Silva, Alberto. The WebComfort Framework: an Extensible Platform for the Development of Web Applications. In *Proceedings of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2008)*, pages 19–26. IEEE Computer Society, September 2008. doi:10.1109/SEAA.2008.12. ISBN 978-0769532769.
4. Baptista, Frederico, **Saraiva, João**, and Silva, Alberto. eCT – The B2C e-Commerce Toolkit for WebComfort Platform. In *Proceedings of the International Joint Conference on e-Business and Telecommunications (ICETE 2008)*. INSTICC, July 2008.

5. **Saraiva, João** and Silva, Alberto. The ProjectIT-Studio/UMLModeler: A tool for the design of and transformation of UML models. In *Actas de la Conferencia Ibérica de Sistemas y Tecnologías de la Información (CISTI 2008)*, pages 687–698. June 2008. ISBN 978-8461244744.
6. Silva, Alberto, **Saraiva, João**, Silva, Rui, and Martins, Carlos. XIS – UML Profile for eXtreme Modeling Interactive Systems. In João M. Fernandes, Ricardo Jorge Machado, Ridha Khédri, and Siobhán Clarke, editors, *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007)*, pages 55–66. IEEE Computer Society, Los Alamitos, CA, USA, March 2007. doi:10.1109/MOMPES.2007.19. ISBN 0769527698.

The following article has been published in a national conference with peer-reviewing:

1. **Saraiva, João** and Silva, Alberto. Web-Application Modeling With the CMS-ML Language. In Luís S. Barbosa and Miguel P. Correia, editors, *Actas do II Simpósio de Informática (INForum 2010)*, pages 461–472. September 2010. ISBN 978-9899686304.

The following article has also been produced in the context of a national doctoral symposium:

1. **Saraiva, João** and Silva, Alberto. Development of CMS-based Web-Applications Using a Model-Driven Approach. In Kenneth Boness, João M. Fernandes, Jon G. Hall, Ricardo Jorge Machado, and Roy Oberhauser, editors, *Proceedings of the Simpósio para Estudantes de Doutoramento em Engenharia de Software (SEDES 2009, co-located with ICSEA 2009)*, pages 500–505. IEEE Computer Society, September 2009. doi:10.1109/ICSEA.2009.79. ISBN 978-0769537771.

Finally, the following technical reports have been produced and are publicly available on the Internet:

- **Saraiva, João** and Silva, Alberto. MYNK User’s Guide. Technical Report, Instituto Superior Técnico, November 2011.¹³
- **Saraiva, João** and Silva, Alberto. CMS-IL User’s Guide. Technical Report, Instituto Superior Técnico, September 2011.¹⁴
- **Saraiva, João** and Silva, Alberto. CMS-ML User’s Guide. Technical Report, Instituto Superior Técnico, April 2010.¹⁵

¹³http://isg.inesc-id.pt/Modules/WebC_Docs/ViewDocumentDetails.aspx?DocumentId=96 (accessed on March 15th, 2012)

¹⁴http://isg.inesc-id.pt/Modules/WebC_Docs/ViewDocumentDetails.aspx?DocumentId=95 (accessed on March 15th, 2012)

¹⁵http://isg.inesc-id.pt/Modules/WebC_Docs/ViewDocumentDetails.aspx?DocumentId=93 (accessed on March 15th, 2012)

- Silva, Alberto, **Saraiva, João**, Ferreira, David, Videira, Carlos (INESC-ID), and Nunes, Nuno Jardim (Univ. da Madeira). POSC/EIA/57642/2004 – Técnicas de desenvolvimento de software baseadas na engenharia de requisitos e em modelos no âmbito do programa de investigação ProjectIT. Final Project Report, POSC-FCT, November 2007.

1.9 Dissertation Structure

This dissertation is organized in eleven chapters and two appendixes, laid out according to the structure outlined below.

Introduction. This part, which consists of the current chapter, defines the research problems, thesis statement, and the research approach that are the basis for the work presented in this dissertation.

State of the Art. Chapters 2–5 present the State of the Art for the topics that were relevant for our work, namely MDE languages and approaches, and CMS systems. More specifically, each chapter addresses the following subjects:

- Chapter 2 (Modeling Language Design) presents a small set of generic MDE-related languages and software development approaches, namely those defined by the Object Management Group (OMG). This chapter also defines the concepts of *model* and *metamodel*, as well as the activity of *metamodeling* and a range of aspects that can influence it. Finally, this chapter also presents some similarities that can be observed between metamodeling and activities that typically take place during the definition of a programming language.
- Chapter 3 (Model Transformations) focuses on the topic of model transformations, namely on the OMG’s MDE-related languages that address model-to-model and model-to-text transformations. This chapter also presents a set of transformation techniques that are frequently used in traditional software development; not only do these techniques address a range of issues that can also be found in the aforementioned model transformation languages, but they also have similar rationales, which may lead to the possibility of adapting these techniques for usage in MDE.
- Chapter 4 (Model-Driven Approaches for Web Application Development) provides a comparison of a set of web application-oriented modeling approaches and languages, and identifies some language aspects that are addressed by some (or all) of those languages. An in-depth analysis of these languages is also provided in Appendix A.

- Chapter 5 (Content Management Systems) compares a set of CMS systems, and identifies some aspects that CMS-oriented languages should at least take into consideration. An analysis of each of these CMS systems is provided in Appendix B.

Proposed Solution. Chapters 6–9 present the main contributions of this dissertation, namely the solution that was developed throughout our research work. Each of these chapters presents a different subject:

- Chapter 6 (The CMS Multi-Language Model-Driven Approach) provides an overview of the proposed MDE approach, based on the usage of multiple modeling languages, for the development of CMS-based web applications. It also presents ReMMM, the metamodel used for the definition of the CMS-ML and CMS-IL languages (presented in the subsequent chapters).
- Chapter 7 presents CMS-ML (CMS Modeling Language), a graphical language that addresses the modeling of CMS-based web applications in a high-level and platform-independent manner.
- Chapter 8 presents CMS-IL (CMS Intermediate Language), a CMS-oriented textual language that provides a low level of abstraction over computation concepts (much like traditional programming languages), while still remaining CMS-oriented and platform-independent.
- Chapter 9 presents MYNK (Model sYNchronization framework), a textual model synchronization language that is used to synchronize changes between ReMMM-based models (namely CMS-ML and CMS-IL models).

Validation. This part, comprised only of Chapter 10, presents the validation of the solution that is proposed in this dissertation. In particular, this chapter discusses the models that were defined for the WebC-Docs case study, which was used as a guide for our validation efforts.

Conclusion. This dissertation is concluded in Chapter 11, which provides a final overview of the research work that was performed, as well as a retrospective analysis of the results obtained. In addition to these final notes, this chapter also points out some topics and issues that can lead to possible future work.

Appendices. Appendices A and B provide additional information that was not included in the main body of this dissertation:

- Appendix A (State of the Art: Model-Driven Approaches for Web Application Development) presents an in-depth analysis of the web application modeling languages and approaches that were previously compared in Chapter 4.

- Appendix B (State of the Art: CMS Systems) presents a more detailed analysis of the CMS systems (and their corresponding development approaches) that were compared in Chapter 5.

1.10 Adopted Notations and Conventions

In order to establish a clear distinction between plain text and certain relevant terms and designations, this dissertation adopts a typographical convention that is reflected in the following styles:

- The **bold face** font is used to highlight the *definition* of concepts that are important to this dissertation.

Example:

“A **web application** is an executable program that (...)”

- The *italic* font is used to emphasize a relevant name or segment of text. Foreign terms are also represented using an italic font.

Example:

“Is it feasible to have *models* as the project’s *de facto* main artifact (...)”

- The `typewriter` font, when used in its normal size, indicates *language keywords* or *code snippets* (e.g., names of classes, attributes, or methods). On the other hand, a smaller `typewriter` font is used when representing *values* of elements (e.g., the name of a certain class, or the value assumed by a specific attribute).

Example:

“The instances of UML’s `Class` element – namely `Social Network`, `Person`, `Role`, and `Relationship` – are represented (...)”

- Quotation marks (“”) are used with one of the following purposes: (1) to represent a *word*, and not its meaning; (2) to represent a *string*; (3) to mention the title of a literary work (e.g., a user’s manual); or (4) as scare quotes (i.e., indicators that the corresponding text should not be interpreted literally).

Example:

“Several definitions for the term “metamodeling” can be found (...)”

Furthermore, whenever a figure is extracted or adapted from another document, the figure’s caption will explicitly indicate this fact and make a reference to the corresponding

document. All the original figures created for this dissertation (i.e., not extracted or adapted from some external source) have no such reference.

When suitable, figures will consist of diagrams specified using the UML modeling language [OMG 11.e], as it is assumed that readers are familiar with this standard. These diagrams also assume that the default cardinality for the extremities of each association (including aggregations) is 1, unless the diagram explicitly indicates otherwise.

Finally, this dissertation uses some terms that are synonym to other terms also found in literature. The most relevant cases to mention are the following:

- The terms “metalevel” and “metalayer” have the same meaning. Most of the available literature uses the term “metalevel”, and thus we use “metalevel” in this dissertation; nevertheless, the Object Management Group (OMG) uses the term “metalayer” [OMG 11.b]. Furthermore, we only use the term “layer” when referring to technology (and not metamodeling), or when referring to models that have dependencies on elements in other models (e.g., a UI model is a layer that uses elements from a domain layer); and
- The terms “website” and “web application” assume the meaning that was previously provided in this chapter: a web application is a program that uses web-based technologies, and a website is a deployment of a web application. This means that, in this dissertation, a website can be understood as being an instance of a web application. Nevertheless, because these concepts are closely related, the terms “website” and “web application” will sometimes be used in an interchangeable manner, and the reader can often consider these terms as synonyms (unless the text explicitly states otherwise).

Summary

In this chapter, we have provided a brief introduction to the research work that was performed in the context of this PhD dissertation, namely by presenting (1) the research problem that motivated this work, (2) the thesis statement, and (3) the research objectives that the dissertation aims to accomplish.

The following chapters are dedicated to the presentation of the State of the Art that was considered relevant for this research work. In particular, the next chapter is dedicated to presenting the MDE paradigm in greater detail, as well as some important modeling language creation approaches that are of interest regarding the work presented in this dissertation.

Chapter 2

Modeling Language Design

A different language is a different
vision of life.

FEDERICO FELLINI

To improve the manner in which a software program is developed, it is necessary that the software program's stakeholders have the ability to express their domain knowledge, intents, and objectives using suitable languages with which they can relate. Such languages can be graphical (e.g., pictorial languages such as UML) or even textual (e.g., traditional programming languages such as Python, Ruby, C#, Java, or many others that exist today), depending on factors such as the stakeholder's concerns and technical background.

This need for stakeholder-friendly languages, in turn, originates the need for mechanisms to create those languages, to support the definition of new ways to create and specify models representing solutions for problems. Furthermore, the subject of language creation is closely related with the continuous effort by software engineers and researchers to elevate the abstraction level at which they perform their software analysis, design, and development tasks.

In the context of this research, we have studied and analyzed the following language design topics and approaches: (1) Model-Driven Engineering (MDE); (2) Model-Driven Architecture (MDA), the Object Management Group's approach; (3) Domain-Specific Modeling (DSM); (4) metamodeling; and (5) the similarities between the activities of designing modeling languages and programming languages. Although other modeling language design approaches and respective supporting tools exist – such as (1) MIC (Model-Integrated Computing)¹ [SK 97] and its GME tool (Generic Modeling Environment)²,

¹<http://www.isis.vanderbilt.edu/research/MIC> (accessed on March 15th, 2012)

²<http://www.isis.vanderbilt.edu/Projects/gme> (accessed on March 15th, 2012)

(2) Microsoft’s Software Factories³ [GSCK 04], or (3) the Eclipse GMP (Graphical Modeling Project)⁴ – we consider that they share similar underlying concepts with the analyzed approaches.

It should be noted beforehand that the analysis presented is not intended to be exhaustive. This is because some aspects of such an analysis would be out of the scope of this research work. Furthermore, each of these topics has already been the subject of numerous books and articles, and so presenting such an analysis would extend this dissertation’s length significantly, while providing little added-value. Instead, the presented analysis is focused only on aspects that are of particular relevance to the scope of this work, namely the aspects that are identified in Section 2.4.

2.1 Modeling and Model-Driven Engineering

Software systems are reaching such a high degree of complexity that even current third-generation programming languages, like Java or C#, are becoming unable to efficiently support the creation of such systems [Sch 06]. One of the problems with such current languages is that they are still too oriented toward specifying *how the solution should work*, instead of *what the solution should be*. This leads to a need for mechanisms and techniques that allow the developer to abstract over the programming language itself and focus on creating a simple (and, whenever possible, elegant) solution to a certain problem.

A considerable research effort has been made in recent years to raise the abstraction level into Model-Driven Engineering (MDE). Sometimes called Model-Driven Development (MDD), MDE is an emerging paradigm based on the systematic use of models as the cornerstones of the solution specification [Sch 06, MCF 03, SV 05.b]. Unlike previous (source code-based) software development paradigms, models become the most important entities, and artifacts such as source code or documentation can then be obtained from those models in an automated manner, relieving developers from issues such as underlying platform complexity or inability of third-generation languages to express domain concepts.

MDE is not a new idea. In fact, in the 1980s and 1990s, Computer-Aided Software Engineering (CASE) tools were already focused on endowing developers with methods and tools to design software systems using graphical general-purpose language representations. The developer would then be able to perform different tasks over those representations, such as verification or transformations to/from code. However, these CASE tools failed to accomplish their intended purpose, due to issues such as [Sch 06]: (1) poor mapping of general-purpose languages onto the underlying platforms, which made generated code

³<http://www.softwarefactories.com> (accessed on March 15th, 2012)

⁴<http://www.eclipse.org/modeling/gmp> (accessed on March 15th, 2012)

much harder to understand and maintain; (2) inability to scale, because they did not support concurrent development; and (3) source code was still the most important entity in the development process, while models were seen as only being suited for documentation. Nowadays, the stage is more favorable for the MDE paradigm to succeed. This is because of the aforementioned inability of third-generation languages to address the increasing degree of complexity of current software systems, combined with the choices of development platforms currently available (e.g., Java) to which models can be mapped.

There are currently a significant number of MDE-oriented approaches, of which we highlight the Model-Driven Architecture (MDA) and Domain-Specific Modeling (DSM). It is important to note that *MDE does not define any particular approach*; instead, MDE itself is a paradigm that is addressed by such approaches, while being *independent of language or technology* [Fav 04_b, Fav 04_a].

Nevertheless, all MDE-oriented approaches share the same basic concepts. A **model** is an interpretation of a certain domain – a fragment of the real world over which modeling and system development tasks are focused – according to a determined structure of concepts [SV 05_a]. In other words, a model can be viewed as a *reduced representation of a system*, highlighting properties that are of interest according to a specific perspective [Sel 03]. It can also be regarded as a system, which itself provides *answers* about the system that is being analyzed without the need to directly consider the latter [Fav 05].

The structure of concepts that defines a model is provided by a **metamodel**, which is an attempt at describing the world around us for a particular purpose, through the precise definition of the constructs and rules needed for creating models [SS 08_a, SS 10_b]. This means that a metamodel provides a language that can be used to create a model, as Figure 2.1 illustrates; similarly, a metamodel that defines the language in which another metamodel is specified is called a *metametamodel*.

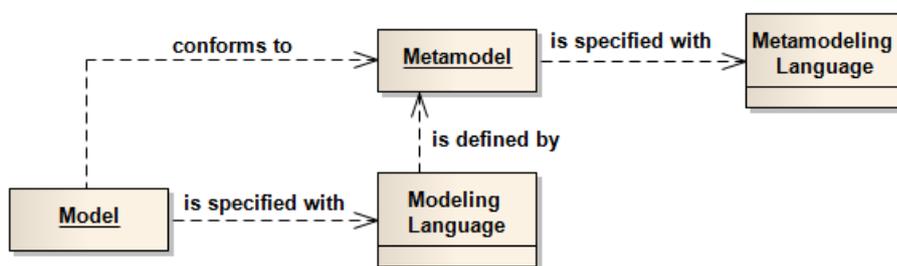


Figure 2.1: The relationship between a model and a metamodel (adapted from [SS 08_a]).

Also noteworthy is the concept of **metalevel** (sometimes called *metalevel*, namely in OMG's specifications). A metamodel is often said to be in a metalevel above the corresponding model's metalevel. Thus, defining a stack of metalevels provides a way to hierarchically structure a set of models in relation to their respective metamodels. The

concept of metalevel is particularly important when defining metamodeling architectures such as OMG's (which is further explained in Subsection 2.2.2).

From a practical perspective, and considering Figure 2.1, it is noteworthy to point out that most modeling tools typically deal with only one logical level (i.e., user model editing and a hardcoded metamodel). This is because creating such a tool is easily done with a typical Object-Oriented Programming (OOP) language [CN 91], by taking advantage of the class-instance relation and making the logical level be supported by the instance level.

Now that these basic modeling concepts have been presented, we are now able to provide an analysis of some relevant MDE approaches and languages, according to the aspects that were identified at the beginning of this chapter.

2.2 The OMG Approach

The Object Management Group (OMG) has defined a well-known approach to address the MDE paradigm. This approach, called Model-Driven Architecture (MDA), is based on a set of OMG standards that make use of techniques for modeling language specification and model transformation. Of particular relevance to this chapter is the fact that some of these standards (namely MOF and UML, presented next) provide mechanisms to define new modeling languages.

2.2.1 Unified Modeling Language (UML)

The Unified Modeling Language (UML) Superstructure [OMG 11_e] (typically called only UML) is a general-purpose modeling language. Originally designed to specify, visualize, and document software systems, it is also used as a basis for a variety of tools, methodologies, and research efforts regarding software development and source code generation.

UML 2.4.1 [OMG 11_e] (the current version as of the writing of this dissertation) is composed of 14 different kinds of diagram (as illustrated in Figure 2.2), divided into two categories: (1) structure diagrams, which deal with the static structure of the objects in the system being modeled; and (2) behavior diagrams, which specify the dynamic aspects and behavior of the system's objects. UML defines the following structure diagrams: (1) class diagram, (2) component diagram, (3) composite structure diagram, (4) deployment diagram, (5) object diagram, (6) package diagram, and (7) profile diagram. It also defines the following behavior diagrams: (1) activity diagram, (2) communication diagram, (3) interaction overview diagram, (4) sequence diagram, (5) state machine diagram, (6) timing diagram, and (7) use case diagram.

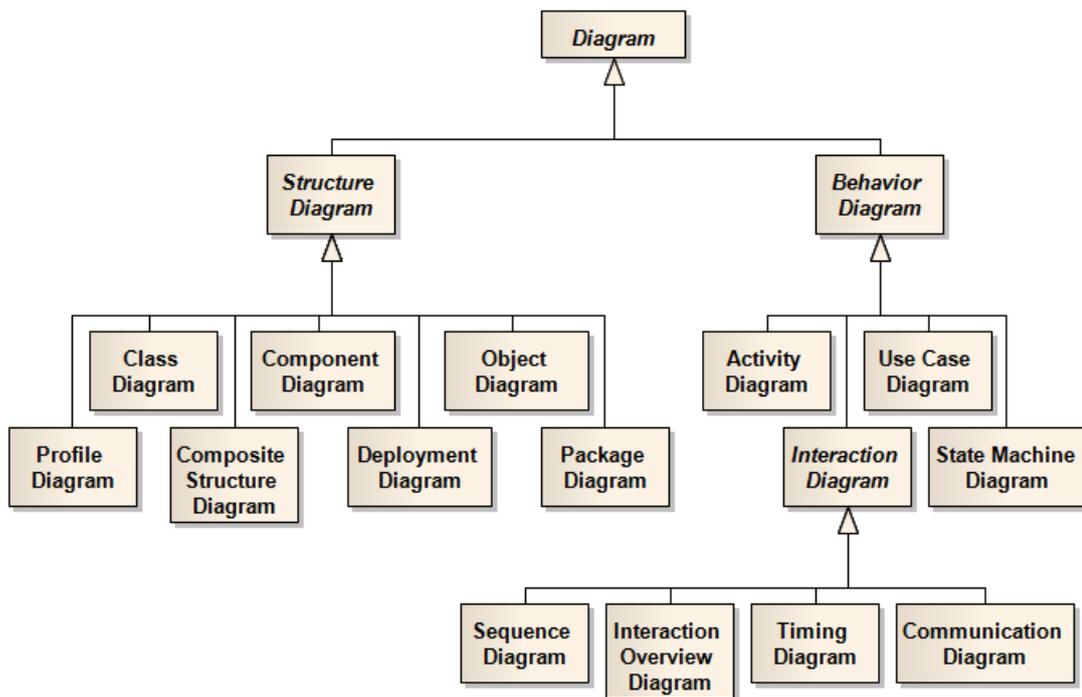


Figure 2.2: The different kinds of UML diagrams (adapted from [OMG 11.e]).

Although UML was a step forward in setting a standard understood by the whole Software Engineering community and aligning it toward MDE, it is still criticized for reasons such as: (1) being easy to use in software-specific domains, such as IT or telecom-style systems, but not in other substantially different domains, like biology or finance [Tho 04]; (2) not being oriented to how it would be used in practice [Hen 05]; or (3) being too complex as a whole [SC 01]. Nevertheless, UML is often the target of excessive promotion that raises user expectations to an unattainable level, which usually influences the criticisms that follow [FGDS 06]. An example of such a criticism is the one regarding the difficulty in using UML to model domains not related to software: although UML is a general-purpose modeling language, it is oriented toward the modeling of software systems, and not intended to model each and every domain.

UML is traditionally used as a metamodel (i.e., developers create models using the language established by UML). However, the UML specification also provides the Profile mechanism, which enables the definition of new notations or nomenclatures, providing a way to extend UML metaclasses (such as `Class` or `Association`) and adapt them for different purposes. Profiles can be considered as a collection of `Stereotypes`, `Tagged Values`, and `Constraints` [SV 05.a]. A `Stereotype` defines additional element properties, but these properties must not contradict the properties that are already associated with the original model element. Thus, a Profile does not allow the user to *edit* the existing UML metamodel, but only to *extend* it, making it more specific.

2.2.2 Meta Object Facility (MOF)

The Meta Object Facility (MOF) [OMG 11_b], along with UML, is the foundation of OMG’s approach to MDE. UML and MOF were designed to be themselves instances of MOF. This was accomplished by defining the UML Infrastructure Library [OMG 11_d], which provides the modeling framework and notation for the UML Superstructure and MOF, and can also be used for other metamodels. Figure 2.3 illustrates the dependencies between UML and MOF; note that MOF can be described using itself, making it *reflexive* [NNC 06]. Besides UML, the OMG has also defined other MOF-based standards like the XML Metadata Interchange (XMI) and Query/View/Transformation (QVT).

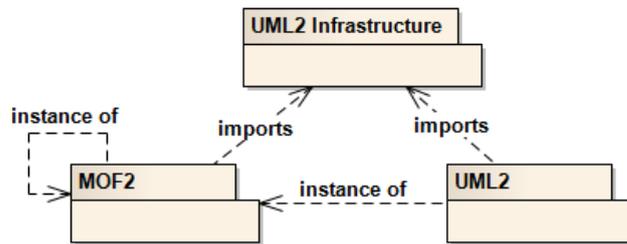


Figure 2.3: The dependencies between UML and MOF (adapted from [SS 08_a]).

When people refer to MOF, they are actually indicating one of the two MOF metamodel variants [OMG 11_b], EMOF or CMOF.

The **EMOF** (Essential MOF) is a subset of MOF that is particularly suited for use in the domain of object-oriented modeling approaches [AK 05] and XML. Thus, EMOF enables scenarios in which MOF models are mapped to XML documents (e.g., using the XMI format, presented further down this section) or even to object-oriented programming concepts. Figure 2.4 presents an overview of the facilities defined in the context of EMOF.

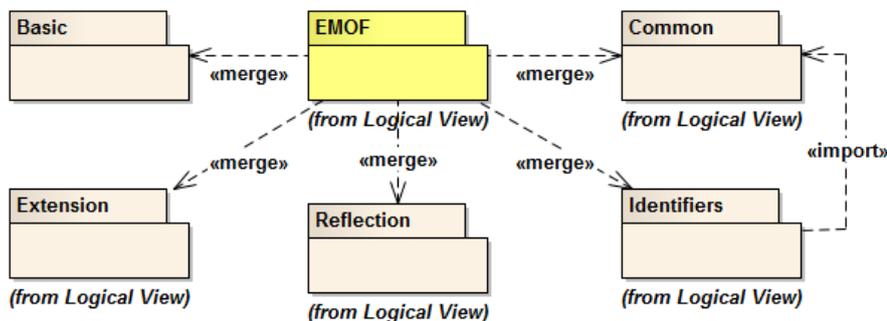


Figure 2.4: The EMOF model subset of MOF 2.0 (adapted from [OMG 11_b]).

On the other hand, the **CMOF** (Complete MOF) is what is typically meant when people mention MOF. CMOF results from combining (or *merging*, via the `Package Merge` operator) EMOF with its extensions, which provide element redefinitions that make CMOF

adequate for basic metamodeling needs [OMG 11.b]. Figure 2.5 depicts CMOF as the combination of EMOF with these additional facilities.

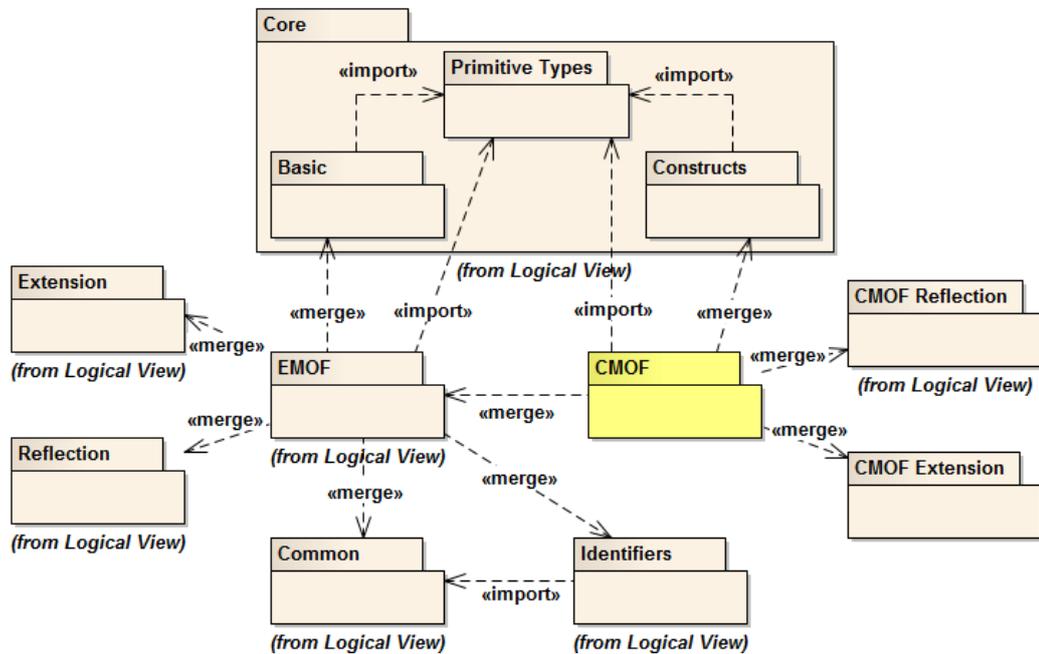


Figure 2.5: The CMOF model packages of MOF 2.0 (adapted from [OMG 11.b]).

The XML Metadata Interchange (XMI) [OMG 11.c] allows the exchange – by using XML – of any metadata whose metamodel can be specified in MOF. This allows the mapping of any MOF-based metamodel to XML, providing a portable way to serialize and exchange models between tools. Nevertheless, users often regard XMI as a last resort for exchanging models, because tools frequently use their own vendor-specific XMI extensions, potentially losing information when exchanging models between tools from different vendors.

On the other hand, the QVT specification [OMG 11.a] (further explored in Subsection 3.1.1) defines a way of transforming source models into target models, by allowing the definition of query, view, and transformation operations over models. One of the most interesting ideas about QVT is that the transformation itself is a MOF-based model, which leads to QVT being MOF-compliant.

A particularly important standard for QVT and MOF-based languages is the Object Constraint Language (OCL) [OMG 10]. Although it was originally closely tied to UML, its scope has since then expanded, and OCL has become a part of OMG’s MOF-based approach. OCL is a textual declarative language, used to specify rules and queries for MOF-based models and metamodels (including UML) when the available diagrammatic operators do not have sufficient expressiveness to specify such rules.

MOF was designed to be the cornerstone for OMG’s approach. The OMG defined a four level metamodel architecture, based mainly on MOF and UML, that not only allows users to define their own UML models, but also enables the definition of further MOF-based languages, such as the Common Warehouse Metamodel (CWM) [OMG 02]. Figure 2.6 depicts this architecture: (1) MOF is the metamodel, in the M3 metalevel; (2) UML – an instance of MOF – is the metamodel, in the M2 metalevel; (3) the user model contains model elements and snapshots of instances of these model elements at a specific time or state, specified using UML, in the M1 metalevel; and (4) the M0 metalevel contains the runtime instances of the model elements defined in the M1 metalevel.

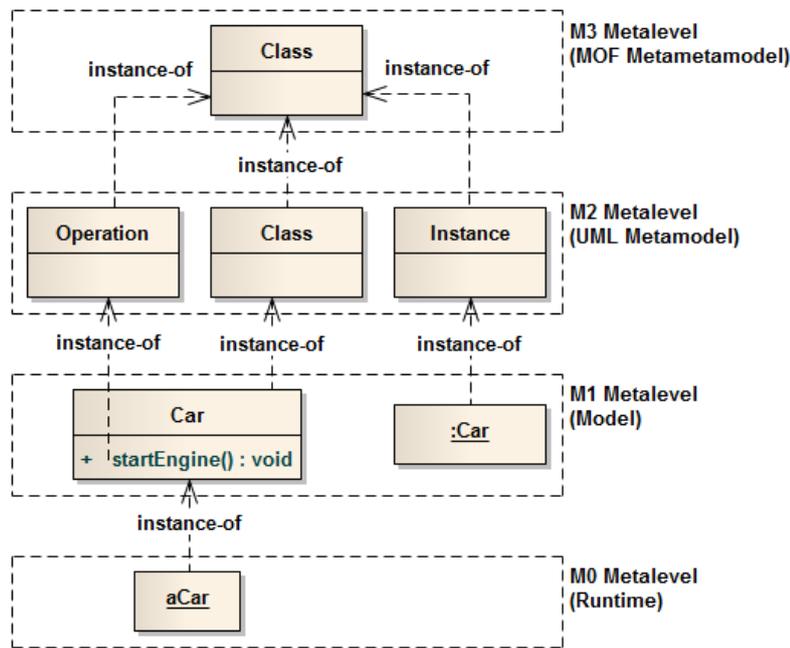


Figure 2.6: The OMG’s four level metamodel architecture (adapted from [SS 08_a]).

As Figure 2.6 shows, MOF plays the role of a metamodel (i.e., it provides a metamodeling language, according to Figure 2.1). Thus, the concepts defined by MOF (namely by EMOF) are not meant to be adequate for defining models of software systems, but rather for defining new concepts in a metamodel (as is the case in the M2 metalevel, containing the UML metamodel), thus establishing a new modeling language. Those new concepts should then be adequate for modeling the language’s intended domain.

Furthermore, another variant of MOF has been defined in the context of the Eclipse Modeling Framework⁵ [SBPM 08]. This variant, called *ECore*, is similar (but not identical) to the EMOF; however, interchange mechanisms between both languages have been defined [GR 03], enabling scenarios in which ECore models can be used by code generators (or other model-processing tools) that have been designed to work with MOF models.

⁵<http://www.eclipse.org/emf> (accessed on March 15th, 2012)

2.2.3 Model-Driven Architecture (MDA)

The Model-Driven Architecture (MDA)⁶ [KWB 03, Mel 04, OMG 03] is OMG's approach to address the software development lifecycle, driven by the activity of modeling the software system. It is mainly based on OMG's MOF standard, and places a greater emphasis on model transformations than on metamodeling itself.

MDA defines three different kinds of model [KWB 03]: (1) the Computation-Independent Model, (2) the Platform-Independent Model, and (3) the Platform-Specific Model. A **Computation-Independent Model (CIM)** is a model that depicts the system's requirements, namely by defining the characteristics of the environment in which the system will operate, and describing what the system is meant to do. On the other hand, a **Platform-Independent Model (PIM)** is a model with a high level of abstraction, and completely independent of any implementation technology. This makes it suitable to describe a software system from a generic perspective, without considering implementation or technological details such as specific relational databases or application servers. Finally, a **Platform-Specific Model (PSM)** is also a model of a software system; however, unlike a PIM, a PSM is specified in terms of implementation technology.

A PIM can be transformed into a PSM (or multiple PSMs, each targeting a specific technology) by means of *model-to-model transformations* [OMG 03] (which are further analyzed in Chapter 3). The possibility of having several PSMs derived from a single PIM is due to the fact that it is very common for current software systems to make use of several technologies (e.g., a web application server using JBoss combined with a MySQL database system). MDA prescribes the existence of transformation rules, but it does not define what those rules are; in some cases, the vendor may provide rules as part of a standard set of models and profiles.

Figure 2.7 provides an overview of MDA (CIM models are not represented for simplicity): the solid lines connecting the boxes are transformations, which are defined by transformation rules.

However, MDA still faces criticism in the Software Engineering community [Fav 04_b] because of issues such as [Tho 04]: (1) its usage of UML, a language that is still criticized; (2) when problems occur, developers have to manually debug code that was automatically generated by a tool, which can sometimes be troublesome; or (3) source code generators are usually able to generate a significant portion of an application, but the resulting source code often still requires that developers make significant efforts to integrate that code with existing platform-specific APIs (such as Java or .NET).

⁶<http://www.omg.org/mda> (accessed on March 15th, 2012)

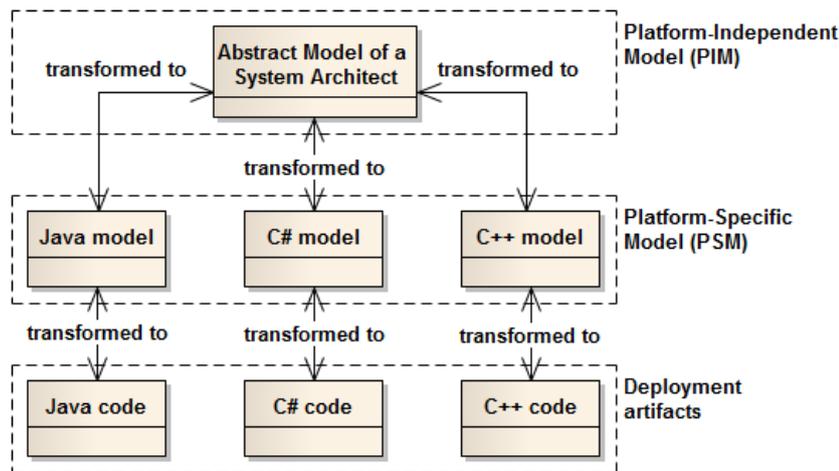


Figure 2.7: An overview of the MDA approach (adapted from [SS 08_a]).

2.3 Domain-Specific Modeling (DSM)

Domain-Specific Modeling (DSM)⁷ [KT 08, CJKW 07] uses problem-domain concepts as the basic building blocks of models, unlike traditional CASE tools which use programming language concepts (e.g., `Class`, `Object`).

From a technological perspective, DSM is supported by a DSM tool, which can be considered as an application for making domain-specific CASE tools (or, in other words, an environment to create CASE tools that can be used to produce applications). Thus, DSM tools add an abstraction level over traditional CASE, enabling the domain-specific configuration of the resulting modeling application, as illustrated in Figure 2.8. Because of this abstraction level, DSM tools are also called *language workbenches* or *metamodeling tools*.

DSM is closely related to the concept of Domain-Specific Language (DSL). A DSL is a language designed to be useful for a specific task (or set of tasks) in a certain problem-domain, as opposed to a general-purpose language [KT 08].

As Figure 2.9 illustrates, DSLs (i.e., domain metamodels) and the corresponding generators are usually specified by developers that are experts in the problem-domain, because of the highly specialized nature that underlies a DSL. Other developers, less experienced with the mapping between domain concepts and source code, will then perform their work while using the DSL.

A well-known example of a DSL is the Standard Query Language (SQL), which is a standard textual language for accessing databases (in other words, SQL is a textual DSL for the problem-domain of database querying and manipulation). A Unix shell is another typical example of a DSL, this one oriented for the domain of data and file

⁷<http://www.dsmforum.org> (accessed on March 15th, 2012)

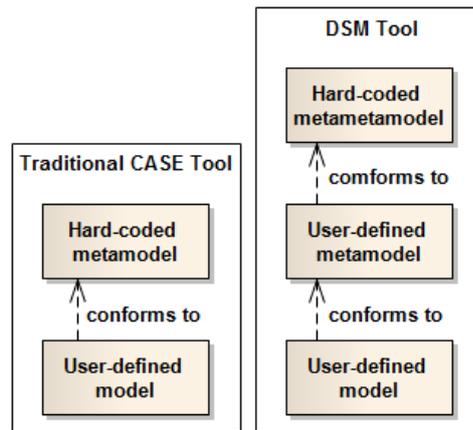


Figure 2.8: The difference between CASE and DSM tools regarding the metalevels supported (adapted from [KT 08]).

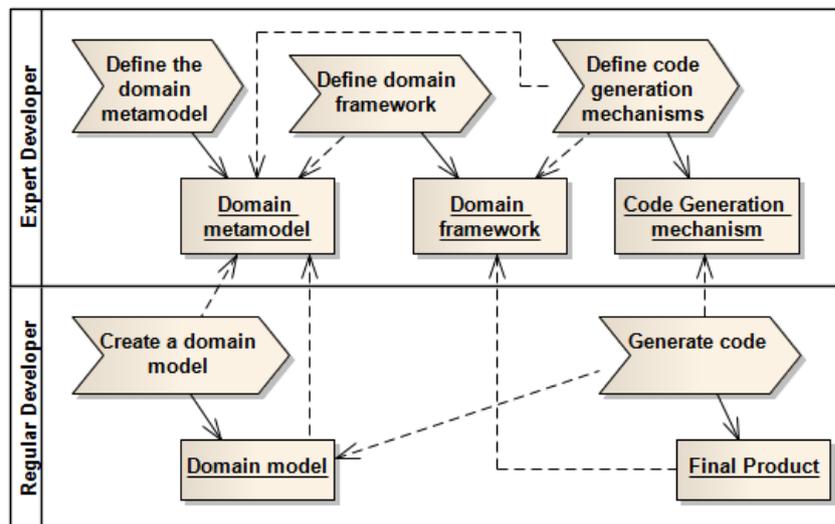


Figure 2.9: Typical DSM process: some experts develop domain-oriented tools for use by less-experienced developers (adapted from [SS 08_a]).

manipulation, as it provides a number of relevant domain concepts (e.g., file, directory, stream) and operations over those concepts, such as pipes and tasks like word counting or string replacement.

Developers typically prefer using DSLs rather than generic modeling languages such as UML, because of the concepts and abstractions that are provided by each [KT 08]. The latter uses object-oriented programming-related concepts, which places models at an abstraction level little above source code's. On the other hand, a DSL is meant to use concepts from the problem-domain, which leads to developers being able to abstract themselves from how those concepts will map to source code, in turn leading to a greater

focus on solving the problem at hand, instead of wasting time on the mapping between the problem’s solution and that solution’s implementation.

Nevertheless, even generic modeling languages are sometimes considered as DSLs, when they are meant to be used for a specific purpose. MOF itself is often considered a DSL, in which the language’s purpose is the definition of new modeling languages [AK 05]. UML can also be used to define DSLs, more specifically with the Profile mechanism.

Another important aspect to consider when defining a DSL is its *quality* (in particular its *usability*). There is a significant amount of related work regarding this topic [GGA 10, BAGB 11, BAGB 12, BMA⁺ 12], which highlight some criteria that can be used to assess the quality of a DSL. These criteria include [BAGB 11]: (1) *effectiveness*, ascertaining whether the developer can complete language sentences without committing mistakes; (2) *efficiency*, measuring the amount of effort that the developer must perform to complete language sentences; (3) *satisfaction*, determining whether the developer becomes exasperated when using the language; and (4) *accessibility*, which measures whether the language is easy to learn and memorize. The issues underlying these criteria were taken into account during our research work but, because the focus of this dissertation is on the definition of a model-driven approach (namely of its languages and synchronization mechanism), we will not further explore this topic in this document.

2.4 Metamodeling

Now that OMG’s MDA and DSM approaches have been introduced, we can focus our analysis on the underlying topic of *metamodeling*, which is crucial for the process of creating modeling languages.

By looking at the approaches presented, we can see that they are actually related among themselves, as we have a recurring pattern – the usage of metamodels and their instances, models – and one of the most significant difference (*in modeling terms*) between these approaches is the number of logical levels (i.e., metalevels that can be edited by users) considered by each one. So, aside from a question of vocabulary, they are based on the same principle of defining languages that represent a problem-domain, and then using those languages to model a corresponding solution.

Several definitions for the term “metamodeling” can be found in literature [KWB 03, SAJ⁺ 02], but all those definitions actually convey the same meaning for that term: **metamodeling** is the activity of producing a metamodel that will be used to create models. Furthermore, as was stated earlier in this chapter, a metamodel is a conceptual model of the constructs and rules needed for creating models [SV 05_b, SS 08_a].

Creating a modeling language involves some activities [SV 05_b], namely:

- Define its (1) *abstract syntax*, (2) *semantics*, and (3) *concrete syntax*, and
- Address a number of typical metamodeling issues, such as (1) defining a suitable number of metalevels that the metamodel should consider, (2) the possible usage of the Level Compaction technique, (3) reducing complexity, and (4) establishing clear rules for creating instances of modeling elements.

Of course, most of these aspects (which are presented in the following sections) are deeply related among themselves. Thus, the reader should not regard these aspects as an ordered sequence of steps in a modeling language creation process, but rather as aspects to be considered when creating modeling languages.

2.4.1 Modeling Language Components

A modeling language is composed of three main components [SV 05_b]: (1) the abstract syntax, (2) the semantics of the various modeling elements defined in the abstract syntax, and (3) the concrete syntax. The language's metamodel itself consists of the abstract syntax and its semantics [SV 05_b].

Abstract Syntax (Concepts). One of the most important aspects of creating a new modeling language is the definition of its *abstract syntax*. The **abstract syntax** of a language consists of its set of concepts and respective relationships [SV 05_b], which together establish a structure for the domain's information that is relevant to models. **Concepts** define the domain information that can be stored by models (e.g., sets of entities, each with its own specific attributes). On the other hand, **Relationships** define how concepts are related among themselves (e.g., a concept may act as a container for other concepts).

It is important to note that the abstract syntax is independent of any particular representation or encoding. Thus, it can be expressed with any kind of modeling language, such as UML, and it is not a requirement that it be designed with a computer-supported modeling tool (although doing so may help in forthcoming stages, when the concrete syntax – presented next – for the new language is being defined).

Figure 2.10 provides examples of the abstract syntax for a very simple metamodel (meant to support a social network), represented in UML: Figure 2.10a models the concepts and relationships between them (by means of UML **Classes** and **Associations**, although other modeling languages or formalisms could be used), while Figure 2.10b defines those concepts as extensions to the UML metamodel (by using the Profile mechanism).

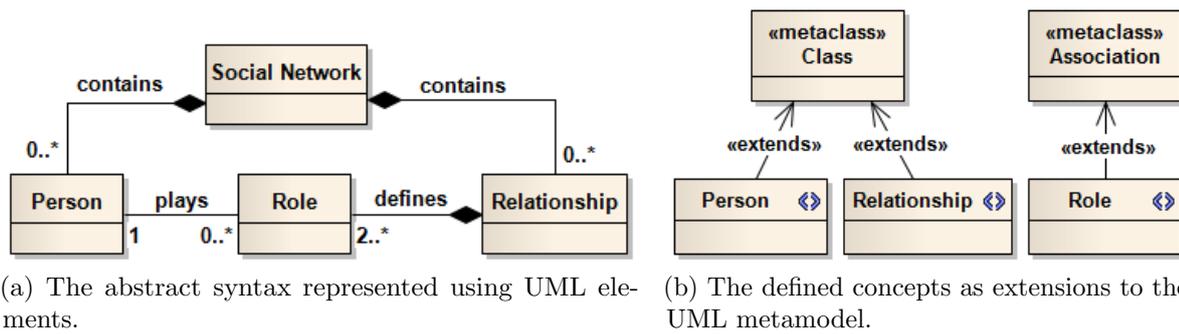


Figure 2.10: Example of the abstract syntax for a simple Social Network metamodel (adapted from [SS 08_a]).

Semantics (Rules). Semantics describe the *meaning* of each element – and of each orchestration of elements, such as sentences, diagrams, or compositions of modeling elements – that constitutes a model.

Although technology has not yet evolved to a point in which we can make a computer “understand” a model (namely to ensure its correctness), semantics can nevertheless still be addressed when creating a metamodel. This is done by specifying *rules* that restrict the different possibilities for element orchestrations in a model, in order to prevent model designers from using invalid ones.

There are some *rule specification languages*, such as OCL [OMG 10] or Alloy⁸, that can be used to define such constraints in a declarative manner. Nevertheless, most modeling language specifications typically just provide a *natural language description* of their relevant constraints, so that modeling tool creators can quickly understand those constraints (and implement them afterward). An example is the UML specification [OMG 11_e], which defines most of its constraints in natural language, using OCL for only a subset of those constraints.

Concrete Syntax (Notation). Another important aspect of modeling language design is its *concrete syntax*. The **concrete syntax** of a language (or as it is also called, its *notation*) consists of all the visual symbols (e.g., drawings, text characters) that can be used to represent domain information in the language’s models.

Metamodeling and modeling language design, because of their frequent association with MDE, are usually connoted with graphical or pictorial languages, although it is possible for a modeling language to define a textual representation instead of a graphical one. In fact, it is possible to consider most programming language grammars (e.g., Java,

⁸<http://alloy.mit.edu> (accessed on March 15th, 2012)

C#, C++) as metamodels and, with the same rationale, a corresponding program can be considered a model.

It is also possible for a modeling language to define multiple concrete syntaxes (e.g., a textual and a graphical representation). This is particularly relevant when creating modeling tools to support that language, as a textual syntax may be more adequate for model serialization purposes than a graphical syntax, even if the language’s metamodel (i.e., its abstract syntax and structural semantics) is described in a graphical manner. An example of this is that XMI can be considered as an alternate textual syntax for MOF-based models.

Regarding the definition of a modeling language’s concrete syntax, [KT 08] provides some industry best practices, such as: (1) the representation of each concept should focus on presenting the most important and relevant information; (2) different concepts should be represented with different symbols, easily distinguishable from each other; (3) the symbols for the most important concepts and relationships should be more visible than others; and (4) limited use of visual “sweeteners” such as color gradients and complex drawings or bitmaps. These best practices are particularly important in the initial stages of the language’s definition, when the concrete syntax is still usually subject to change due to the language’s ongoing validation by its users.

Figure 2.10a, which previously provided the abstract syntax for a simple UML-based metamodel, is also an example for a language’s concrete syntax – in this case, the language being UML, in accordance with OMG’s specification [OMG 11_e]. The instances of UML’s `Class` element – namely `Social Network`, `Person`, `Role`, and `Relationship` – are represented with rectangular boxes, while `Association` instances are represented as lines (with some ends indicating composition) along with their associated elements’ respective multiplicities.

2.4.2 Metamodeling Issues

Besides the definition of the modeling language components that were described in the previous subsection, the language creation activity also involves some issues and caveats that should be considered.

2.4.2.1 Strict and Loose Metamodeling

One of the best practices of metamodeling concerns the elements in a model and the metamodel elements that they instantiate, more specifically the metalevel(s) at which they are located in relation to the model elements. Regarding this issue, Atkinson and

Kühne [AK 02, Küh 09] identify two different schools of thought: (1) strict metamodeling and (2) loose metamodeling.

The **Strict metamodeling** principle states that it should be possible to fully understand any metalevel as being instantiated from *only* the metalevel immediately above it; that is, if any of the elements of a model A is an instance-of another element in a model B , then *every* element of A is an instance-of some element in B [AK 02]. Furthermore, the instance-of relationship is the only element that can cross a metalevel boundary. A consequence of strict metamodeling is that each instance-of relationship (and only these) should cross *exactly* one metalevel boundary.

On the other, the **Loose metamodeling** principle states that, even if an element of model A is an instance of an element of another model B (i.e., its metamodel), that does not mean that *every* element in A is an instance of an element in B . This principle, however, brings some problems, namely [AK 02] (1) the loss of well-defined metalevel boundaries, and (2) the need to use a non-trivial instantiation mechanism in the context of object-oriented approaches.

Figure 2.11 illustrates the difference between these two principles. More specifically, Figure 2.11a obeys the strict metamodeling rule (by having each instance-of relationship cross exactly one metalevel boundary), while Figure 2.11b uses loose metamodeling (and breaks the strict metamodeling rule) by having an instance-of relationship that does not cross any metalevel boundaries. Furthermore, Figure 2.11b deserves two additional notes: (1) the M1 metalevel is a UML model, defining instances of `Class` and `InstanceSpecification` (`Car` and `:Car`, respectively); and (2) although the instance-of relationship between `Car` and `:Car` is usually not represented in UML diagrams, the relationship is there nevertheless (in UML’s specification [OMG 11.e, p. 25], this instance-of relationship appears as an unnamed association between `InstanceSpecification` and `Classifier`, the latter being a generalization of `Class`).

2.4.2.2 Is-A: Instantiation and Specialization

One of the most important issues to be aware of when dealing with metamodeling is the ambiguity of the term “is a”, which can refer to either *specialization* or *instantiation*. Each of these possibilities carries a completely different set of semantics with it [AK 02, Küh 09]. Figure 2.12 illustrates the usage, from a metamodeling perspective, of instantiation and specialization, respectively.

Instantiation (sometimes called *classification*) consists of establishing that a certain object O_I (e.g., an in-memory Java `Object` that represents some data) has an instance-of relationship with a certain *classifier* object O_C (i.e., object O_C provides a description of

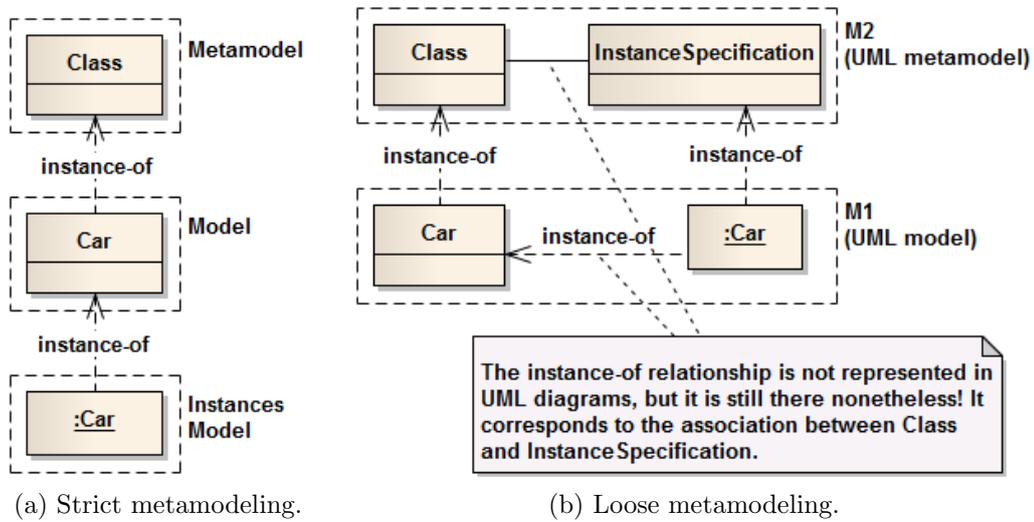


Figure 2.11: Strict and Loose metamodeling.

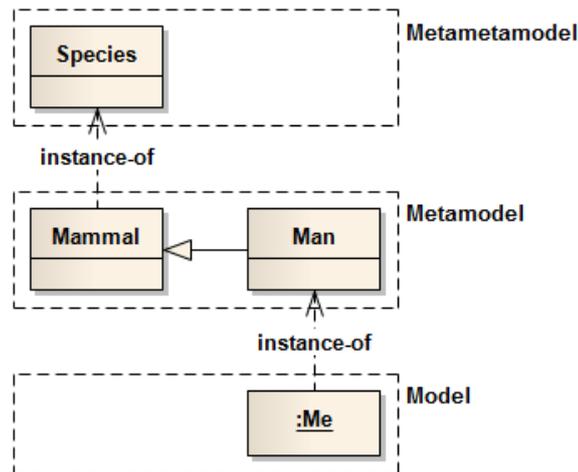


Figure 2.12: Is-A: example of instantiation and specialization.

characteristics of object O_I). For instance, the `Mammal` from Figure 2.12 has an instance-of relationship with `Species`, its classifier.

On the other hand, **Specialization** indicates a refinement of another object (i.e., taking an object and adapting to a more specific set of circumstances). Specialization can take place in a variety of forms, namely (1) by adding attributes (or slots, depending on the nomenclature used) or (2) by adding constraints.

Another way of looking at specialization is from an ontological perspective: if we have a classifier $C_{Generic}$ that describes a wide set of objects or instances, and another classifier $C_{Specific}$ that contains $C_{Generic}$'s description but with additional constraints, then we can say that (1) $C_{Specific}$ is a *specialization* of $C_{Generic}$ (or $C_{Generic}$ is a *generalization* of $C_{Specific}$), and (2) every instance of $C_{Specific}$ is *also* an instance of – or can be classified

as $C_{Generic}$, although this does not mean that all instances of $C_{Generic}$ are also instances of $C_{Specific}$. Figure 2.12 depicts an example: **Man** is a specialization of **Mammal** (an instance of **Man** has two legs, stands upright, has the ability to speak), and so every instance of **Man** is also an instance of **Mammal**, but it *cannot* be said that every instance of **Mammal** is an instance of **Man**.

Although there are similarities between the two approaches, there are also some subtle – but important – differences between them, which can help metamodel designers to decide when to use one or the other. The most relevant for our work (and possibly the easiest to use) is that the instance-of relationship is *not transitive*, but specialization is [Küh 09]. An example of this can be seen in Figure 2.12: we can say that the object **:Me** is an instance of both **Man** and **Mammal**, because of the transitivity of specialization, a statement that is backed by common sense. On the other hand, **:Me** is not an instance of **Species**, although the **Mammal** concept itself is.

2.4.2.3 Logical Levels and Level Compaction

As was previously mentioned, one of the most significant differences – *from a modeling perspective* – between the presented modeling approaches is their number of *logical levels* (i.e., *metalevels that can be edited by users*). OMG’s approach defines a metamodel architecture that consists of four metalevels (illustrated in Figure 2.6) but considers only a single logical level, the M1 level; MOF-based modeling is not available in most modeling tools, although there are exceptions [NNC 06]. On the other hand, DSM tools typically use a metamodel architecture based on three metalevels (hardcoded metamodel–user metamodel–user model).

Theoretically, the number of metalevels could be infinite. However, any particular approach should have a specific and finite number of such levels, otherwise its usage and implementation would be impractical, if not impossible [SS 08_a].

Implementing a modeling tool – as a means to create models using the tool’s corresponding approach – with *just one logical level* (i.e., hardcoded metamodel and user model editing) can be easily done with current Object-Oriented Programming (OOP) languages, by taking advantage of the class–instance relationship and making the logical level be implemented by the instance level. However, metamodeling adds *one or more logical levels* to the modeling tool (i.e., the user can edit multiple metalevels), which complicates the tool’s implementation because the instance level now has to hold two or more logical levels [AK 03].

Level Compaction [AK 05] is an OOP-oriented technique that addresses this issue, by having a logical level’s representation format supplied by the tool, instead of being defined by the level above. In other words, the OOP’s class level defines the representation

format for each logical level, and the instance level holds the various (one or more) logical levels that the modeling tool should support, as illustrated in Figure 2.13. This technique has the advantage of separating the OOP class–instance relationship from the metamodel–model relationship, which in turn removes the OOP-inherited limitation of having a single logical level *because* of the single instance level. However, it does require the use of an alternate instantiation mechanism (although not represented in Figure 2.13 for simplicity, the `instantiate` relationship should also be defined in the OOP class level, in order to be usable in the instance level).

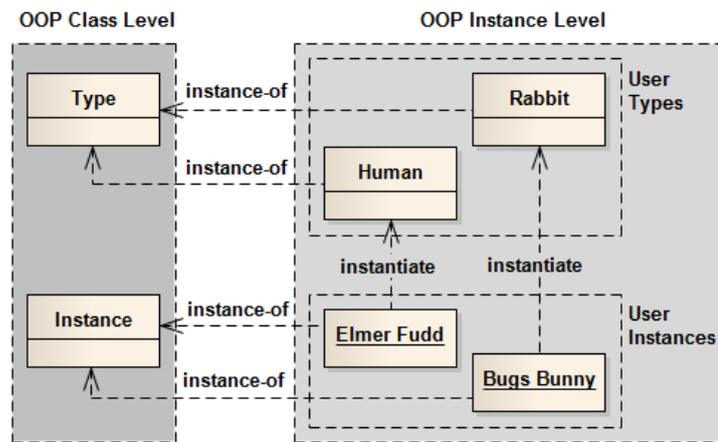


Figure 2.13: Using Level Compaction to compact two logical levels (User Types and User Instances).

Although the Level Compaction technique is important for practical support of multiple logical levels, it is also important to determine what modeling elements should be included in the hardcoded metamodel (defined in the class level). This decision, in turn, will significantly influence the architecture of the tool to support the approach, as it determines whether the number of metalevels in the tool’s architecture should be restricted or potentially unbounded⁹. Due to this influence, [AK 05] defines two different approaches, the Language metaphor and the Library metaphor. These approaches are depicted in Figure 2.14.

When using the **Language metaphor** (illustrated in Figure 2.14a), the basic elements for each of the supported metalevels (e.g., `Object`, `Class`, `MetaClass`) are contained in the hardcoded metamodel itself. If the user intends to add other basic elements necessary to support additional metalevels (e.g., `MetaMetaClass`), it would become necessary to change the hardcoded metamodel. This metaphor is typically easier (and feels more natural)

⁹Although it might seem that tools with possibly infinite metalevels might be of a limited practical value for its users (because such tools are bound to be very complex to manipulate), in practice tools can mitigate this shortcoming by treating those metalevels in a *phased* manner (i.e., as sets of metamodel–model pairs), an approach that is followed in [NNC 06].

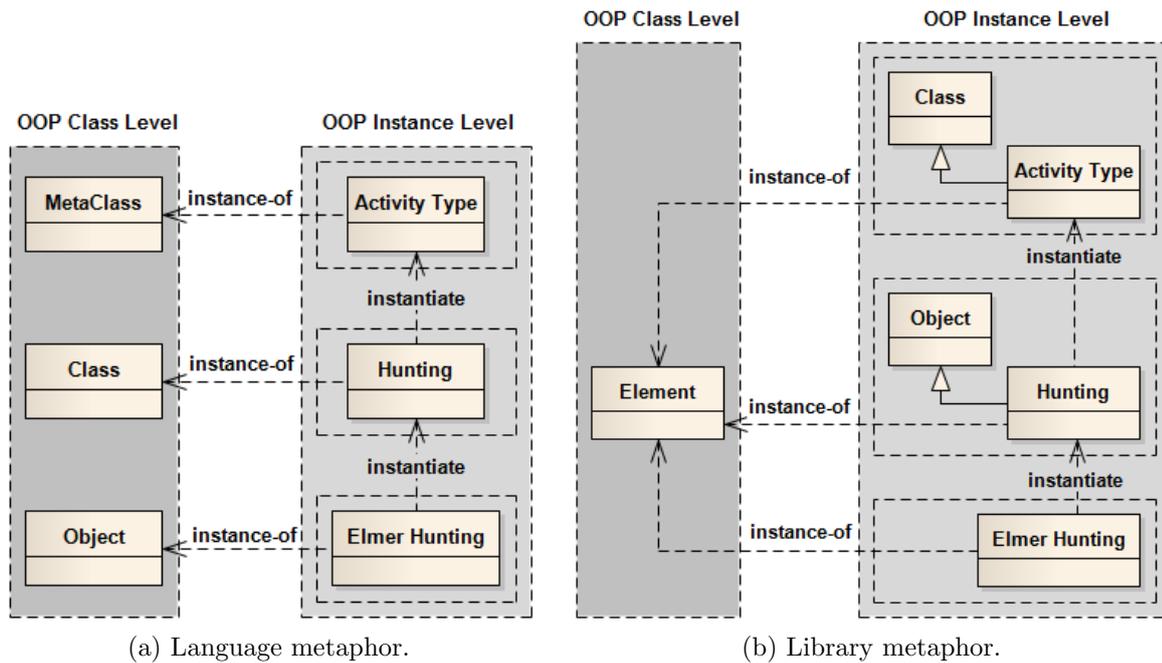


Figure 2.14: Language and Library metaphors (adapted from [AK 05]).

for a programmer to implement, and it helps in enforcing the use of a standard such as MOF [AK 05], but it does present the possible disadvantage of restricting the number of metalevels that the language can support.

On the other hand, in the **Library metaphor** (illustrated in Figure 2.14b), the hardcoded metamodel consists only of a minimal core language, and the basic elements for each metalevel are available as predefined types in *libraries*, to which the user can add elements (or remove them, if the tool allows it). This metaphor presents the advantage that users can experiment with (and change) all metalevels, because only the minimal core is hardcoded, and the burden of syntax checking and language semantics is placed on each metalevel library. However, this metaphor can also complicate the implementation of a modeling tool, as it becomes necessary to define an hardcoded *framework* (which will then be used to define each metalevel).

2.4.2.4 Ontological and Linguistic Instantiation

In addition to the Level Compaction technique, the examples illustrated in Figures 2.13 and 2.14 make a clear distinction between two different types of instantiation. These instantiation types, which are represented in the figures by **instance-of** and **instantiate** respectively, have a particular characteristic that distinguishes them from each other: all relationships marked with **instance-of** cross the OOP Class–Instance level boundary, while those marked with **instantiate** stay *within* the Instance level.

These different kinds of instantiation are identified in [AK 03, AK 05] as (1) linguistic instantiation and (2) ontological instantiation (also sometimes called just logical instantiation), respectively. **Linguistic instantiation** corresponds to the instantiation relationship that has been used until now: it is a *language-supported link* between two elements, indicating that one of the elements (the *instance*, located in the OOP Instance level) is classified according to the other (the *class*, located in the OOP Class level). On the other hand, **ontological instantiation** is a *convention-based link* between two elements (the instance and the class) that are both located in the OOP Instance level.

Thus, ontological instantiation relationships must be *explicitly* defined by the metamodel designer (e.g., in Figures 2.13 and 2.14, there should be another element `Instantiate` defined in the OOP Class Level, from which the `instantiate` links in the Instance level would themselves be instances), while the linguistic instantiation relationship (i.e., the `instance-of` link used in the aforementioned figures) is automatically provided by the modeling language itself.

2.4.2.5 Accidental and Essential Complexity

One of the most problematic issues in the activity of designing a metamodel lies in its complexity, namely in ensuring the metamodel is as simple as possible. This issue is of particular relevance for the work described in Chapters 7–8, because languages featuring a larger set of elements are likely to be harder to organize and remember (thus making it more complex and harder to learn) [Har 07]. Although this is not always necessarily true – a language can have syntactic sugar elements and not be more complex because of them, as its users are not required to learn how to use those elements –, while searching for related work regarding the complexity of modeling languages, we have found some work (such as [Ris 93] or the concept of Effective Complexity¹⁰) which makes us consider that, in most cases, it is usually possible to find a correlation between the complexity and the size of the metamodel – number of elements in the abstract syntax and the number of semantic constraints defined – in modeling languages (or grammars in programming languages).

This possible correlation between complexity and metamodel size, in turn, leads to the notion that language designers should try to keep the metamodel as simple as possible, and that they should exercise some care when specifying the abstract syntax, in order to prevent potential issues regarding the cognitive process of learning and using the language by its users. On the other hand, common sense also tells us that the number of elements in the abstract syntax is always directly influenced by the complexity of the problem-domain:

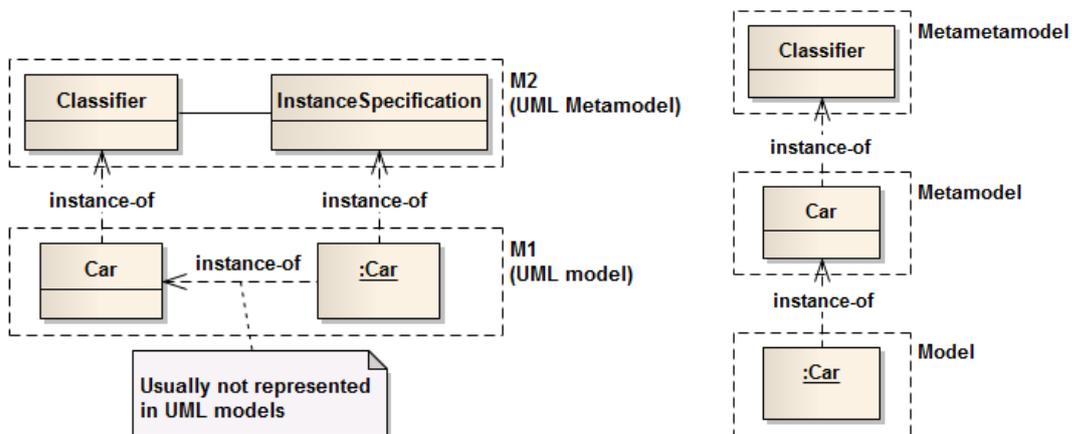
¹⁰<http://www.moscodelprado.net/complexity-language.html> (accessed on March 15th, 2012)

if the problem-domain is inherently complex, it should be expected that a corresponding language will have a proportional degree of complexity.

The existence of different kinds of complexity leads to the need for their explicit identification. Some authors provide an explicit definition of two different kinds of complexity [Bro 87, AK 08], (1) *essential complexity* and (2) *accidental complexity*.

Essential complexity is the complexity that is derived from the problem-domain itself. This kind of complexity cannot be avoided, because simpler solutions would not be sufficient to correctly represent (or solve) the problem at hand. Nevertheless, metamodel designers can – and should – try to mitigate a metamodel’s essential complexity, by ensuring that the metamodel defines *only* the modeling elements and constraints that are absolutely necessary to correctly represent the problem.

On the other hand, **accidental complexity** is not derived from the problem-domain itself, but rather from the language designer’s *approach* to represent and solve the problem; in other words, accidental complexity is introduced by the language designer (and not by the problem-domain), and is often a result of the metamodel used to define the new modeling language. This kind of complexity is typically characterized by the need to use a greater number of modeling elements than what is really necessary, and is usually derived from using modeling patterns like type–instance in the same metalevel [AK 08]. Figure 2.15 illustrates the problem that accidental complexity causes: (1) Figure 2.15a depicts a UML model, in which four relationships – three instance-of and one unnamed – are used to represent a simple model with a classifier and an instance (**Car** and **:Car**, respectively); and (2) Figure 2.15b provides a nearly equivalent model, that requires only two instance-of relationships and is thus less complex than the model in Figure 2.15a.



(a) A modeling language that suffers from accidental complexity.

(b) Accidental complexity removed.

Figure 2.15: Example of accidental complexity.

It should be noted that the model in Figure 2.15b does not explicitly define the concept of *instance*, although that is not really defined in Figure 2.15a either. Instead, the latter defines a *classifier of instances* (`InstanceSpecification`), which we consider to be a role that `Classifier` is already performing anyway.

2.4.3 The MoMM Metamodel

In addition to the metamodeling issues presented, we have also found some related work regarding the definition of metamodels for *multiple metalevels*. Namely, [AK 01] describes a metamodel called **MoMM** (short for *Metamodel for Multiple Metalevels*), which is represented in Figure 2.16.

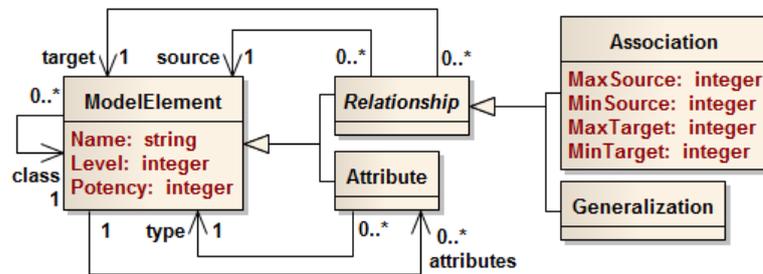


Figure 2.16: The MoMM metamodel (adapted from [AK 01]).

Meant to support the definition of modeling languages spanning multiple metalevels, MoMM is based on the notion that *everything in a model is a **ModelElement***. Some instances of `ModelElement`, in turn, may themselves be `Relationship`s between two other `ModelElement`s. Finally, a `ModelElement` can contain a set of `Attribute`s, which are themselves also `ModelElement`s.

MoMM is also *reflexive* (i.e., it can describe itself), which means that not only does it support the definition of multiple metalevels, but also *each* of those metalevels can itself contain a (re)definition of MoMM. This is of particular relevance for our research work, which uses a modified (but also reflexive) version of this metamodel, because: (1) our modeling languages (presented in Chapters 7 and 8) take advantage of this reflexiveness to easily support the use of three metalevels; and (2) our model synchronization mechanism (presented in Chapter 9) makes extensive use of the fact that everything is a `ModelElement`.

The MoMM metamodel also addresses other metamodeling issues (namely potency and deep vs. shallow instantiation) that are not fundamental for explaining the research work presented in this dissertation, which is why we will not explore them here.

Finally, there is some related work that should be mentioned regarding the MoMM metamodel. In particular, the `ModelElement` concept can be considered to be similar to the concept of *clabject* [AK 00, AK 01] (presented by the same authors that defined

MoMM), in which any element in a model – or metamodel, or even metametamodel – is simultaneously considered as *both an object and a class*. Of course, this still requires the definition of an ontological instantiation mechanism, in order to establish instance-of relationships between different clobjects.

There are also some similarities between MoMM and *category theory* [Fia 04], an area of mathematics that deals with the formalization of mathematical concepts, in order to consider them as *sets of objects and arrows*. These objects and arrows, in turn, can be considered analogous to MoMM’s `ModelElement` concept and the links that occur between `ModelElement` instances, respectively. Detailed descriptions of category theory, and its possible applications to model-driven engineering and transformations, can be found in [Dis 05_a, Dis 05_b, Fia 04].

2.5 Similarities to Programming Language Design

The design of modeling languages and the construction of compilers for text-based programming languages (e.g., C, C++, Java) present many similar aspects among themselves. Of particular relevance for this dissertation are the similarities regarding the definition of the language’s abstract syntax, semantics, and concrete syntax.

Text-based programming language compilers – or computer-based textual language processors of any kind – are typically constructed by defining [Sco 09]: (1) a scanner; (2) a parser; and (3) a parse tree processor. Figure 2.17 illustrates the various stages in the traditional compilation of a computer program, namely the components that are used; note that what we designated in this list as a parse tree processor corresponds in Figure 2.17 to the last four stages of the compilation process.

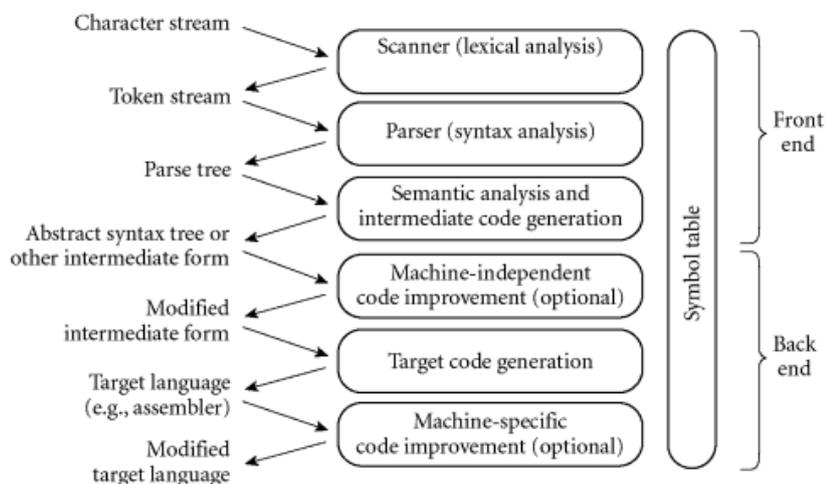


Figure 2.17: The different stages in the compilation of a program (extracted from [Sco 09]).

A **scanner**, also called a *lexical analyzer* or *lexer*, is a tool to support the process of (1) taking a sequence of characters as input and (2) outputting a corresponding sequence of tokens (also called a *token stream*). This kind of tool is defined by using support tools like `lex` [LMB 92] (or using a similar syntax), depending on the supporting technologies or frameworks. A scanner is responsible for handling the language’s textual syntax, and is typically specified as a set of pairs <recognized input characters, token to output>. At runtime, it will try to match input characters with each of the specified pairs (in order, from first to last); if a match is found, the corresponding token will be added to the output token stream. Nevertheless, it is possible for each pair to provide additional processing code, which can help determine whether the pair should be considered a match. Listing 2.1 provides a small example of a lex-based scanner specification, which captures numbers and words (returning the tokens `NUMBER` and `WORD`, respectively), and ignores whitespace.

Listing 2.1: Example of a scanner specification (adapted from [LMB 92]).

```

1 % {
2 #include "y.tab.h"
3 %}
4 %%
5 [0-9]+           { return NUMBER; }
6 [a-zA-Z]+       { return WORD; }
7 \n              /* ignore end of line */ ;
8 [ \t]+          /* ignore whitespace */ ;
9 %%

```

A **parser**, also called a *syntactic analyzer*, is another tool to support the compilation process. In particular, it is used to (1) receive a token stream as input and (2) generate and return the corresponding *parse tree*. A parser is typically based on tools like `yacc` [LMB 92], and it is defined as a *formal grammar* (i.e., a set of rules or constraints that must be followed, otherwise the input token stream will be considered invalid and the parsing fails). In turn, this grammar is specified using BNF (Backus-Naur Format) or a similar formalism. Furthermore, it is possible for each grammar rule to provide additional processing source code, which has the added advantage of enabling further semantic constraints checks. Listing 2.2 also provides a small example of a yacc-based parser specification.

Thus, a parser is responsible for mapping the program’s string (written in the language’s textual concrete syntax) into a parse tree, a data structure representing the captured elements of the language’s concrete syntax (e.g., Java’s `classes` and `interfaces`). Although optional, it is possible to further translate this parse tree into another data structure that more accurately reflects the language’s abstract syntax, instead of its concrete syntax.

Finally, the *parse tree processor* is a program that receives a parse tree and performs a set of actions (e.g., verifications, transformations to another language). An example is again

Listing 2.2: Example of a parser specification (extracted from [LMB 92]).

```
1  %{
2  #include <stdio.h>
3  %}
4  %token NAME NUMBER
5  %%
6  statement: NAME '=' expression
7           | expression           { printf("= %d\n", $1); }
8           ;
9
10 expression: expression '+' NUMBER { $$ = $1 + $3; }
11           | expression '-' NUMBER { $$ = $1 - $3; }
12           | NUMBER                { $$ = $1; }
13           ;
```

depicted in Figure 2.17, in which the parse tree processor consists of a sequence of steps that translate the input parse tree into the corresponding machine-specific assembly code. Nevertheless, it is possible to perform other operations over the parse tree: an example can be found in WebC-Docs [SS 09_b, SS 11_a], which defines a processor that receives a parse tree (obtained from a user’s search string) and converts it into a semantically equivalent object structure, which is then provided to the underlying indexing technology.

As previously mentioned, the metamodeling aspects already presented in this chapter can be considered as similar to these programming language definition components. More concretely, the scanner definition process can be considered as the equivalent to defining a program that can read the modeling language’s *concrete syntax*. Furthermore, the activity of defining a parser (and the facilities to convert the parse tree into a more abstract tree) can be considered as equivalent to defining a program that can receive a stream of the modeling language’s concrete syntax elements and mapping them to the corresponding *abstract syntax* model. Finally, the parse tree processor is equivalent to whichever facilities are created to deal with models, such as ensuring that semantic constraints are not violated or applying a model transformation to obtain source code. Even code generation can be regarded as being equivalent to model-to-model transformations (which are analyzed in Chapter 3), if we consider a destination modeling language with a textual concrete syntax.

Summary

In this chapter, we have presented the MDE paradigm and some important modeling language creation approaches which are relevant to the work presented in this dissertation, namely the MDA software development approach (which is based on the usage of model transformations and of standards like the UML or MOF modeling languages) and the DSM approach (which is supported by the definition and usage of DSLs).

Furthermore, we have explained and discussed some metamodeling and modeling language design issues, namely (1) a modeling language’s abstract syntax, semantics, and concrete syntax, (2) the usage of Strict or Loose metamodeling, (3) the potential ambiguity of the is-a relationship between two modeling elements, (4) the Level Compaction technique and its effect on the possible number of modeling levels allowed, and (5) the accidental vs. essential complexity in modeling languages. Figure 2.18 shows a simple mindmap overview of the analyzed approaches and the issues that were presented and discussed.

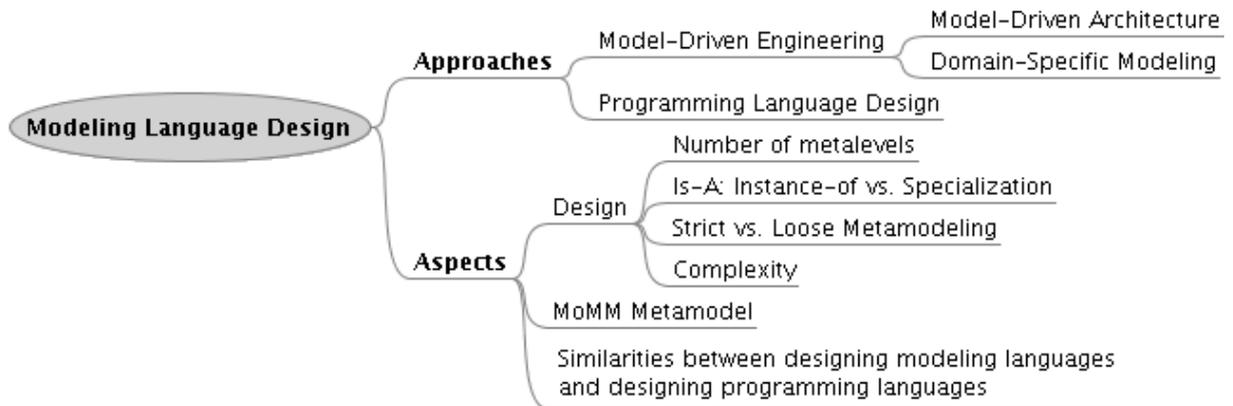


Figure 2.18: Overview of approaches and metamodeling aspects.

In the next chapter, we elaborate on the subject of model transformations in model-driven engineering and in traditional software development activities. These transformations assume particular relevance in this dissertation, because some of the presented transformation mechanisms provide the underlying concepts and ideas for the solution presented in Chapters 6 and 9. The solution presented in those chapters requires the definition of a model synchronization mechanism, which is based on these transformations.

Chapter 3

Model Transformations

I object to doing things that
computers can do.

OLIN SHIVERS

In Chapter 2 we present some approaches for creating modeling languages, as well as some aspects that modeling language designers should take into consideration. However, it would be unreasonable to assume that metamodels – and models – do not change over time. Some of those changes will be performed by users (e.g., editing the model, adding a new element to the metamodel, changing a name), but other model changes – namely those resulting from repetitive tasks, such as updating class name prefixes (because of changes to the metamodel) or adding trace elements – should be performed automatically.

The issue of how to change a metamodel assumes particular relevance because models, consistent in the context of a certain metamodel, can become inconsistent with even a small number of changes to that metamodel (e.g., if an element is removed from the metamodel) [MV 11]. Obviously, this introduces a potential element of disruption that should be avoided.

One possible way of ensuring the validity of existing models, when changing their corresponding metamodel(s), is through the definition of a migration process based on *model-to-model transformations*: for any change to a metamodel, a corresponding transformation must be defined, receiving a (previously consistent) model and producing a new model that is now consistent with the updated metamodel. If each model transformation is *correct* (i.e., it correctly specifies how to transform an input model to another output model) and the input model is also correct (i.e., it is well-formed according to its metamodel *and* it correctly expresses the reality that the model aims to represent), then the model that results from composing those transformations will also be correct. In other words, if we assume that:

- M_a represents a metamodel in a version a (i.e., the metamodel M is in a certain stage a of its lifetime),
- m^{M_a} represents a model that expresses a *certain meaning* m using the concepts provided by metamodel M_a ,
- $\forall a, b : \delta_b^{M_a}(M_a) \Rightarrow M_{a+1}$
 (Applying any change δ_b to a metamodel M_a originates a new metamodel M_{a+1} , which consists of metamodel M but now in the next stage, $a + 1$, of its lifetime),
- $\forall a, b : \delta_b^{M_a} \rightarrow T_b^{M_a}$
 (Any change $\delta_b^{M_a}$ that is applicable to a metamodel M_a , can also be mapped to a *corresponding transformation* $T_b^{M_a}$ that receives an M_a model as input), and
- $\delta_c^{M_a}(M_a) = M_{a+1} \rightarrow T_c^{M_a}(m^{M_a}) = m^{M_{a+1}}$
 (A model transformation T_c that results from a metamodel change $\delta_c^{M_a}$, when applied to a model m^{M_a} defined using metamodel M_a , originates a new model $m^{M_{a+1}}$ that has the same meaning but is defined with metamodel M_{a+1}),

then we can infer that composing the transformations that result from the various changes to a metamodel will result in a transformation that effectively *upgrades* corresponding models to another version of the metamodel:

$$\forall x, i = 1, \dots, n : \delta_{i+1}^{M_{x+1}} \circ \delta_i^{M_x}(M_x) = M_{x+2} \rightarrow T_{i+1}^{M_{x+1}} \circ T_i^{M_x}(m^{M_x}) = m^{M_{x+2}}$$

Of course, metamodel changes are not the sole motivation for the definition of model-to-model transformations. Another worthwhile (and more common) scenario is providing a model M_A and obtaining a corresponding model M_B in another language.

In the context of our research work, we have studied and analyzed a small set of MDE-oriented model transformation languages, namely: (1) the OMG's Query/View/Transformation (QVT) language [OMG 11_a]; (2) the OMG's MOF Model To Text Transformation Language (MOFM2T) [OMG 08]; and (3) AtlanMod's ATL Transformation Language (ATL). We have also analyzed some traditional software development topics and mechanisms that can be considered as related to the topic of model transformations, namely (1) the usage of SQL scripts, (2) the usage of Object-Relational Mapping (ORM) frameworks and the definition of Migrations, (3) source code management tools, and (4) the compilation of computer programs to executable form. Finally, we also provide an analysis of XSLT (eXtensible Stylesheet Language Transformations), as it can be considered a model transformation mechanism for XML-based models.

These languages and mechanisms (some of which are frequently used in software development processes) were chosen either because of their relevance for our research work, or because they share some underlying concepts with existing model transformation

languages. Furthermore, like in Chapter 2, it is important to note that this analysis is not meant to present these languages and mechanisms in detail.

3.1 Transformations in Model-Driven Engineering

The topic of model transformations is the subject of a number of MDE-related research efforts, the most famous of which is perhaps OMG’s QVT. Nevertheless, other model transformation languages exist. Of these, besides QVT we also highlight and introduce OMG’s MOF Model To Text Transformation Language (MOFM2T) and AtlanMod’s ATL Transformation Language (ATL).

3.1.1 Query/View/Transformation (QVT)

The Query/View/Transformation (QVT) specification [OMG 11_a], created to be a cornerstone of OMG’s MDA approach, defines a standard way of transforming a source models into another target model, in which both the source and target models conform to an arbitrary MOF-based metamodel.

A QVT transformation is specified as a model that defines queries, views, and transformations over an input model. This QVT model is defined according to one of the three metamodels provided by QVT [OMG 11_a]: (1) *QVTOperational* (also called *Operational Mapping*); (2) *QVTRelation* (also known as just *Relations*); and (3) *QVTCore* (also known as just *Core*). The first metamodel, *QVTOperational*, provides an imperative language (with side effects), while the other metamodels define declarative languages. Each of them is also MOF-compliant, and so QVT-based model transformations are also MOF-compliant, which enables scenarios such as storing such model transformations in a model repository, for example. Figure 3.1 illustrates the relationship between these QVT metamodels and MOF.

These three metamodels are the backbone of QVT’s architecture, depicted in Figure 3.2. Transformations are ideally specified using the *Relations* language (and *Operational Mapping*, if the transformation requires side effects) and then converted to the *Core* language, by using the *RelationsToCore Transformation* mechanism. Furthermore, QVT also allows the usage of external processing utilities (e.g., an XML transformation using XSLT, which is described further in Section 3.4) via the *Black Box* mechanism.

As Figure 3.1 shows, QVT depends not only on MOF but also on OCL, as the latter is used in all of the QVT metamodels to define predicates and constraints. Nevertheless, QVT also extends OCL with imperative features, which are then used to endow *QVTOperational* with a procedural programming style.

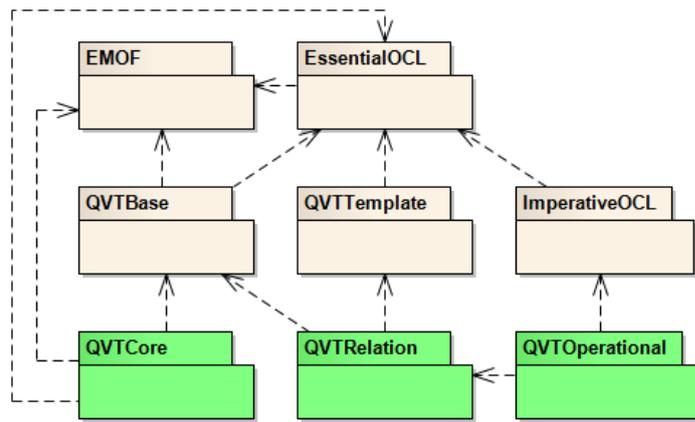


Figure 3.1: QVT metamodels and their relationship to MOF and OCL (adapted from [OMG 11_a]).

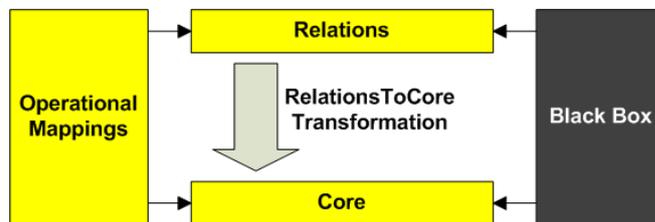


Figure 3.2: Overview of QVT's metamodel architecture (adapted from [OMG 11_a]).

Relations. The *QVTRelation* (Relations) metamodel enables the specification, in a declarative manner, of a transformation as a set of **relations** between models. Each **relation** involves two or more **domains**, as well as a **where** clause and a **when** clause (each of which is optional). A **domain** can be considered as a typed parameter for a transformation (i.e., the source or the target models). A **relation** defines, for each of its **domains**, a *pattern* that should hold (i.e., be matched) if the transformation is to be considered a success. The necessity for a certain **relation** to hold is determined by its type: if it is a **top relation**, then it *must* hold; otherwise, it must hold only if is used (directly or indirectly) in the **where** clause of another relation that must hold. The **when** clause specifies the circumstances under which the **relation** must hold – besides the circumstances involved in the context of the **relation's** invocation –, while the **where** clause specifies the condition(s) that must be verified by the model elements involved in the **relation**. When comparing these clauses to computer programming concepts, the **when** and **where** clauses are similar to a *guard* and a *postcondition*, respectively.

Furthermore, each **domain** is either **enforced** or **checkonly**, which indicates if its elements may be modified by the transformation or not, respectively (in order to uphold the corresponding pattern). This, in turn, leads to the fact that a **relation** is also either unidirectional or bidirectional, depending on whether its **domains** are all **enforceable**.

Another characteristic of this language is that it implicitly establishes *trace relationships* between the elements in the source and target models.

QVTRelation provides both a graphical and a textual concrete syntax, as suggested by Listing 3.1 and Figure 3.3: the former contains an excerpt of a textual QVTRelation transformation that converts a UML model to a relational model, while the latter provides a graphical diagram of a relation equivalent to the `ClassToTable` relation in Listing 3.1.

Listing 3.1: Example of a QVTRelation transformation (excerpt from [OMG 11_a]).

```

1 transformation umlToRdbms(uml:SimpleUML, rdbms:SimpleRDBMS) {
2   top relation PackageToSchema { // map each package to a schema
3     pn: String;
4     checkonly domain uml p:Package {name=pn}; enforce domain rdbms s:Schema {name=pn};
5   }
6   top relation ClassToTable { // map each persistent class to a table
7     cn, prefix: String;
8     checkonly domain uml c:Class {namespace=p:Package {}, kind='Persistent', name=cn};
9     enforce domain rdbms t:Table {schema=s:Schema {}, name=cn, column=cl:Column
10      {name=cn+'_tid', type='NUMBER'}, key=k:Key {name=cn+'_pk', column=cl}};
11     when { PackageToSchema(p, s); }
12     where { prefix = ''; AttributeToColumn(c, t, prefix); }
13   }
14   relation AttributeToColumn {
15     checkonly domain uml c:Class {}; enforce domain rdbms t:Table {};
16     primitive domain prefix:String;
17     where {
18       PrimitiveAttributeToColumn(c, t, prefix); ComplexAttributeToColumn(c, t, prefix);
19       SuperAttributeToColumn(c, t, prefix);
20     }
21   }
22   function PrimitiveTypeToSqlType(primitiveType:String):String {
23     if (primitiveType='INTEGER') then 'NUMBER'
24     else if (primitiveType='BOOLEAN') then 'BOOLEAN' else 'VARCHAR' endif
25   }
26   (...)
27 }

```

Operational Mapping. On the other hand, the *QVTOperational* (Operational Mapping) metamodel enables the specification of model transformations in an imperative manner, either by defining *operational transformations* (i.e., transformations that are expressed imperatively) or by complementing relational transformations with imperative operations [OMG 11_a].

An operational transformation can only be unidirectional and, like a relational transformation, defines a signature that consists of the types of its input and output models (e.g., UML, RDBMS). Other than that, various similarities can be established between operational transformational and Object-Oriented Programming (OOP) classes, namely

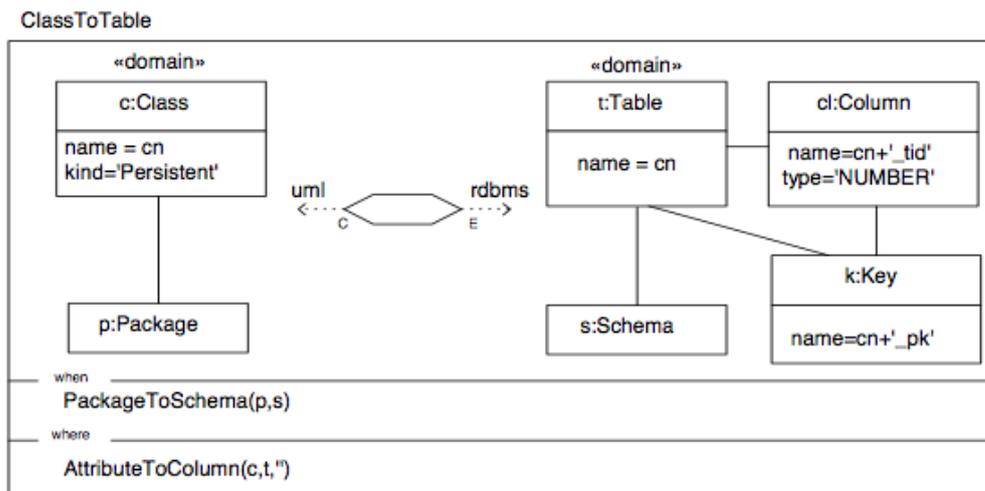


Figure 3.3: Example of the concrete syntax for a QVTRelation relation (extracted from [OMG 11_a]).

that each transformation can be instantiated as an object. Also, due to its nature, an operational transformation provides an executable entry point called `main`.

A transformation is composed of, among other elements, a set of **mapping** operations. A **mapping** operation, as its name suggests, establishes a mapping between elements in the source and target models. Each **mapping** operation is defined by (1) its signature (its source and target elements), (2) a **where** clause and a **when** clause (both optional), and (3) a body. The body, in turn, is composed of a set of instructions (OCL combined with the aforementioned imperative extensions), divided into **init**, **population**, and **end** sections. The **init** section contains instructions to run before the target model elements (the operation's results) are instantiated, while the **population** section provides instructions to configure and populate those results, and the **end** section contains instructions to run before ending the operation. QVTOperational provides instructions for purposes such as (1) object creation and population, (2) invoking helpers (operations that receive a set of source elements and return a result), (3) using intermediate classes and properties (transient data attached to target elements), or (4) invoking auxiliary transformations.

It is also possible to use some facilities typically present in imperative languages like C and C++. One of these is **disjunction**, which is similar to the `switch` statement and allows choosing between which **mapping** operation to use, depending on their guards (the element types and the **when** clause). Another facility is **type extensions**, which defines new types that act as templates for other existing types, in a manner very similar to C's `typedef` statement.

Finally, the metamodel also provides a set of variables out-of-the-box, namely: (1) `this`, referencing the transformation instance itself; (2) `self`, which references the source model

element that establishes the context for the `mapping` operation; and (3) `result`, which references either a) the resulting model element, if the `mapping` declares a single result element, or b) a tuple of the resulting elements. The `null` value is also supported, like in typical OOP languages, to indicate an absence of a value.

Unlike Relations, the QVTOperational metamodel provides only a textual concrete syntax, an excerpt of which is illustrated in Listing 3.2 (any characters between `--` and the line's end represent comments). This example depicts another way to represent the UML-to-relational model transformation that was previously presented in Listing 3.1.

Listing 3.2: Example of a QVTOperational transformation (excerpt from [OMG 11_a]).

```

1 transformation Uml2Rdb(in srcModel:UML,out dest:RDBMS) {
2   -- entry point: 1. tables created from classes, 2. tables updated with foreign keys implied by associations
3   main() {
4     srcModel.objects()[Class]->map class2table(); -- first pass
5     srcModel.objects()[Association]->map asso2table(); -- second pass
6   }
7   -- maps a class to a table, with a column per flattened leaf attribute
8   mapping Class::class2table () : Table when {self.kind='persistent';} {
9     init { -- performs any needed initialization
10      self.leafAttribs := self.attribute->map attr2LeafAttribs("", "");
11    }
12    -- populate the table
13    name := 't_' + self.name;
14    column := self.leafAttributes->map leafAttr2OrdinaryColumn("");
15    key_ := object Key { name := 'k_' + self.name; column := result.column[kind='primary']; };
16  }
17  -- mapping to update a Table with new columns of foreign keys
18  mapping Association::asso2table() : Table {
19    init {result := self.destination.resolveone(Table);}
20    foreignKey := self.map asso2ForeignKey(); column := result.foreignKey->column;
21  }
22  -- mapping to build the foreign keys
23  mapping Association::asso2ForeignKey() : ForeignKey {
24    name := 'f_' + self.name; refersTo := self.source.resolveone(Table).key_;
25    column := self.source.leafAttribs[kind='primary']->map leafAttr2ForeignColumn(self.source.name+'_');
26  }
27  (...)
28 }

```

Core. Finally, the *QVTCore* (Core) establishes a small language oriented toward pattern matching, like Relations. Also declarative in nature, Core is as expressive as Relations, although it is simpler (i.e., it defines a smaller number of modeling elements) [OMG 11_a]. Although it is possible for QVT designers to specify a model transformation entirely with Core instead of Relations, Core is meant for a more traditional scenario in which (1) designers specify their transformations with Relations, and then (2) use the RelationsToCore transformation, shown in Figure 3.2, to convert it to Core.

Because Core defines a smaller number of elements, it is more verbose than Relations, as it requires a greater number of modeling element instances to express the same intent than the latter. However, because Core is meant to be as expressive as Relations, it can be deduced that Relations consists of syntactic sugar for Core.

Furthermore, some of the elements that are established implicitly in Relations must be explicitly defined in Core, such as traceability elements (which establish correspondence between elements in the source and target models).

Like in Relations, a Core transformation consists of a set of mappings between different **domains** (corresponding to models of different types), which define patterns that must hold in the source and target models. Each transformation can be run in either *enforcement mode* or *checking mode*. When running in **enforcement mode**, the transformation is run in a specific direction, from a source to a target model. On the other hand, in **checking mode**, the transformation only verifies whether the constraints hold in both models.

Core separates each **domain's** patterns into *guard* and *bottom* patterns. It also defines an extra set of patterns, called *middle*, that are used to establish the trace elements between the source and target models (among other purposes) and can depend on patterns from other **domains**, as illustrated in Figure 3.4. The guard patterns serve the same purpose of Relations' **when** clause (i.e., to provide further constraints on whether the transformation can occur), while the bottom patterns define the structure of the model elements that the transformation will actually check or enforce. Thus, the model transformation, defined in the bottom patterns, will only take place if the corresponding guard patterns are successfully matched.

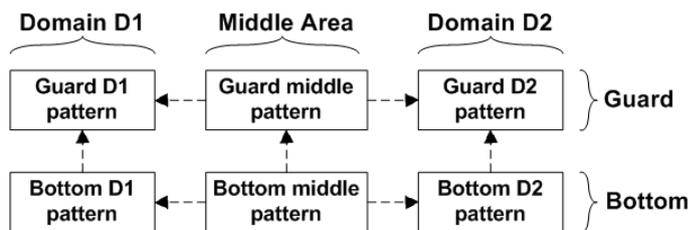


Figure 3.4: QVT Core patterns and dependencies between them (adapted from [OMG 11_a]).

A successful pattern match can establish a set of *bindings*. A binding consists of a set of unique values – model elements – that are assigned to the variables (i.e., names whose value may vary through time) in the matched pattern.

A pattern consists of [OMG 11_a]: (1) variables, identifying model elements of interest in successful pattern matches; (2) predicates, specifying verifications to be made; (3) realized variables, which allow the modification of their values; (4) assignments, which attribute values to objects; and (5) black-box operations, which make changes to enforced models,

and are only used in bottom patterns. The last three can have side effects, and so they are only really effective when the transformation is run in enforcement mode.

In the case of realized variables, it is not the binding itself that causes the side effects, but rather the possibility of modifying them, namely by creating and/or deleting their values (which, in turn, corresponds to model changes in the enforced model).

Finally, assignments can be either default or non-default. **Default assignments** only take place (i.e., are evaluated) when enforcing patterns, and are used to set the value of an object when enforcing a bottom pattern of either the target domain or the middle. On the other hand, **non-default assignments** take place even when just performing pattern checks (and thus also perform the role of predicates), and can be used to verify if a model provides a match for a pattern that involves some sort of algorithm or computation.

Unlike Relations, Core only provides a textual concrete syntax, which is depicted in Listing 3.3 (like in QVTOperational, any characters between `--` and the line's end are comments). This listing provides an excerpt from a Core transformation, defined in [OMG 11.a], to which we have added some comments to pinpoint the usage of some concepts that were just now presented. Like the examples in Listings 3.1 and 3.2, this transformation receives a UML model and transforms it into a relational model.

The QVT specification also mentions an expected usage of Core as an intermediate language for a *QVT virtual machine*, in a manner very similar to Java's bytecode and virtual machine.

3.1.2 MOF Model To Text Transformation Language (MOFM2T)

The MOF Model To Text Transformation Language (MOFM2T) [OMG 08], or *Mof2Text*, is also part of OMG's MDA approach, and is aimed at enabling the transformation of graphical MOF-based models to text, such as source code or documentation. In the context of MDA (previously illustrated in Figure 2.7), MOFM2T is used for *PSM to source code* transformations, while QVT is used for *PIM to PSM* transformations.

Unlike QVT (which establishes mappings between patterns in source and target models), MOFM2T's approach is based on the usage of *templates*. It should be noted this kind of approach is also frequently used in traditional software development, namely in code generation tools; a typical example is the generation of a data access layer (DAL) based on a model, such as a UML model or even a relational database's table structure.

Like other template-based approaches [SV 05_b, SSF⁺ 07], defining a MOFM2T model transformation consists of defining templates that receive elements in a source model and produce the corresponding output text. A template consists of *plain text* to be included in the resulting artifact, combined with *placeholders* into which data from the model will be

Listing 3.3: Example of a QVTCore transformation (excerpt from [OMG 11_a]).

```

1 -- A transformation definition from SimpleUML to SimpleRDBMS
2 module UmlRdbmsTransformation imports SimpleUML, SimpleRDBMS {
3   transformation umlRdbms { uml imports SimpleUML; rdbms imports SimpleRDBMS; }
4   -- Package and Schema mapping
5   class PackageToSchema {
6     composite classesToTables : Set(ClassToTable) opposites owner;
7     composite primitivesToNames : Set(PrimitiveToName) opposites owner;
8     name : String;
9     umlPackage : Package; -- uml
10    schema : Schema; -- rdbms
11  }
12  map packageToSchema in umlRdbms {
13    uml () { p:Package } -- domain
14    rdbms () { s:Schema } -- domain
15    where () { -- empty middle guard pattern
16      p2s:PackageToSchema | p2s.umlPackage = p; p2s.schema = s; -- middle bottom pattern
17    } -- p2s is a realized variable
18    map { -- composed mapping example
19      where () { p2s.name := p.name; p2s.name := s.name; p.name := p2s.name; s.name := p2s.name; }
20    }
21  }
22  map primitiveToName in umlRdbms {
23    uml (p:Package) { prim:PrimitiveDataType | prim.owner = p; } -- prim is a variable
24    check enforce rdbms () { sqlType:String } -- domain is checkable and enforceable
25    where (p2s:PackageToSchema | p2s.umlPackage=p) {
26      realize p2n:PrimitiveToName | p2n.owner := p2s; p2n.primitive := prim; p2n.typeName := sqlType;
27    }
28    map { where () { p2n.name := prim.name + '2' + sqlType; } }
29  }
30  (...)
31 } -- end of module

```

inserted. Placeholders, in turn, are expressions that (1) express queries over elements of the source model, and (2) select specific data from those elements and outputs it. Within the template, a placeholder is contained between the characters [and].

Furthermore, MOFM2T supports *template composition*, which enables the definition of simple templates that invoke other templates, instead of forcing transformation designers to define complex templates with a large amount of logic within placeholders. Listing 3.4 presents two examples of template definitions that are semantically equivalent (as they produce the same text): (1) the `classToJava` template in lines 1–6 invokes the `attributeToJava` template (which is defined in lines 7–9), while (2) the `classToJava` template in lines 11–18 performs the same function but without invoking any other template.

Although MOFM2T transformations just receive an input model and output plain text, it is possible to redirect the output to a file. The template illustrated in Listing 3.5 redirects all output to a file called `<class name>.java` (line 2), while outputting the string “processing `<class name>`” to a file `log.log` (line 3).

Listing 3.4: Example of MOFM2T templates, with and without composition (excerpt from [OMG 08]).

```

1 [template public classToJava(c : Class)]
2 class [c.name/] {
3   // Attribute declarations
4   [attributeToJava(c.attribute)/]
5 }
6 [/template]
7 [template public attributeToJava(a : Attribute)]
8 [a.type.name/] [a.name/];
9 [/template]
10
11 [template public classToJava(c : Class)]
12 class [c.name/] {
13   // Attribute declarations
14 [for(a : Attribute | c.attribute)]
15   [a.type.name/] [a.name/];
16 [/for]
17 }
18 [/template]

```

Listing 3.5: Example of a MOFM2T template that redirects output to different streams (excerpt from [OMG 08]).

```

1 [template public classToJava(c : Class)]
2 [file ('file:\\'+c.name+'.java', false, c.id + 'impl')]
3 [file('log.log', true)] processing [class.name/] [/file]
4 class [c.name/] {}
5 [/file]
6 [/template]

```

Finally, MOFM2T also supports the definition of *macros*, a mechanism – very similar to C’s own macros – through which transformation designers can add new syntactic sugar elements to the language. These new elements, in turn, are able to receive a number of parameters (possibly none) as well as a parameter of type `Body`. Listing 3.6 provides an example of a macro: lines 1–7 define a macro called `javaMethod` that generates the Java source code for a method, and lines 9–11 invoke that same `javaMethod` macro. When such an invocation is made, the macro is expanded, and the body (specified between `[javaMethod ...]` and `[/javaMethod]`) is included in the output text.

3.1.3 ATL Transformation Language (ATL)

The ATL (ATL Transformation Language)¹ is a model transformation language developed by the AtlanMod research team². Originally developed to answer the QVT Request For

¹<http://www.eclipse.org/at1> (accessed on March 15th, 2012)

²<http://www.emn.fr/z-info/atlanmod> (accessed on March 15th, 2012)

Listing 3.6: Example of a MOFM2T macro (excerpt from [OMG 08]).

```

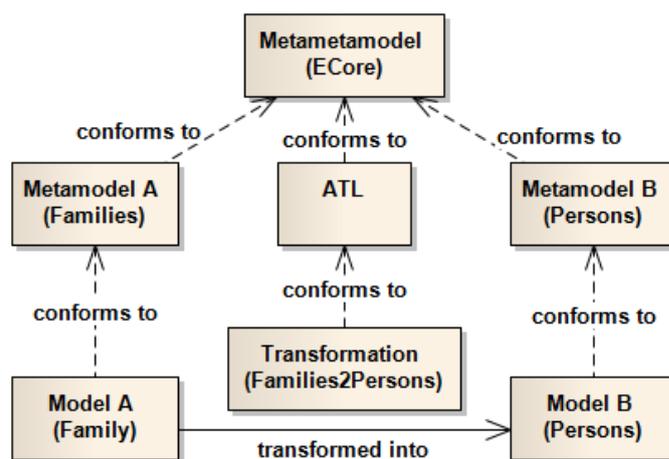
1 [macro javaMethod(Type type, String methodName, String resultName, Body body)]
2 public [typeName(type)/] [methodName/] () {
3   [typeName(type)/] [resultName/] = null;
4   [body/]
5   return [resultName];
6 }
7 [/macro]
8
9 [javaMethod([query.oclExp.type/], query.name, "result")]
10 result = [javaocl(query.oclExp)/];
11 [/javaMethod]

```

Proposals, it provides a more natural approach to model transformation, and it has been implemented as a plugin for Eclipse’s M2M (Model To Model)³ project. A public repository for ATL transformations⁴ has also been created.

In addition to the Eclipse-based modeling and transformation tools produced in the context of this project, the ATL project has also resulted in KM3 (Kernel Meta Meta Model) [JB 06], a metamodeling language that provides concepts similar to EMOF’s or ECore’s, and can be used to create metamodels, namely when defining new DSLs.

ATL requires that the source and target metamodels conform to the ECore metamodel. Nevertheless, most metamodels that are used today – namely MOF, KM3, and even ECore itself – fulfill this requirement, although a model transformation may be necessary in some cases (e.g., to convert KM3 models to ECore models⁵). Figure 3.5 illustrates ATL’s model transformation architecture, with an example of a `Family2Persons` transformation receiving a `Families` model and producing a corresponding `Persons` model.

Figure 3.5: ATL overview (adapted from <http://wiki.eclipse.org/ATL/Concepts>).³<http://www.eclipse.org/m2m> (accessed on March 15th, 2012)⁴<http://www.eclipse.org/m2m/at1/at1Transformations> (accessed on March 15th, 2012)⁵<http://www.eclipse.org/m2m/at1/at1Transformations/#KM32EMF> (accessed on March 15th, 2012)

An ATL transformation, an example of which is depicted in Listing 3.7, consists of a **module**, which receives an input model `IN` and produces an output model `OUT`. Lines 3–4 of Listing 3.7 show a **module**, `Families2Person`, that receives a `Families` model and outputs a `Persons` model; the `Families` and `Persons` metamodels are declared in lines 1–2.

Listing 3.7: Example of an ATL transformation (excerpt from http://www.eclipse.org/m2m/at1/doc/ATLUseCase_Families2Persons.pdf).

```

1 -- @path Families=/Families2Persons/Families.ecore
2 -- @path Persons=/Families2Persons/Persons.ecore
3 module Families2Persons;
4 create OUT : Persons from IN : Families;
5 helper context Families!Member def: isFemale() : Boolean =
6   if not self.familyMother.ocllsUndefined() then true
7   else
8     if not self.familyDaughter.ocllsUndefined() then true
9     else false
10    endif
11   endif;
12 (...)
13 rule Member2Male {
14   from s : Families!Member (not s.isFemale())
15   to t : Persons!Male ( fullName <- s.firstName + ' ' + s.familyName )
16 }
```

A **module** includes a number of **rules**. A **rule** consists of a mapping between a pattern (a non-empty set of elements) from the `IN` model and a corresponding pattern for the `OUT` model. Like in QVTCore, each pattern can have a set of variables, which will be *bound* to each successful pattern match on the corresponding model. It is also possible to use *assignments* to fill the values on elements of the target `OUT` model. Lines 13–16 of Listing 3.7 provide an example of a **rule**, in which: (1) line 14 declares a variable `s` that will be bound to any `Member` elements of metamodel `Families` that has its `isFemale` function returning `false`; and (2) line 15 takes a source `Member`'s `firstName` and `familyName`, concatenates them, and assigns the resulting string to the target `Male`'s `fullName`.

Furthermore, a **module** can also include **helpers**. An **helper** is a function that computes some result that will be necessary for the evaluation of a **rule**. Lines 5–11 of Listing 3.7 illustrate a **helper** that defines a auxiliary function, for `Member`, to determine whether the family member is female.

Although we will not delve deeper into ATL's details, interested readers may find additional information at the project's website, which also includes some illustrative examples⁶.

⁶Namely, a more detailed description of the transformation in Listing 3.7 can be found at http://www.eclipse.org/m2m/at1/doc/ATLUseCase_Families2Persons.pdf and http://wiki.eclipse.org/ATL/Tutorials_-_Create_a_simple_ATL_transformation.

3.2 Graph-based Transformations

It should be noted that there are many similarities between the topics of model transformation and *graph transformations*, especially because any model that can be represented using a formal language (i.e., a language which establishes well-defined rules that constrain the set of allowed symbols and their arrangement into words or models) can also be represented as a graph [Dis 05_a].

In particular, we highlight the usage of *graph grammars* as a means to produce a new graph from another (original) graph [Roz 97, EEKR 99]. A **graph grammar** (also called *graph rewriting* in literature) consists of a set of rules in the form $O \rightarrow R$. These rules, when successfully matched against a graph G (i.e., when a part of G can be matched to the O part of a rule), will produce another graph G' , in which O has been replaced by R .

The work described in [MWCS 11, MV 11] also takes advantage of this strict relationship between models and graphs to propose an approach that uses graph grammars as the drivers to support the evolution of metamodels (and the corresponding updates to models), according to the rationale that was provided at the beginning of this chapter. More specifically, this approach considers that any *breaking operation* which occurs over a metamodel MM should result in a corresponding graph grammar that will be applied over models conforming to MM . The result, then, would be that a model M would always be conforming to its metamodel MM , even when MM was changed.

Finally, [Syr 11] also defines a model transformation framework that is based on the representation of models and model transformations (which are themselves models) as graphs. An interesting outcome of this work is that, because of its usage of models and model transformations in the same manner, it also enables the definition of an HOT (Higher Order Transformation), which is defined as a model transformation whose input and/or output models are themselves model transformations.

3.3 Transformations in Traditional Software Development

Following the same rationale of Chapter 2's comparison between the design of traditional programming and modeling languages, it makes sense to also consider topics and techniques regarding traditional software development, such as the usage of SQL scripts or Object-Relational Mapping (ORM) frameworks. The analyzed techniques, which are related to the issue of handling the change and evolution of software systems, are relevant because they may be adaptable for usage in MDE-oriented approaches and modeling languages.

3.3.1 SQL Scripts

A frequent example of model transformations is the definition and usage of SQL (Standard Query Language) scripts [RG 02]. SQL, as was mentioned in Section 2.3, is a standard textual language used to access and manipulate relational databases (i.e., repositories of data structured according to the relational model). A database can be considered as a set of tables, which can be considered as corresponding to domain entities. A table, in turn, consists of a set of columns (entity attributes) and rows (entity instances) [RG 02].

A SQL script is a set of SQL statements that are used to perform queries and other changes in a database. The SQL standard defines a set of commands that can be used to querying and manipulate data (i.e., table rows), namely [RG 02]: (1) `select`, for obtaining a set of rows that obey the provided constraints; (2) `update`, for *changing* data in one or more rows of a table; (3) `insert`, to *add* new data to a table; and (4) `delete`, to remove rows from a table. There are also a number of commands for manipulating the database's definition itself, such as: (1) `create table`, to create a table with a set of columns and constraints; (2) `alter table`, to change the table's structure, by means of `drop column` to remove the column or `add` to insert a new column; (3) `drop table`, to remove the table's rows *and* its definition; and (4) `truncate table`, to remove all rows from the table while maintaining the table's definition.

SQL also defines the concept of *transaction*, which is paramount to ensuring the ACID (Atomicity, Consistency, Isolation, and Durability) properties [RG 02] that are required of a reliable DBMS (Database Management System). The available commands for dealing with transactions are: (1) `start transaction`, signaling the beginning of a new transaction; (2) `commit`, used to attempt to successful finish the current transaction; and (3) `rollback`, to signal that the current transaction, and its changes, should be discarded.

It is also possible to define constraints over each table column. SQL considers a limited set of constraint, such as: (1) `foreign key`, specifying that the values of a column C_{T_1} should be present in another column C_{T_2} (in other words, C_{T_1} should reference C_{T_2}); (2) `unique key`, which specifies that each row should have a value (in that column) not present in other rows; (3) `primary key`, which is a refinement of `unique key`, and specifies that the column can be used as a key for other tables (via the `foreign key` constraint); (4) `default` value, specifying a default value for the column, which is only used if no value is provided when `inserting` the row; and (5) `null` and `not null`, which specify whether or not the column should accept NULL values, respectively.

Listing 3.8 provides a simple example of a SQL script, in which (1) a table `Folders` is created with three attributes (`FolderID`, `Name`, and `ParentFolderID`), (2) the `FolderID` column is designated as the `primary key` for the `Folders` table, and (3) a new row (with its `Name` being `Root Folder`) is inserted into this new table.

Listing 3.8: Example of a SQL script.

```
1 CREATE TABLE Folders (  
2   FolderID INTEGER NOT NULL,  
3   Name NATIONAL CHAR VARYING(255) NOT NULL,  
4   ParentFolderID INTEGER,  
5   PRIMARY KEY (FolderID)  
6 )  
7  
8 INSERT INTO Folders (FolderID, Name, ParentFolderID)  
9 VALUES (0, 'Root_Folder', NULL)
```

Finally, most DBMSs support the definition of *triggers*. A **trigger** is a set of procedural SQL statements that are executed whenever a specific event takes place in the database. Such events can be data modifications – row insertions, updates, or removals –, which makes triggers an adequate mechanism for issues such as (1) auditing (i.e., maintaining a log of the various operations performed, as well as their arguments), (2) intercepting an operation (e.g., to add a timestamp), or (3) upholding a set of semantic constraints. Listing 3.9 provides a small example of a database trigger that is executed just before a row is inserted into the `Articles` table. The trigger performs the simple task of inserting a corresponding row (with the `Hits` column’s value set to 0) into the `Articles_Hits` table, if the `ArticleID` is provided and if such a row does not already exist.

Listing 3.9: Example of a SQL trigger.

```
1 CREATE TRIGGER IF NOT EXISTS CreateArticleHitEntry  
2 BEFORE INSERT ON Articles  
3 WHEN  
4 NEW.ArticleID IS NOT NULL AND  
5 NOT EXISTS (SELECT ArticleID FROM Articles_Hits WHERE  
6   ArticleID = NEW.ArticleID AND ModuleID = NEW.ModuleID AND  
7   LanguageID = NEW.LanguageID)  
8 BEGIN  
9   INSERT INTO Articles_Hits (ModuleID, ArticleID, LanguageID, Hits)  
10  VALUES (NEW.ModuleID, NEW.ArticleID, NEW.LanguageID, 0);  
11 END
```

SQL scripts are relevant to the topic of model-to-model transformations because manipulation of the database’s definition can be considered as equivalent to *changing a metamodel*. Likewise, data manipulation (i.e., row inserting, deletion, or updating) can be considered as changing a model.

Furthermore, database-supported applications (e.g., applications that use a small SQLite database⁷ as their data store) often provide an upgrade path that, among other

⁷<http://www.sqlite.org> (accessed on March 15th, 2012)

steps, also uses a SQL script to upgrade the database's definition *while* maintaining consistency of the existing rows (i.e., its metamodel and model, respectively).

3.3.2 Object-Relational Mapping and Migration Frameworks

The definition of SQL scripts is an efficient solution for the problem of manipulating the database's definition – or its mapping to MDE, the metamodel – and ensuring that existing data (the model) is kept consistent. Nevertheless, this approach presents an additional problem, as different DBMSs usually have differences in the supported SQL dialect(s). Typical examples of such differences are that (1) Microsoft's SQL Server supports the Transact-SQL dialect, while PostgreSQL and Oracle's databases support the PL-SQL dialect, or that (2) PostgreSQL supports the automatic increment of a primary key's value, while Oracle requires that the database designer explicitly define a sequence to provide such values. Such differences present significant difficulties when developing applications that support various databases, like some CMS systems.

To address this issue, developers sometimes use the Strategy design pattern [GHJV 95] (or similar) to decouple the application's logic from its data access operations; WebComfort⁸ [SS 08_b] uses this approach to provide support for the Microsoft SQL Server, PostgreSQL, and Oracle DBMSs. Another tactic is the dynamic rewriting of the SQL scripts that are used when accessing the database; this tactic is used by Drupal⁹ [BBH⁺ 08], which only issues MySQL statements, to enable support for databases like Microsoft SQL Server (although, at the time of the writing of this dissertation, Drupal's developers do not recommend the usage of this feature). All of these strategies, although solving the problem, lead to a loss of productivity when developing applications with support for various kinds of databases. This is because the same kind of work – establish mappings between source code (or domain concepts) and the relational model, namely the SQL dialect for each supported DBMS – must be performed (either manually or by using template-based data access code generators) by developers whenever any changes to the domain occur.

A strategy that effectively addresses the mismatch between object-oriented programming language concepts and the relational model is the use of an **Object-Relational Mapping (ORM) framework**. Each ORM framework presents its own tactic to (as the name suggests) establish a mapping between a certain language's – or framework – programming concepts and the relational model. A brief list of examples of ORM frameworks (and their respective tactics) follows:

⁸<http://www.siquant.pt/WebComfort> (accessed on March 15th, 2012)

⁹<http://drupal.org> (accessed on March 15th, 2012)

- The ActiveRecord library¹⁰ of Ruby on Rails (RoR)¹¹ uses the ActiveRecord architectural pattern [Fow 03] to define Ruby classes that map to SQL tables. Developers are not required to specify SQL statements: they only have to specify Ruby classes (which must inherit from a predefined class provided by ActiveRecord, `ActiveRecord::Base`), and the library will handle the mapping to SQL. This tactic is also followed by Django¹², in which Python [Lut 09] classes inherit from the `django.db.models.Model` class¹³;
- Hibernate¹⁴ and NHibernate¹⁵ allow developers to map Java and .NET classes (respectively) to SQL tables. These frameworks require that developers either (1) create an *XML mapping file* that defines the database schema and the classes that they correspond to, or (2) decorate their source code classes with annotations that will guide the mapping procedure;
- Microsoft ADO.NET Entity Framework (EF)¹⁶ also contemplates a number of mapping approaches: (1) in the *Database First* approach, a database *D* is created first, and then an *Entity Data Model* is automatically generated from *D*'s schema; (2) in the *Model First* approach, the Entity Data Model is specified first, and afterward the database and source code classes are generated; and (3) in the *Code First* approach, the source code classes and a special `DbContext` class are defined first, and the mapping to SQL is automatically performed by EF (this approach relies on the usage of a set of annotations, as well as the developer's adherence to a set of naming conventions).

However, some of these ORM frameworks do not provide a mechanism to handle change to the database's *schema* itself. Frameworks such as Django or Microsoft's Entity Framework require that developers make manual changes to both the database and the source code or mapping files. On the other hand, frameworks such as Hibernate do consider this issue, although in a limited manner¹⁷, as its schema upgrade mechanism is based on the definition of increments to the database's schema; this mechanism only allows the *addition* of tables and columns, but not their *removal*.

Another strategy to address this schema change issue is the use of *Migrations* (which, in turn, are typically based on the ActiveRecord pattern). **Migrations** consist of com-

¹⁰<http://rubyforge.org/projects/activerecord> (accessed on March 15th, 2012)

¹¹<http://rubyonrails.org> (accessed on March 15th, 2012)

¹²<http://www.djangoproject.com> (accessed on March 15th, 2012)

¹³“Chapter 5: Models”, *The Django Book*, <<http://www.djangobook.com/en/2.0/chapter05>> (accessed on March 15th, 2012)

¹⁴<http://www.hibernate.org> (accessed on March 15th, 2012)

¹⁵<http://nhforge.org> (accessed on March 15th, 2012)

¹⁶<http://msdn.microsoft.com/ef> (accessed on March 15th, 2012)

¹⁷Daniel Ostermeier, “Incremental schema upgrades using Hibernate”, <<http://www.alittlemadness.com/2006/08/28/incremental-schema-upgrades-using-hibernate>> (accessed on March 15th, 2012)

binning ActiveRecord with the Command design pattern [GHJV 95], and are supported by frameworks such as Ruby on Rails' ActiveRecord (via the `ActiveRecord::Migration` class), PyMigrate¹⁸ for Python, or Microsoft Entity Framework's Migration mechanism (which is still under development as of the writing of this dissertation).

The definition of a Migration in the aforementioned ORM frameworks typically involves: (1) considering the database's definition as a set of classes (via the ActiveRecord pattern or similar); (2) applying the Command design pattern [GHJV 95] to define *database transformation unit* classes, which provide *Execute* and *Undo* methods. These methods are specified in normal source code, which takes advantage of the classes provided by the ActiveRecord implementation. Such implementations typically provide a set of operations over a database, namely to edit its definition (by creating, editing, or removing tables and columns) or even to change the data contained in the database.

Listing 3.10 provides a very simple example of a Migration for the Ruby on Rails framework. More specifically, this Migration adds a boolean column, named `ssl_enabled` and with default value `true`, to the `accounts` table. The Undo method (`down`) simply removes the added column.

Listing 3.10: Example of a Ruby on Rails Migration (extracted from <http://api.rubyonrails.org/classes/ActiveRecord/Migration.html>).

```

1 class AddSsl < ActiveRecord::Migration
2   def up
3     add_column :accounts, :ssl_enabled, :boolean, :default => 1
4   end
5
6   def down
7     remove_column :accounts, :ssl_enabled
8   end
9 end

```

Finally, because of their nature, Migrations that involve loss of data (such as the removal of a column) cannot provide an Undo method (`down`, in the case of Ruby on Rails, see line 6 of Listing 3.10). Such Migrations typically just return an error (or throw an exception, depending on the framework used), signaling that undoing changes is impossible.

3.3.3 Source Code Management

As we have previously mentioned in Section 2.5, source code itself can be considered as a model, in which the modeling language provides a textual concrete syntax. Following this same reasoning, it makes sense to look at how typical software development processes handle the task of *transforming source code*, in order to maintaining a code base consistent

¹⁸<http://code.google.com/p/pymigrate> (accessed on March 15th, 2012)

among the various workstations of a development team (or, in other words, how to ensure that any changes to the source code in workstation W_A are reflected in the other workstations $W_{B,C,D,\dots}$, so that developers do not work on outdated versions of the code).

Some source code management techniques exist, depending on the developer's ingenuity and ranging from the typical (but disaster-prone) zip-file storage to Revision Control Systems (RCS) or even more complex systems like Microsoft's Team Foundation Server (TFS)¹⁹. In this dissertation, we will only analyze the *modus operandi* of RCS techniques, because presenting an analysis of the other techniques mentioned would not be relevant for the scope of our work. This is because the complexity of many TFS-like systems derives not from source code management (and the inherent model transformations) but from the additional features (e.g., reporting) provided by the system, and zip-file storage would be equivalent to storing whole snapshots of a model, without involving any kind of transformation or synchronization (apart from the compression process itself).

There are a number of revision control systems – also called source control management or version control systems – available nowadays, such as CVS (Concurrent Versions System)²⁰, SVN (Subversion)²¹, Mercurial²², Git²³, or Bazaar²⁴. These systems typically present a number of differences from each other, such as (1) whether it is a distributed or a centralized RCS, or (2) whether it uses an underlying database system (such as SQLite or BerkeleyDB), or just the file system, to store code. Regardless of the number of RCS systems, they basically operate in the same manner, which is depicted in Figure 3.6.

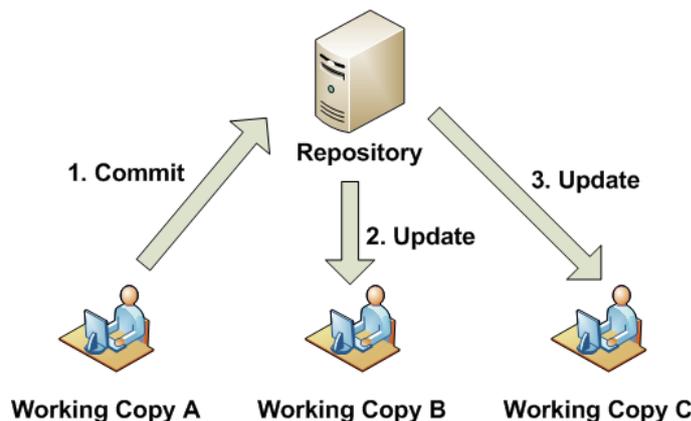


Figure 3.6: Overview of the operation of an RCS system.

¹⁹<http://www.microsoft.com/visualstudio/en-us/products/2010-editions/team-foundation-server> (accessed on March 15th, 2012)

²⁰<http://cvs.nongnu.org> (accessed on March 15th, 2012)

²¹<http://subversion.tigris.org> (accessed on March 15th, 2012)

²²<http://mercurial.selenic.com> (accessed on March 15th, 2012)

²³<http://git-scm.com> (accessed on March 15th, 2012)

²⁴<http://bazaar.canonical.com> (accessed on March 15th, 2012)

A **repository** is a program, running in a server, that is responsible for recording and propagating the various changes to the source code, namely by (1) storing the source code files – and other supporting artifacts, if necessary – in a persistent manner, and (2) receiving requests to either a) store changes to the source code or b) transmit any changes that have occurred since a specified point in time. It should be noted that in distributed RCS systems there is no centralized server, as each working copy is also a full-fledged repository.

Each set of changes that is stored in the repository is called a **revision** and, depending on the RCS system, it can be referenced by a simple number (assigned in a sequential manner) or another kind of unique identifier. Thus, it is possible to request a repository's state for a particular revision, instead of being forced to specify a point in time.

On the other hand, developers can duplicate a repository's state to their own workstations, by creating a **working copy** that reflects a certain revision (typically the most recent). This working copy will then be subject to a number of changes (e.g., adding or removing files, editing file contents) that are made by the developer. Nevertheless, the developer is allowed to *update* the working copy to the most recent state at any time, thus keeping all developers synchronized with each other. Finally, when the developer is ready to share these changes with the rest of the development team, they are *committed* to the repository (operation 1 in Figure 3.6), from which the other developers can then update their own working copies with the latest changes (operations 2 and 3 of Figure 3.6).

RCS systems typically store a file – and subsequent changes – as a set of **deltas** (also called *diffs*); in other words, they do not store each revision of the file itself, but rather the differences between each revision. Although the implementation of this mechanism varies from system to system, it can be considered that each delta stores: (1) the location in the file where the change takes place; (2) the previous state (i.e., the old version) of the file's contents in that location; and (3) the new state that replaces the previous state.

Each of these parts has an underlying rationale. The *location* of the change is used to identify (within the file) where to perform the replacement, and storing a tuple <line, column> is usually considered sufficient. The *new state* indicates what to place within the file contents, in the specified location. On the other hand, the *previous state*'s purpose is twofold: (1) it contains the set of characters to remove from the file's contents (e.g., so that a large string can be replaced with a smaller string); and (2) it enables checking whether the file's contents are consistent with the change to take place.

This last consistency check is used to detect situations in which:

1. A developer D_A makes changes to a working copy W_A , with revision R_1 , of a repository S ;

2. Meanwhile, another developer D_B also makes changes to a different working copy W_B , also in revision R_1 , of S ;
3. D_B successfully commits W_B 's changes, thus creating a new revision, R_2 ;
4. Afterward, D_A tries to commit W_A 's changes.

RCS systems typically deal with this scenario by checking the region where each change takes place. If the changes in W_A take place over different regions of the file (and thus do not overlap with W_B 's changes), the new changes are stored and a new revision R_3 is created. Otherwise, if the changes in W_A do overlap with W_B 's, then a *conflict* is detected and it will be up to D_A to solve it, by *manually merging* W_A 's changes with the most recent revision, R_2 . The possibility of catching this kind of problems makes this check extremely important in collaborative software development, as it becomes very unlikely that changes can accidentally overwrite one another.

3.3.4 Code Compilation

Considering that source code can be considered as a model, code generation – typically considered as a model-to-text transformation – can be considered a model-to-model transformation with a textual destination language. Figure 3.7 presents an example of code generation as a model transformation, in which a very simple UML model is converted to the corresponding Python skeleton code.

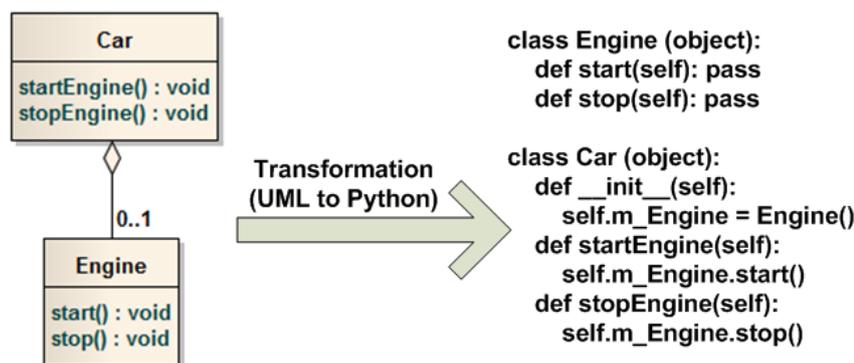


Figure 3.7: Example of code generation as a model-to-model transformation.

Still following the same rationale, even the process of compiling source code to assembly language – or to executable machine code – can be regarded as a case of a model-to-model transformation, in which the source is a textual modeling language and the destination is a binary language (or bytecode, in the case of Microsoft's .NET Framework, Oracle's Java, Python, Ruby, or any language that is interpreted at runtime).

3.4 XSLT

A very frequent usage of a model transformation is the transformation of XML documents, from an XML schema Sch_A to another schema Sch_B . A possible way to define such a transformation is to create a program that (1) receives an XML document (compliant with Sch_A), (2) parses it using a programming mechanism like SAX (Simple API for XML)²⁵ or DOM (Document Object Model)²⁶, and (3) produces a new XML document that complies with Sch_B . This approach was used in the first version of the XIS modeling approach [Sil 03, SLMC 03], which used XMI to convert UML models (created with the Rational Rose modeling tool) to a simpler XML format, which was then processed by a source code generation tool.

However, there is another technique available that can avoid the added work of defining such a program: **XSLT**, which stands for “*XSL Transformations*” (XSL, in turn, meaning “eXtensible Stylesheet Language”). XSLT is a declarative transformation language in which a developer specifies a set of XML nodes, called **templates**. Each **template**, in turn, will be matched with XML nodes present in the input document; this match is determined by a pattern defined in XPath²⁷. Each **template** also defines its output within the XML element’s body, as a set of XSLT nodes, which can do a wide variety of tasks, like (1) sorting lists, (2) delegating the transformation to another XSLT node, (3) testing for specific conditions, (4) producing the value returned by an XPath expression, (5) declaring new variables, (6) producing XML nodes, if the output is meant to be in XML format, or (7) producing plain text. It should be noted that XSLT transformations do not necessarily output XML documents. These transformations are allowed to output any textual format, such as HTML, XHTML, or even plain text.

Listing 3.11 depicts an example of an XSLT transformation. This transformation (adapted from the WebC-Docs document management system [SS 09_b, SS 11_a]) takes an XML document, containing the description of a set of documents with information such as their names and authors, and outputs a XHTML document to be viewed in a web browser.

Listing 3.11 also illustrates that XSLT transformations can receive *arguments* provided by their calling program. Line 3 illustrates the declaration of an XSLT argument, called **MaxAuthors**, while Line 12 shows the usage of that same argument in a verification to check whether the number of authors has exceeded the value of **MaxAuthors** (and replacing any additional authors with the string “et al.”).

Establishing a connection to MDE-related transformations (which were presented in Section 3.1), we can see that XSLT can be regarded as conceptually similar to QVT

²⁵<http://www.saxproject.org> (accessed on March 15th, 2012)

²⁶<http://www.w3.org/DOM> (accessed on March 15th, 2012)

²⁷<http://www.w3.org/TR/xpath> (accessed on March 15th, 2012)

Listing 3.11: Example of an XSLT transformation.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:fn="http://www.w3.org/2005/02/xpath-functions" >
3   <xsl:param name="MaxAuthors" ></xsl:param>
4   <xsl:template match="document[@name!='']" >
5     <xsl:element name="div" >
6       <xsl:value-of select="@name" disable-output-escaping="yes" />
7       <xsl:variable name="numberOfAuthors" select="count(authors/author[@name!='' and @order_
        !='])_&gt;_0" ></xsl:variable>
8       <xsl:if test="$numberOfAuthors &gt; 0" >
9         <xsl:for-each select="authors/author[@name!='' and @order!='']" >
10          <xsl:sort select="@order" data-type="number" order="ascending" />
11          <xsl:choose>
12            <xsl:when test="position()_&lt;=_$MaxAuthors" >
13              <xsl:value-of select="@name" disable-output-escaping="yes" />
14              <xsl:if test="position()_!=_last()" >
15                <xsl:text>,</xsl:text>
16              </xsl:if>
17            </xsl:when>
18            <xsl:when test="position()_=_$MaxAuthors + 1" >
19              <xsl:text> et al.</xsl:text>
20            </xsl:when>
21          </xsl:choose>
22        </xsl:for-each>
23      </xsl:if>
24    </xsl:element>
25  </xsl:template>
26  <xsl:template match="/" >
27    <xsl:element name="table" >
28      <xsl:for-each select="documents/document" >
29        <xsl:apply-templates select="." />
30      </xsl:for-each>
31    </xsl:element>
32  </xsl:template>
33 </xsl:stylesheet>

```

(although the technologies and languages used are different): both declare a set of *transformation units*, which establish (1) a pattern to be recognized, and (2) a corresponding output pattern that can use information obtained from the matched pattern (e.g., a class name).

Summary

In this chapter, we have presented the context for the usage of model transformations in MDE, and some model transformation languages that are of particular relevance for the work presented in this dissertation. In particular, we have presented the OMG's QVT and MOFM2T transformation languages, as well as ATL, the basis for Eclipse's M2M project.

We have also analyzed a set of traditional software development techniques which address some problems that MDE-oriented model transformation languages try to solve, namely ensuring that two models (e.g., source code artifacts) are semantically equivalent. Nevertheless, the analyzed software development techniques also address problems that the analyzed model transformation languages do not, such as (1) the assurance that multiple sets of concurrent changes to the same model do not accidentally overwrite each other (and thus information is not lost), or (2) handling changes not only to a model but also to a *metamodel*.

Figure 3.8 provides a simple mindmap overview of this analysis.

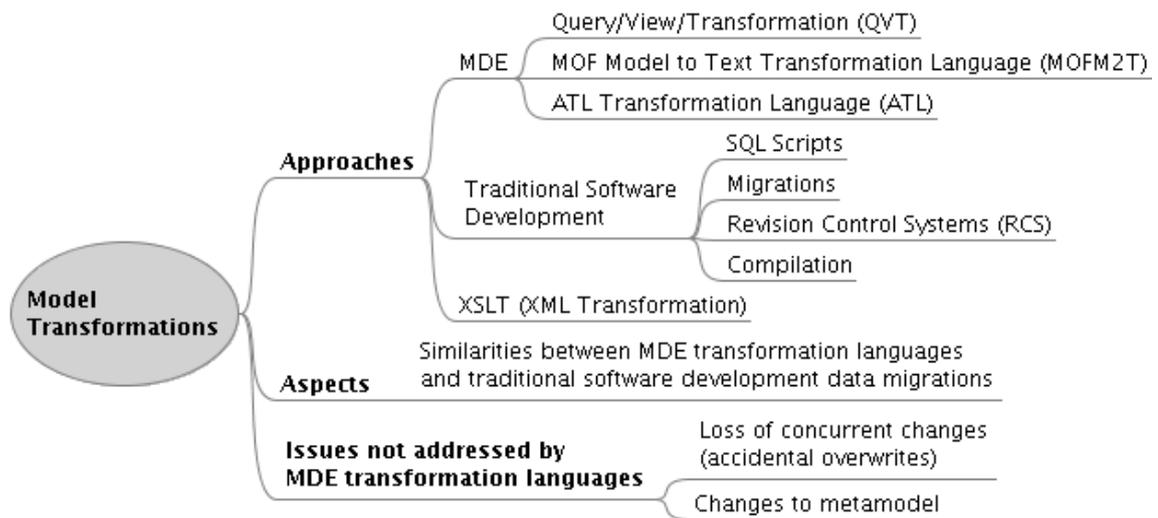


Figure 3.8: Overview of aspects and problems regarding model transformations.

In the next chapter, we narrow the focus on this presentation of the relevant State of the Art, and present a set of *web application modeling languages*. These languages, each following the MDE principles (namely the usage of models as the most important artifacts in software development) to some extent, allow their users to model *web applications* (i.e., applications that operate by using web-based concepts) for various purposes.

Chapter 4

Model-Driven Approaches for Web Application Development

Design and programming are human activities; forget that and all is lost.

The C++ Programming Language

BJARNE STROUSTRUP

Web application development, combined with new perspectives on to scale and distribute the application (such as Software-as-a-Service) to as many users as possible, has become the default way to create new applications (or to update old ones). Currently there are some model-driven approaches for web application development. However, the development approach is typically driven by the expressiveness of the modeling language that is used in that approach.

In this chapter, we analyze the following MDE approaches for web application development and their supporting languages:

1. WebML¹ [BCFM 07, CFB⁺ 02];
2. UWE² [KK 08];
3. XIS2 [SSSM 07];
4. Microsoft Sketchflow³ [1532012]; and
5. The OutSystems Agile Platform⁴ [OutSystems 10].

The WebML, UWE, and OutSystems Agile Platform approaches were selected for analysis in this dissertation (instead of other possible MDE approaches) because they are well-

¹<http://www.webml.org> (accessed on March 15th, 2012)

²<http://uwe.pst.ifi.lmu.de> (accessed on March 15th, 2012)

³http://www.microsoft.com/expression/products/Sketchflow_Overview.aspx (accessed on March 15th, 2012)

⁴<http://www.outsystems.com> (accessed on March 15th, 2012)

-known in the web application development field (from what we could tell during our search for related work). XIS2 is also covered in this analysis because it was developed during the course of our previous work in ProjectIT-Studio [SSF⁺ 07], which dealt with model-driven application development and code generation. Finally, although Microsoft Sketchflow is not an MDE approach or language *per se*, and it is not specifically oriented at web applications, we include it in this analysis because it can be used to generate cross-platform interactive web applications, and because it addresses a very important aspect not considered in other approaches: providing a stakeholder-friendly vision of the application design.

4.1 Comparative Analysis

In this section, we present our analysis of these web application development approaches: (1) the analysis criteria are introduced first, namely by describing the aspects under scrutiny; and (2) the results of this analysis are presented afterward, as well as the rationale for the various obtained values. This analysis is further detailed in Appendix A.

4.1.1 Analysis Criteria

This analysis is mostly focused on the modeling language used by the approach and whether it addresses a set of modeling concerns, such as domain modeling, business logic modeling, and user interface modeling. Nevertheless, we also analyze some aspects regarding the generation of parts of the web application, such as the usage of model-to-model transformations or the considered deployment environment(s). These characteristics have been adapted from [SS 10_a], which defines a reference model for MDE-oriented web application languages that has resulted from our analysis.

A – Domain modeling. Domain modeling concerns the identification of problem-domain concepts, and their representation using a modeling language (e.g., UML diagrams). This aspect is analyzed regarding: (1) whether it is supported by the language; and (2) whether it is independent from persistence and user interface details (i.e., domain models do not need to be “tweaked” to support those layers).

B – Business Logic modeling. Although the definition of business logic modeling can be considered somewhat subjective, in this dissertation we consider it as the specification of the web application’s behavior. This aspect is analyzed regarding the following subjects: (1) whether it supports querying and manipulating domain concepts (namely using patterns); (2) whether this querying and manipulation support is low-level, in a manner

similar to traditional source code; and (3) support for process specification. It should be noted that the subject of low-level support is considered relevant because it often reflects the expressiveness of the language: typical source code-oriented languages (such as C or Java), although complex, are nevertheless very expressive.

C – Navigation Flow modeling. The approach’s support for specifying the navigation flow (in the context of the modeled web application) between different HTML pages, or even inside HTML pages, is also an aspect under analysis.

D – User Interface modeling. Another important aspect is the approach’s support for modeling the user interface (UI). The analyzed subjects are the following: (1) whether the UI modeling language is platform-independent (i.e., does not require specific software to present the UI); (2) supports access control specification (i.e., certain controls are shown or hidden according to the authenticated user); (3) allows the definition of custom UI elements; (4) allows the usage of interaction patterns (e.g., create, edit, or associate and dissociate); and (5) supports binding between UI elements and domain model elements.

E – Model-to-model transformations. This aspect analyzes whether the approach uses (or even considers) the usage of model-to-model transformations. This kind of transformations is typically used to accelerate the task of designing the web application, by using model analysis and inference mechanisms to automatically specify some parts of the web application model, thus releasing the model designer from some repetitive (and error-prone) tasks.

F – Generated application is complete. This aspect determines if the approach’s tool support (at the time of writing of this dissertation) is able to *completely* generate the application (i.e., it does not require the manual implementation of specific features by programmers).

G – Independent from deployment environment. Finally, this aspect studies the target platform(s) considered by the approach, namely whether there is a tight coupling between the approach and the target platform.

4.1.2 Analysis Results

The analysis of these web application development approaches, according to the analysis criteria listed in the previous subsection, has yielded the results that are presented in Table 4.1.

Table 4.1: Relevant characteristics of the analyzed MDE approaches and languages.

	Web- ML	UWE	XIS2	Out- Systems	Sketch- flow
A. Domain modeling	✓	✓	✓	✓	✗
<i>Independent from persistence</i>	✓	✓	✓	✗	—
<i>Independent from UI</i>	✓	✗	✓	✓	—
B. Business Logic modeling	✓	✓	✓	✓	✓
<i>Domain manipulation using patterns</i>	✓	✗	✓	✓	✗
<i>Custom patterns</i>	✓	—	✓	✗	—
<i>Low-level specifications</i>	✓	✓	✗	✓	✓
<i>Domain query</i>	✓	✓	—	✓	✓
<i>Domain manipulation</i>	✓	✗	—	✓	✗
<i>Process specification</i>	✗	✓	—	✓	✗
C. Navigation Flow modeling	✓	✓	✓	✓	✓
D. User Interface modeling	✓	✓	✓	✓	✓
<i>Access control specification</i>	✓	✗	✓	✓	✗
<i>Custom interface elements</i>	✓	✗	✗	✓	✓
<i>Interaction patterns</i>	✓	✗	✓	✓	✓
<i>Custom interaction patterns</i>	✗	—	✗	✗	✓
<i>UI elements bound to domain elements</i>	✓	✓	✓	✓	✓
<i>Bindings are customizable</i>	✗	✗	✓	✓	✗
E. Model-to-model transformations	✗	✓	✓	✗	✗
F. Generated application is complete	✗	✗	✗	✓	✗
G. Independent from deployment environment	✓	✓	✓	✗	✓

A – Domain modeling. Domain modeling is an essential aspect in any application development approach. The analyzed approaches enable independence between the domain model and other parts of the application design, but only WebML and XIS2 address domain modeling in a manner that is completely independent of the remaining application details, by not requiring that the domain model be adjusted with UI- or persistence-oriented details. In UWE’s case, the Content Model is supposed to be somewhat oriented toward the Presentation Model (e.g., in the Address Book example at the UWE tutorial⁵, the `AddressBook` class has an attribute `introduction`, used for the main screen of the application

⁵<http://uwe.pst.ifi.lmu.de/teachingTutorial> (accessed on March 15th, 2012)

to show a small introductory message to the user). In the Agile Platform, the domain model is modeled like a database schema, although with different terms (**Entities** and **Attributes** instead of tables and columns); although this lack of abstraction may be regarded as a bad thing, in practice it does endow the designer with a greater level of control over the web application's real database schema. WebML also features a way to adjust the domain model to particular needs of other layers, by means of the Derivation Model, and XIS2 provides the BusinessEntities View to address the manipulation of domain entities by other layers.

B – Business Logic modeling. Business logic modeling is addressed by all of the analyzed approaches, albeit in very different ways. XIS2 considers this aspect by means of **Business-Entities** – allowing other layers to manipulate elements that are coarser-grained than domain elements – and patterns (the typical create/read/update/delete operations are supported, and the designer can add more operations, although they must be implemented in a manual fashion). WebML uses similar mechanisms (the Derivation Model for adjusting the domain model to other layers, **Content Units** and **Operation Units** to query and manipulate the domain, and the designer can add new **Operation Units** that must be implemented manually), but the designer can orchestrate them in a flowchart-like manner, enabling the specification of data manipulation workflows. UWE, however, only considers this aspect by means of the Process Model – class diagrams that represent the relationships between the various processes, and activity diagrams that specify the steps of each process – and OCL constraints; the implementation itself must be done manually. Sketchflow, albeit a prototype-creation tool, allows designers to imbue their prototypes with some behavior, by means of the **Behaviors** mechanism; it can also be extended with additional behavior patterns, but the creation of such functionality involves some manual development. Finally, the Agile Platform allows designers to specify complex business logic in a graphical manner: by defining **Actions** such as domain query and manipulation patterns, designers can specify most (if not all) of the functionality that would usually have to be manually coded.

C – Navigation Flow modeling. Navigation flow modeling is addressed by all of the analyzed approaches, because it is fundamental to any kind of web application. Due to their web application modeling nature and all that it implies (e.g., the existence of HTML pages, hyperlinks to navigate between pages, using request parameters or a similar mechanism to provide necessary parameters), they all follow the same guidelines (although the terms used are somewhat different): a directed graph is drawn, where nodes correspond to HTML pages, and edges correspond to the possible navigation flows that are available

to the user within the context of a certain page. Nevertheless, the analyzed approaches do differ in whether the designer can specify the sets of edges (i.e., actions or links) available in each node. In WebML, each `DataUnit` node has a well-defined set of links, for input and output, and the designer cannot specify additional links. On the other hand, XIS2 allows the designer to specify actions, associated to UI elements in a page, and the navigation flows will afterward be associated with those page actions; XIS2's approach to this aspect is very similar to what can be found in the Agile Platform, Sketchflow, and UWE.

D – User Interface modeling. Except for WebML, all of the analyzed approaches address UI modeling in a graphical WYSIWYG (What You See Is What You Get) manner. WebML only allows the specification of what elements will be present on the page, but not *where* they will be located. Also, the Agile Platform, WebML, and Sketchflow support the creation of new page or UI elements to be reused in other application screens. This allows designers to specify certain screen sections only once, and import those sections into some/all of the application's screens, with the obvious added advantage that changes to such a section need to be done only in a single point in the model; a good example of such a component would be a website banner, or a website navigation tree.

The behavior aspect is explicitly addressed by XIS2, WebML, and the Agile Platform through the capture of *UI patterns*, enabling applications to interact with users instead of just displaying requested resources. This focus on the capture of UI-oriented patterns is also present in Sketchflow, in the form of **Behaviors**. Sketchflow itself is the only tool, of those analyzed, that supports adding new interaction patterns in a reusable manner. Also, the specification of access control (i.e., which users or roles can access which UI components) is supported in all approaches, except for UWE and Sketchflow. Although in Sketchflow this shortcoming is understandable (considering its nature as a prototyping tool), in UWE this hampers its adequacy to model real-world scenarios.

All of the analyzed approaches allow binding UI elements to domain elements (e.g., configure the rows of a table to show the values of a field in specific instances). However, only XIS2 and the Agile Platform allow the customization of those bindings in the model (e.g., change a cell in a table row to show a different value for each instance). Although not being able to customize these bindings can simplify the model and avoid designer errors, in practice this is also a limitation of the language, which forces developers to change generated source code in order to address specific requirements.

E – Model-to-model transformations. Support for model-to-model transformations is an aspect that is usually not addressed by web application modeling approaches; from the analyzed approaches, only the UML-based ones (UWE and XIS2) address this aspect.

The two approaches use model-to-model transformations to accelerate modeling tasks, by taking the information already modeled at the time the transformation is performed and generating new views (called Models in UWE, and Views in XIS2). These transformations reflect how those views would typically be modeled, and the designer is free to change the generated views to suit the application's needs (e.g., showing an extra field in the UI, or adding an extra step in an activity diagram).

F – Generated application is complete. Of the analyzed approaches, only the Agile Platform considers that source code should not be edited manually: only the model itself can be edited, and generated code and databases are always kept out of the developer's reach. All the other approaches consider traditional programming (i.e., the manual editing of generated source code artifacts) as an activity to occur during the development life cycle, in order to account for particular requirements that may not be expressible by the approach's modeling language.

G – Independent from deployment environment. Except for the Agile Platform, all approaches are actually independent of the deployment environment. Because of their usage of high-level elements, WebML, UWE, and XIS2 can generate code for any web-oriented platform (e.g., ASP.NET, Apache Tomcat); XIS2 can also generate code for desktop-based platforms. The web-oriented version of Sketchflow generates a Silverlight application which, although requiring the Silverlight plugin installed on a web browser, can be considered as cross-platform because it is available for the Microsoft Windows, Linux, and Mac OS X platforms. Finally, models created in the OutSystems Agile Platform can only be deployed to OutSystems's deployment stacks, which use either (1) the JBoss application server and Oracle database, or (2) Microsoft's IIS application server and SQL Server Express database. Although in this aspect the Agile Platform is apparently more limited than the other approaches, this is what allows it to automate most of the web application development life cycle, namely deployment and application upgrade/rollback scenarios.

4.2 Additional Related Work

Besides the web modeling approaches analyzed in this section, we have also found other work regarding development of modeling languages for web-based systems.

The work presented in [WSSK 07] describes the authors' results in defining a metamodel that is common to both WebML and Object-Oriented Hypermedia (OO-H)⁶; the authors

⁶http://gplsi.dlsi.ua.es/iwad/ooH_project (accessed on March 15th, 2012)

plan to support UWE in the near future. The objective is to (1) define a set of model-to-model transformations that enable bidirectional transformations between the supported languages, and (2) ultimately define a web-oriented modeling language, which the authors designate as Unified Web Modeling Language, that unifies all of the contemplated web modeling languages. However, there are some semantic mismatches between these different metamodels. An immediate example is the definition of the user interface (based on typical HTML elements), which in UWE is performed in the Presentation Model, while in WebML such details are derived (when defining pages, the focus of WebML is on defining data manipulation workflows, and not the interface that will support those workflows). The authors do not specify how they plan to handle such mismatches, namely regarding the possible loss of information during transformations between models of different languages.

On the other hand, the work described in [Wri 09] also proposes a web modeling language, called Internet Application Modelling Language (IAML). However, instead of trying to define generic web concepts and adapting them to a set of specific contexts, the author uses RIA (Rich Internet Application) concepts to shape IAML into a RIA-oriented modeling language. In our perspective, the results of this work could be used to complement the results of our research work (namely the modeling languages that we defined, which are presented in Chapters 7–8), as the research domains are connected in the sense that they both deal with model-driven web application development. Nevertheless, such an integration would be beyond the scope of this dissertation, which is why we consider this only as possible future work.

Summary

Although currently most web applications are still developed in a manual fashion (i.e., through typical programming tasks), the concept of developing web applications in a model-driven manner is rapidly growing in popularity. Examples of this growth trend can be found in the number of web application modeling languages, such as those analyzed in this chapter and further detailed in Appendix A.

In this chapter, we have presented an analysis of a select set of web application-oriented modeling approaches and languages. This analysis, in turn, was focused primarily on aspects that are relevant to this kind of modeling languages. Figure 4.1 provides a simple mindmap overview of the analyzed aspects.

From this analysis, we have also extracted a set of problems (which will be presented in Chapter 6), as well as some considerations that should be taken into account when proposing a solution.

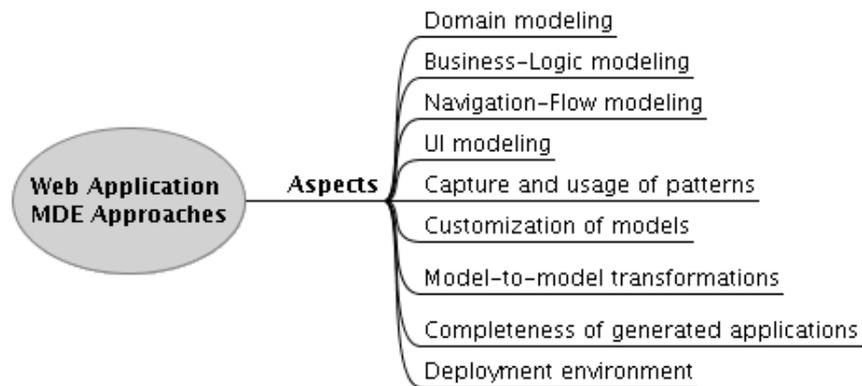


Figure 4.1: Overview of aspects and problems regarding web application modeling languages.

In the next chapter, we provide an analysis of Content Management System (CMS) systems, another important topic of this dissertation. These systems, besides providing their users (not only content administrators but also consumers) with a useful set of features like modularity or access management, also present some characteristics that make them suitable for the development of web applications, namely the extensible architecture with which they are usually built.

Chapter 5

Content Management Systems

A bad website is like a grumpy salesperson.

JAKOB NIELSEN

Although the idea of managing content has been around since the dawn of the Internet, it was only in the last years that we have begun witnessing an explosion of CMS systems [Boi 01, SATE 03]. A Content Management System (CMS) is a particular kind of web application that is oriented toward the management and publishing of content (which can be almost anything, like a blog post, a forum entry, some HTML text, or a video). These systems typically endow content administrators and consumers (i.e., regular users that just browse the website) with a set of relevant aspects such as (1) modularity, (2) independence between content and its presentation, (3) access management, (4) user control, or (5) configurable visual appearance and layout of content. Furthermore, development of CMS-based web applications is a topic that has only recently been proposed, as most CMS systems did not provide developers with an adequate set of features until a short time ago.

In this chapter we analyze the following CMS systems, which we consider relevant to our research work:

1. DotNetNuke¹ [HRW 09];
2. Drupal² [BBH⁺ 08];
3. Joomla³ [SC 09];
4. Vignette Content Management⁴; and
5. WebComfort⁵ [SS 08_b].

¹<http://www.dotnetnuke.com> (accessed on March 15th, 2012)

²<http://drupal.org> (accessed on March 15th, 2012)

³<http://www.joomla.org> (accessed on March 15th, 2012)

⁴<http://www.vignette.com> (accessed on March 15th, 2012)

⁵<http://www.siquant.pt/WebComfort> (accessed on March 15th, 2012)

Although there is a constantly increasing number of CMS systems available⁶, these were selected for analysis because of the significant differences that they exhibit among themselves. Other CMS systems (e.g., WordPress⁷, a very popular blogging system that is nevertheless considered by the community as a CMS system), although just as suitable for analysis as the ones previously mentioned, would yield very similar results – for example, an analysis of WordPress would result in nearly the same values as Joomla –, making their inclusion into this analysis redundant.

5.1 Comparative Analysis

In this section, we present our analysis of these CMS systems. Like in Chapter 4, we start by introducing the analysis criteria, and then proceed to the results that have been obtained from this analysis. Furthermore, the analysis of these CMS systems is further detailed in Appendix B.

5.1.1 Analysis Criteria

This analysis is mostly focused on characteristics obtained from (1) our own experiences with using these systems during our research, and (2) the knowledge acquired while developing the WebComfort CMS [SS 08_b] and some web applications that are supported by it, such as WebC-Docs [SS 09_b, SS 11_a]. More specifically, the criteria used covers aspects that range from administrative capabilities (such as configuration of the website’s structure or multi-tenancy support) to developer-oriented features like extensibility or providing a specific approach for the development of CMS-based web applications.

A – Management approach. CMS systems typically use one of two management approaches [Vignette 09]: page-centric and content-centric. A **page-centric approach** considers that the website’s *structure* (i.e., a set of pages and components) must be defined first, and afterward *content* is defined within the context of that structure. On the other hand, a **content-centric approach** dictates that the content itself must be defined, and afterward the administrator can specify a structure of pages that will show some of that content. This aspect analyzes which of these management approaches is used by the CMS.

B – Customizable website structure. This aspect determines whether the CMS system allows its administrator users to customize the website’s structure, in such a way

⁶An extensive list of existing CMS systems is available at <http://www.cmsmatrix.org> and <http://www.opensourcecms.com>.

⁷<http://wordpress.org> (accessed on March 15th, 2012)

that effectively allows visitors to perceive the website's organization as a (structured) hierarchical set of pages. Although this aspect might seem irrelevant (as it appears to favor CMS systems with a page-centric management approach), it is actually used to indicate whether the CMS supports the specification of a website structure; this structure, in turn, is often fundamental to help visitors when navigating the website and accessing published content.

C – Customizable visual layout of page. Besides the possibility of customizing the website's structure, administrators should also be able to customize the website's visual layout (i.e., the website's look-and-feel, such as the colors used, or the relative location of each container that pages will use to show content). This aspect determines whether the CMS supports any such visual mechanism.

D – Supports multi-tenancy. Multi-tenancy consists of whether it is possible for someone to create a logical set of websites within the same CMS installation. In other words, this aspect determines whether a *single physical CMS installation* can support the definition of *multiple logical websites* (e.g., a personal website and an eCommerce website), each of which is usually accessible via a different URL (Universal Resource Locator).

E – Multiple kinds of persistence. A CMS system, due to its dynamic nature, must persist (i.e., store) its information somewhere (e.g., a database, or even the file system). Although at first sight this aspect might only seem to be a technology-related detail, it is important to note that the importance of this aspect is derived from the decoupling of the web application logic and the persistence mechanism, which in turn introduces the notion that domain modeling in a CMS-oriented language should not depend on – or even assume – technology-specific details (such as database views).

F – Can be extended by third-parties. This aspect scrutinizes the mechanisms that are provided by the CMS system for its extension by third-parties (e.g., whether the CMS provides an API to develop new features). This analysis is particularly focused on the following items: (1) what languages (programming languages or otherwise) can be used; (2) whether it is possible to use the security features provided by the CMS to constrain possible actions (or show further information); (3) whether changing the CMS system's *default* behavior is allowed; and (4) if it is possible to add *new behavior* (e.g., new CMS components, or additional code to run when specific events occur) to the system.

G – Development approach. This final aspect analyzes the kind of approach (if any) that the CMS supports for the development of web applications based on it (even if the

web application consists just of customizations, extensions, or anything that changes the system's default behavior). More specifically, we determine: (1) if the CMS does consider any particular approach for the development of such web applications; and, if it does consider such an approach, (2) whether it is model-driven or if it is performed in a more traditional, source code-driven, manner.

5.1.2 Analysis Results

The analysis of these CMS systems, according to the previously listed criteria, has resulted in the values presented in Table 4.1.

Table 5.1: Relevant characteristics of the analyzed CMS systems.

	DotNet-Nuke	Drupal	Joomla	Vignette	Web-Comfort
A. <i>Management approach</i>	Page-centric	Content-centric	Content-centric	Content-centric	Page-centric
B. <i>Customizable website structure</i>	✓	✓	✓	✓	✓
C. <i>Customizable visual layout of page</i>	✓	✓	✓	✓	✓
D. <i>Supports multi-tenancy</i>	✓	✓	✗	✓	✗
E. <i>Multiple kinds of persistence</i>	✗	✓	✗	✗	✓
F. <i>Can be extended by third-parties</i>	✓	✓	✓	✓	✓
<i>Programming language(s)</i>	C#/VB.NET	PHP	PHP	Java	C#/VB.NET
<i>Provides security features</i>	✓	✓	✓	✓	✓
<i>Can change default behavior</i>	✓	✗	✗	✗	✓
<i>Can add new behavior</i>	✓	✓	✓	✓	✓
G. <i>Development approach</i>	Traditional	Traditional	Traditional	Traditional	Traditional

A – Management approach. Regarding the management approach that is used by the analyzed CMS systems, we notice that both approaches (page-centric and content-centric) are used. More specifically, the DotNetNuke and WebComfort systems use a page-centric approach, in which the website's structure (i.e., a set of tabs and modules) is defined first, and afterward content is specified in *modules*. Nevertheless, these two CMS systems

support the sharing of content between modules, via DotNetNuke's *module copy* feature and WebComfort's *module copy* and *module reference* features. On the other hand, the Drupal and Joomla systems use a content-centric approach, in which the administrator first defines the content that will be displayed to the user, and then defines a structure of pages that will show certain parts of the available content. The Vignette system itself can be considered as a mix of these two approaches, because content are defined independently and presented to the user by means of Vignette's templating mechanism, which uses a predetermined website structure to present the existing content. Nevertheless, we consider Vignette as being mainly content-centric, due to the fact that it places greater emphasis on the definition of content instead of on the website's structure definition.

B – Customizable website structure. All of the analyzed CMS systems allow administrators to customize the website's structure as a hierarchical set of pages or nodes (combined with linking mechanisms such as Joomla's menus [SC 09]), depending on the management approach used by the CMS.

C – Customizable visual layout of page. The ability to customize the website's visual layout (i.e., the website's look-and-feel, such as the colors used, or the relative location of each container that pages will use to show content) is supported by all of the analyzed CMS systems.

D – Supports multi-tenancy. Multi-tenancy is supported in some of the analyzed CMS systems. More specifically, only Joomla and WebComfort lack support for multi-tenancy, although in both cases this is an aspect which the developers are currently working on. Nevertheless, we have observed that the CMS systems that do support this feature typically do so by defining a different database table prefix for each website, which means that the various logical websites are often completely independent from each other.

E – Multiple kinds of persistence. Regarding the persistence mechanisms used, only some CMS systems (namely Drupal and WebComfort) support multiple kinds of persistence mechanisms. More specifically, this feature comes in the shape of support for different kinds of DBMS (Database Management System), such as MySQL, PostgreSQL, or Microsoft SQL Server.

F – Can be extended by third-parties. All of these CMS systems allow extension by third-party developers. The aspects that can be used or extended by developers vary between systems, although some are common, such as security features (user and role

management). However, as with the supported persistence mechanisms, the technology-specific details (such as the programming languages used) vary between different CMS systems, which further attests to the fact that CMS-oriented languages should be as technology-independent as possible. Also, all of these systems support adding features and behavior, but only some support *changing* the existing default behavior; in the latter case, this is typically done by using the Strategy design pattern [GHJV 95] (or a variant of it), and using the default built-in behavior only if no other strategy is available.

G – Development approach. None of the analyzed CMS systems consider an approach for customization or the development of extensions. Although, as mentioned in the previous paragraph, each of these systems provides some developer support – in varying forms, but typically consisting of an API for developing source code – the development approach itself is left for developers to determine, in an *ad hoc* manner.

5.2 Additional Related Work

Besides the CMS systems analyzed in this section, we have also found some recent work regarding development approaches for CMS-based web applications.

The work presented in [Car 06] defines a generic CMS metamodel that can be applied to most of the CMS systems that are available nowadays (including the ones analyzed in this chapter). Although this metamodel is highly focused on the structure of a CMS, we reuse parts of this metamodel in our own research work that is presented in this dissertation.

On the other hand, [Sch 08] describes the creation of a model interpreting web application (which the authors call Integration Generator) that interacts with the Limestone CMS⁸. The Integration Generator application receives an XML file (which, in turn, results from the XSLT transformation of a UWE model file, specified in the XMI format), and is responsible for configuring the target CMS system, namely by (1) interacting with the backing data store (such as a database server), and (2) generating the necessary support artifacts (e.g., ASP.NET pages and user controls).

Furthermore, [VVP 08] analyzes the suitability of the Object Oriented Web Solution (OOWS) method to the development of CMS-based web applications, and suggests some improvements derived from Situational Method Engineering. However, the approach only deals with high-level aspects, which leads to a lack of expressiveness when dealing with low-level details (such as specifying particular CSS classes).

⁸This CMS's address, <http://www.limestoneweb.co.uk/Solutions/ContentManagement>, was offline at the time of the writing of this dissertation.

Finally, the MDE approach presented in [SK 09] also aims to support business-users. This is done by defining a modeling language that addresses only high-level details (based on parts of other existing languages), and a conversion mechanism that takes a model and generates a *CMS Configuration File* (an XML file that will be interpreted by the CMS). This approach also suffers from some shortcomings of other MDE approaches, namely the lack of expressiveness to deal with low-level details; developers can change the generated CMS Configuration File using an XML editor, but this file only carries information regarding the high-level concepts that have been modeled (e.g., the steps of a certain business process), not details that are specific to the CMS domain (such as role, user, visual theme, or module). We consider that this is because the authors have also tried to restrain themselves to a single language, which ultimately brings the typical compromise between *low-level details* that can be modeled and *ease of learning and using* the language.

Summary

The usage of CMS systems as the base platform for the definition of new websites is rapidly growing in popularity. Examples of this boom can be found in the amount of CMS systems available and the expanding number of features supported by them, which are steadily increasing.

In this chapter, we have analyzed some CMS systems (an analysis that is further explored in Appendix B), according to aspects that are typically relevant when developing CMS extensions, customizations, or even CMS-based web applications with a considerable degree of complexity. Figure 5.1 provides a simple mindmap overview of the analyzed aspects.

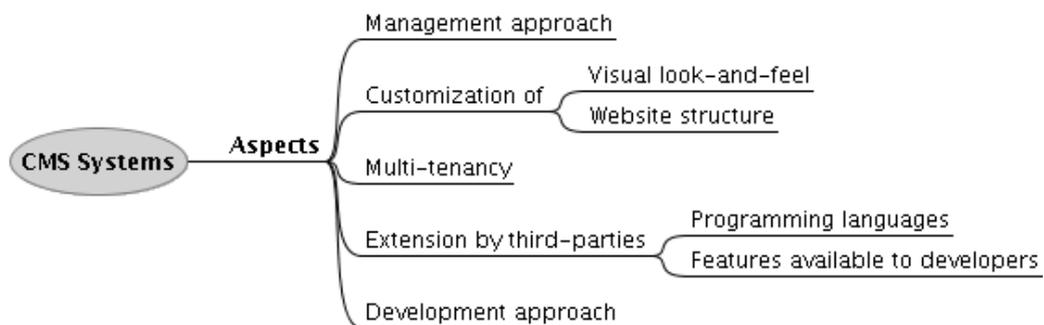


Figure 5.1: Overview of aspects and problems regarding CMS systems.

In the next chapter, we present our proposed solution for the thesis statement presented in Chapter 1. This solution uses the know-how obtained from the analysis presented in Chapters 2–5 to define an MDE-oriented approach for the development of CMS-based web

applications. This approach differs from the other approaches presented in Chapter 4, as it uses two different modeling languages and a model synchronization mechanism to maintain consistency between user models specified in those two languages.

Chapter 6

The CMS Multi-Language Model-Driven Approach

Good design adds value faster than it adds cost.

THOMAS C. GALE

After analyzing the State of the Art presented in Chapters 2–5, we found that there is still a considerable gap between (1) the domain of CMS systems, (2) the development of web applications, and (3) the usage of model-driven development approaches. This gap, in turn, poses a significant problem when developing web applications that would be suitable for deployment on CMS systems, as developers are then typically forced to choose between:

- developing a CMS-based web application by using a traditional – manual – development approach, with all the problems that such approaches entail; or
- specifying the web application using a model-driven development approach (such as the ones presented in Chapter 4 and Appendix A), and often be forced to “reinvent the wheel” for some functionalities that are usually provided out-of-the-box by CMS systems (e.g., support for forums, blogs, or polls).

This chapter provides our proposal for an MDE-oriented approach that we consider effectively addresses this gap.

The current chapter starts by providing a brief analysis of the main problems that we have found regarding the aforementioned topics, which are of particular relevance for the development of CMS-based web applications. Afterward, it presents an overview of the proposed solution.

6.1 Problems

As was observed in previous chapters, the development of web applications supported by CMS platforms is typically done using traditional software development processes, in which source code is the primary artifact, and design models and documentation are considered support artifacts. Such processes are time-consuming and error-prone, as they rely heavily on programmers and their execution of typically repetitive tasks. On the other hand, MDE development approaches aim to leave most of these repetitive tasks to automated model transformations.

Although there are currently some MDE approaches for web application development, there is no such approach for CMS-based web applications. This is understandable, because the idea of using CMS systems as platforms for more complex web applications is fairly recent, but the fact remains that these systems provide a degree of extensibility that would be worthwhile to exploit.

Moreover, most MDE approaches for web application development do not provide adequate support for the various perspectives that the application's stakeholders have regarding that application. These perspectives are often (1) captured in a manual fashion (by using means like requirements documents and meetings with stakeholders), (2) manually interpreted by the application's development team (including system architects and domain experts), and then (3) provided to the application's programmers for implementation into a concrete system. However, most of this processing is done in a manual fashion *because* of the various kinds of stakeholders that are involved. It is typically considered that dealing with all those different points of view is easier to do by using *natural language* (one of the most expressive languages that we have, although sometimes ambiguous), which in turn is a language at which humans are still much more adept than computers.

Another problem in current MDE approaches is that they either: (1) include too many low-level details into the modeling language, in an attempt to make it more expressive; or (2) try to include as few low-level details as possible into the language, to facilitate its learning and usage. Although this would seem obvious (as it would be very difficult, if not impossible, to define a modeling language that would be expressive enough for the needs of each stakeholder), it actually reveals the consequences of the compromise that must ultimately be adopted because of the problem mentioned in the previous paragraph.

Finally, the ultimate goal of most current MDE approaches is still to obtain *source code*, instead of *models*. This often leads to developers "cheating" by modifying the generated source code, while the model itself remains unchanged (and thus unsynchronized with the source code). Although there are techniques to allow manual changes to source code while maintaining its synchronization with the model [SV 05_b], the fact remains that

developers are usually required to manually edit source code when the modeling language is not expressive enough.

6.2 Proposed Solution: a Multi-Language Approach

To address the identified problems, we propose an MDE-oriented approach for the development of CMS-based web applications [SS 09_a]. This approach presents some noteworthy differences from other MDE-oriented approaches for web application development, namely: (1) it is based on the usage of *multiple* modeling languages; and (2) it uses a *model synchronization* mechanism to a) ensure consistency between models of different languages, and b) allow stakeholders to concurrently change different kinds of models (that correspond to different perspectives of the desired web application) without needing to worry about possible loss of information. Figure 6.1 provides a simplified overview of the proposed approach.

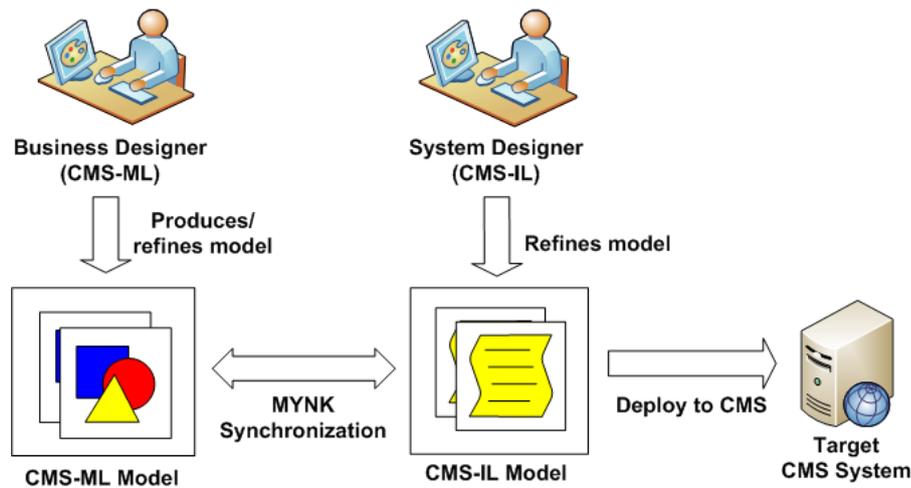


Figure 6.1: Overview of the proposed MDE-oriented development approach.

Instead of defining a single CMS-oriented modeling language, this approach defines *two* such languages: (1) CMS-IL (CMS Intermediate Language), a low-level language that provides a common ground for the specification of CMS-based web applications; and (2) CMS-ML (CMS Modeling Language), which provides a set of elements that can be used to define a high-level model of a web application. In addition to the CMS-ML and CMS-IL languages, the approach also considers the MYNK (Model sYNchronization framework) language, which supports the model synchronization mechanism.

It should be noted beforehand that the CMS-IL and CMS-ML languages are not meant to be sufficient for supporting *every* kind of stakeholder, as such a goal wouldn't be practical. Instead, and in order to validate our thesis statement, in this dissertation we focus on a

very limited set of stakeholder types, which we consider to be sufficiently representative of the stakeholders that typically participate in this kind of software development projects. Because of this possible limitation regarding the types of stakeholder considered, we also propose a brief set of guidelines for the creation of new modeling languages that can be used with this model synchronization mechanism. These guidelines, which are described in Section 6.3, were used in the definition of the two languages, and can be used in the definition of new modeling languages to suit the needs of additional kinds of stakeholder.

6.2.1 CMS-ML

CMS-ML is a graphical modeling language that addresses the specification of CMS-based web applications in a high-level and platform-independent manner. It was designed to enable non-technical stakeholders to quickly model a CMS-supported web application, namely by allowing such stakeholders to easily read, understand, and change a model so that it more accurately reflects their intent. Furthermore, CMS-ML also features an extensibility mechanism, aimed at supporting a stakeholder's specific requirements and the modeling of web applications with a higher degree of complexity.

The target audience for CMS-ML is, as already mentioned, non-technical stakeholders, who will model *what* the web application should be. However, the language's extensibility mechanism does present some modeling elements that are more oriented toward *how* the web application should perform its role. These elements may require that stakeholders have some technical understanding of how the web application should function; nevertheless, the usage of such elements is completely optional.

6.2.2 CMS-IL

CMS-IL is a textual low-level language, with a higher level of expressiveness than CMS-ML, that is nevertheless independent of any particular CMS platform. This language provides structural modeling elements but, unlike CMS-ML, it also places greater focus on *how* the web application should work. Thus, it defines a set of programming language-like modeling elements, which enable the specification of a CMS-based web application at a near-implementation abstraction level.

The ultimate objective of CMS-IL is to provide a common ground for the specification of CMS-based web applications, unlike CMS-ML that aims to provide a way to produce web application models in a simple fashion.

The target audience for CMS-IL is technical stakeholders (namely developers). Of course, these stakeholders should be comfortable with the concepts that can typically

be found in CMS systems, such as the ones presented in the CMS reference metamodel defined by [Car 06] (e.g., DynamicWebPage, WebComponent, Visual Style, or Workflow).

6.2.3 MYNK

MYNK is a textual language that supports the model synchronization mechanism between CMS-ML models and CMS-IL models. This mechanism, in turn, is the cornerstone of the proposed approach. This is because the desired increases in productivity and lower number of modeling errors depend on the usage of automated model synchronizations or model transformations; the alternative would be the manual conversion of CMS-ML models to CMS-IL (and vice versa), which would decrease productivity and increase the number of modeling errors made by designers, due to the error-prone nature of this task.

This language can be regarded as a model-oriented query language (although such a description is not entirely correct). It is loosely based on analyzed model transformation mechanisms and the SQL language (both of which were presented in Chapter 3). Of the former, it inherits the concept of establishing mappings between the considered modeling languages, while the latter is the inspiration for establishing model queries that can just fetch model elements or actually change them.

Unlike CMS-ML and CMS-IL, the MYNK language is not meant for use by stakeholders that participate in the web application's development. This is because these stakeholders typically do not need to be aware of how the model synchronization between CMS-ML and CMS-IL is performed. Instead, this language is meant for *language creators*, which can then use MYNK to extend this approach with additional modeling languages (CMS-oriented or otherwise). A possible reason for defining new modeling languages could be to support additional kinds of stakeholder.

6.2.4 Development Workflow

The proposed approach also considers a workflow that we consider as the one that will be used the most often. This workflow, which was already illustrated in Figure 6.1, considers two main stakeholder types, which we designate as the *Business Designer* and the *System Designer*.

The **Business Designer** – a generic term to identify non-technical stakeholders – can begin the workflow, by creating a CMS-ML model that represents the intended web application, according to some predetermined business requirements.

After using MYNK to obtain a CMS-IL model (corresponding to the aforementioned CMS-ML model), the **System Designer** – which, as opposed to the Business Designer, is a term to identify *technical* stakeholders – should determine whether that CMS-IL model is

satisfactory, namely by identifying any particular requirements that could not be addressed by CMS-ML alone; if any such requirements exist, the obtained CMS-IL model must be further modified by the System Designer to address them. While the System Designer changes the CMS-IL model, the MYNK synchronization mechanism should maintain the CMS-IL and CMS-ML models (which correspond to the same target web application, viewed through the perspective of different stakeholders) consistent with each other.

After the refinement process, the CMS-IL model should be an accurate representation of what the intended web application should be. However, if the Business Designer does not agree with the CMS-ML model (which was automatically updated in the meantime, via MYNK, to reflect the changes made by the System Designer to the corresponding CMS-IL model), then the Business Designer will have to change the CMS-ML model, triggering another round of the MYNK synchronization mechanism once again.

When the CMS-IL model (and, by extension, the corresponding CMS-ML model) is considered to be satisfactory, it is deployed onto a target CMS system in one of two ways, depending on the CMS itself: (1) deployment to a *CMS Model Interpreter* (or just *Interpreter*) component that is already available on the CMS system, or (2) generation of low-level artifacts and subsequent installation.

A CMS Model Interpreter component is a component that is installed on the CMS system, and has the responsibility to (1) receive an input CMS-IL model, and (2) deploy the received model, namely by configuring the CMS system to reflect the facts expressed in the model. This Interpreter component is inspired on [HP 05] which, although not focused on CMS systems or web application development, defines a model-driven runtime (MDR) environment that receives a model (specified using UML class and activity diagrams, annotated with OCL constraints) and interprets it at runtime, effectively allowing designers to *run* their models.

We consider that the first deployment alternative – using a CMS Model Interpreter component – is typically preferable, as it only requires that a CMS Administrator (i.e., a user with administrative privileges) upload the CMS-IL model into the Interpreter. Nevertheless, this first alternative will not be feasible in CMS platforms that do not have such a component available; in such cases, the second alternative (not illustrated in Figure 6.1, for simplicity) requires the intervention of a software developer – to compile the generated artifacts into a form that is executable by the CMS – and of a CMS Administrator, in order to both deploy the compiled artifacts and make any necessary configuration changes.

Although this approach does not remove the need for an iterative development process, we consider that it potentially mitigates the additional work of handling mismatches between the stakeholders' perspectives and their understanding by the developer.

Finally, it should be mentioned that we do not exclude a possible scenario in which (1) a Business Designer defines a CMS-ML model, and afterward (2) deploys it on the CMS Model Interpreter component of the target CMS system, bypassing the model's refinement by the System Designer (e.g., for quickly obtaining a prototype of the desired web application). However, this could easily be addressed by allowing the aforementioned Interpreter component to (1) receive an input CMS-ML model, (2) internally obtain the corresponding CMS-IL model, and (3) deploying the obtained CMS-ML model. Thus, we will not address this particular scenario in this dissertation.

6.2.5 The ReMMM Metamodel

The *MoMM metamodel* (presented in Chapter 2 and described in [AK 01]) presents a set of characteristics that address the issue of defining modeling languages while considering *multiple metalevels* (other than the classical class–instance metalevel duo). However, this metamodel also has some disadvantages, of which we highlight the following:

- MoMM does not consider the *links* between elements of different metalevels to be themselves **ModelElements**. In other words, although there is an ontological instantiation mechanism (see Chapter 2) that is implied by **ModelElement**'s **class** association, it is not possible for a model designer to *reference* a certain instance-of link without referencing the corresponding **ModelElement** instance; and
- It is too simple to define useful metamodels (e.g., unlike UML, MoMM does not define the concept of **Data Type** or **Enumeration**, which in turn are necessary when specifying possible value constraints for **Attributes**). Nevertheless, it should be noted that considering this as a disadvantage is arguable, because MoMM's goal is to be the simplest metamodel that is necessary to define new metamodels with multiple metalevels.

These shortcomings, in turn, were the motivation for defining the **ReMMM** (Revised Metamodel for Multiple Metalevels) metamodel, a variant of MoMM aimed at addressing these issues. Although it is a variant of MoMM, ReMMM can also be considered as being *defined with MoMM*, as the latter is reflexive (and thus could be used to define not only itself but also extensions to itself). The abstract syntax of ReMMM is represented in Figure 6.2 (we will use UML's concrete syntax to define ReMMM-based models in upcoming chapters, in order to facilitate the reader's interpretation).

ReMMM shares MoMM's notion that *a model consists only of a set of **Model Elements** which are linked among themselves*. In fact, ReMMM and MoMM are similar to each other, as the former mostly consists of extensions to the latter. The most noteworthy differences between them are the following:

- The **Association** can also be of one of three types (**Normal**, **Aggregation**, or **Composition**), which have the same semantics as their UML-homonyms;
- The **Concept** element is defined to represent *concepts* (i.e., metaclasses that have some meaning and representation, which is specified by a language designer). The reason why **Model Element** is not abstract (considering all of its specializations) is that it is also used to represent elements that (1) are *only* instances of another **Model Element** and (2) are not themselves metaclasses for other **Model Elements**; and
- The **Attribute** can hold *multiple* values of its type.

Regarding **Attribute**'s link to **Model Element** (its **type**), we opted to maintain this link unaltered (as opposed to linking **Attribute** to **Data Type**, which is the case in UML), in order not to curtail the expressiveness of this metamodel (e.g., by preventing scenarios in which an **Attribute** references a specific **Model Element** instance). Nevertheless, we do not take advantage of this feature in ReMMM-based metamodels, so that we can use UML's concrete syntax as-is and facilitate the reader's interpretation of those metamodels.

Its MoMM-based heritage notwithstanding, ReMMM could also be considered as being defined using MOF (or even UML), although in this case ReMMM as a metamodel would bring no added-value (regarding expressiveness) in relation to its own (meta-)metamodel.

Finally, ReMMM shares MoMM's trait of being *reflexive* (i.e., ReMMM can be used to describe itself). As was previously mentioned in Chapter 2, we take advantage of this fact in our research work, as the metamodels for the CMS-ML and CMS-IL languages were defined using ReMMM as their metamodel. More specifically, we take advantage of (1) the reflexiveness of ReMMM and (2) the explicit definition of the **Instantiation** concept to define these languages using *multiple metalevels* (further details of which will be given in Chapters 7 and 8, where the languages are respectively presented). The MYNK model synchronization language also assumes that the metamodels of the models being synchronized can be themselves specified using ReMMM (a fact that we expect to be true for most existing modeling languages, because of ReMMM's simplicity and high degree of expressiveness); this is because MYNK needs to have a "common ground" that allows it to receive and manipulate many kinds of models.

6.3 Guidelines For Language Specification

As was already mentioned in this chapter, the approach proposed in this dissertation contemplates only two languages, CMS-ML and CMS-IL. However, it would be rather naive to just assume that these two languages would be sufficient to support any kind of stakeholder in the development of a real-life web application. This, in turn, increases the likelihood of cases in which the approach must be extended with additional CMS-oriented

modeling languages, designed to support new stakeholder perspectives. Therefore, it becomes necessary to determine how to create such languages, and make them suitable for usage with MYNK.

During the research work presented in this dissertation, we have identified a small set of guidelines that proved helpful in the construction of the CMS-ML and CMS-IL languages. Although these guidelines should not be interpreted as steps in a language definition methodology, we believe that they can help to address some potential problems that typically occur during a language's design process, such as metamodeling issues (described in Chapter 2) or multiple alternatives for the concrete syntax. Furthermore, it should be noted that some of these guidelines were based on the DSM language definition rules provided in [KT 08], while others were based on our own previous experiences with the development of the XIS2 language [SSSM 07] and on available best practices for DSL design [KP 09, MHS 05].

The identified guidelines are explained below (it is assumed, of course, that the language designer has already searched for a DSL that addresses the desired problem-domain, and found no adequate languages).

Identify the target audience for the language. This is, in our opinion, one of the most important guidelines, as its results should be main factor in most of the language design decisions that will follow. In particular, it is important to ascertain:

- the audience's main concerns, regarding the language's modeling purpose; and
- the audience's expected level of *technical* expertise (considering that the goal of using the language will be to ultimately obtain a model that accurately represents software to run in a computer).

Although this guideline likely seems obvious to readers, it is important nevertheless because a model's ultimate purpose is to *answer questions* about the system that it represents [Fav 04.b]. Thus, common sense dictates that it is important to determine first what are the *questions* to be answered, and only after that define a way to answer those questions.

Identify the problem-domain that the language should address. Just as important as the previous guideline (and closely related to it), a correct and precise identification of the problem-domain is essential, so that the language designer can be aware of *what should be modeled* and what is irrelevant in a model for a desired web application.

A correct identification, in turn, can help to *minimize essential complexity* [Bro 87, AK 08] in the language, by avoiding the inclusion of unnecessary elements that will likely only contribute to making the language harder to learn and use. Furthermore, we are

confident that, depending on factors such as the problem-domain's essential complexity and the language designer's experience regarding that domain, this correct identification can help to *prevent accidental complexity* in the language altogether.

This identification will typically result in the definition of the language's abstract syntax (either formally with a metamodeling language, or just an informal text-based definition), which is why designers can consider that the core activities of this guideline are actually identifying: (1) the problem-domain's concepts that the language *must* support; (2) the relationships between those concepts; and (3) any partitions or views that are suitable for the language's target audience (if applicable) [IEEE 00].

Determine the degree of extensibility that the language should address. After identifying the problem-domain, the language designer should consider whether it will be necessary for stakeholders to change the language (e.g., by adding new elements or changing elements that are already present in the language).

In [Sta 75], Standish identifies three different approaches to programming language extensibility: (1) **paraphrase**, in which new features are added to a language by defining them only in terms of features that are already present in that language; (2) **orthophrase**, in which new features are added to a language without requiring the features that are already included in the language; and (3) **metaphrase**, in which a language's interpretation rules are changed, leading to existing language features being processed in a different manner. Although these approaches were identified in the context of programming language design, it can be considered that such programming languages are themselves textual cases of modeling languages (as was discussed in Chapter 2). Thus, these approaches may also be applied to extensibility in modeling language design, although the practical use of each approach can be subject to discussion, both in terms of (1) feasibility and of (2) added-value versus potential for users to make mistakes when modeling.

It should be noted that, because languages should continuously evolve [KP 09], it is possible (and perhaps recommended) to follow this guideline *after* the initial release of the language under construction, in order to avoid gold plating the language and thus minimize its accidental complexity. However, in such early stages, language designers should consider the target audience and ponder the possibility that the language's users may need to extend it with new features that the language designer could not possibly predict. This can guide the language designer in specifying the language in such a manner that it can be extended in the future without needing to recreate the entire language.

Considering the identified problem-domain, determine the language's modeling levels and their hierarchy. Although closely related to the previous guideline

regarding extensibility, the focus of these two guidelines is somewhat different: the previous guideline aims primarily at determining what aspects of the language should be extendable by its users, while the current guideline is where the language designer formally structures the language's concepts (from the problem-domain) into different *modeling levels* (i.e., metalevels).

More specifically, the language designer should consider the various concepts and relevant objects (previously identified during the analysis of the problem-domain), and determine any *instance-of* or *composition* relationships between them; in other words, the language designer must analyze the identified elements (concepts and objects), and determine (1) what classifies what, and (2) what are the building blocks for what.

This follows the notion of a *stratified design* approach [AS 96]. In this kind of approach, a complex system should be structured according to a set of levels described using different languages. Each level is specified by combining elements that are considered *primitives* at that level, and such primitives are themselves the result of combinations performed in the previous level. The authors also state that the language in each level should thus contain (1) primitives, as well as (2) combination and (3) abstraction mechanisms appropriate for the intended level of detail.

The explicit definition of these modeling levels (metalevels in metamodeling nomenclature) allows the language designer to establish a hierarchical structure between those metalevels. This hierarchy, in turn, ensures the practice of Strict metamodeling [AK 02, Küh 09] (already presented in Chapter 2), if the language designer ensures that: (1) no instance-of relationships are present *within* each metalevel; (2) instance-of is the *only* relationship that occurs between elements in different metalevels; and (3) *all* elements E_{L_1} that occur in a metalevel ML_1 are related (via the instance-of relationship) to elements E_{L_2} in another metalevel ML_2 .

Identify any constraints that may condition the choice of a metamodeling language. Besides the problem-domain's concepts and relationships, the language designer must also determine any constraints that may make certain metamodeling languages unsuitable for specifying the language's abstract syntax.

Although most metamodeling languages do not impose a considerable set of constraints due to their inherent simplicity, language designers should still identify those relevant constraints (related to either the problem-domain or the modeling levels identified), and evaluate whether the selected metamodeling language is able to model or satisfy those constraints in a satisfactory manner. Possible examples of such constraints are the usage of multiple-inheritance (e.g., the Kermeta metamodeling language¹ supports multiple-

¹*Kermeta*, <<http://www.kermeta.org>> (accessed on March 13th, 2012)

-inheritance by including a mechanism to address conflicts regarding operator and feature redefinition), or the need to specify particular restrictions over a specific model (e.g., to model the fact that an automobile should have four wheels).

Aside from the constraints identified, language designers also have to weigh the advantages and disadvantages of relevant issues, when considering whether to use (1) a general-purpose modeling language such as UML, or (2) a DSM-oriented language for the language's metamodel. Each of these approaches will have its own benefits and disadvantages. One of these important issues is tool support, as the adoption of a general-purpose language instead of DSM may facilitate the activity of creating a modeling tool. Another is the *maintainability* of the language's models (i.e., the possibility of new users to correctly *understand* and *change* the model), regarding which [CRR 09] points out an advantage in using DSM instead of UML.

Identify the kind of concrete syntax that the target audience is most comfortable with. This guideline is intended to address the dilemma of whether the language should have a graphical concrete syntax, a textual one, use a mixed approach to the language's concrete syntax (by providing a language containing both graphical and textual elements), or even by providing multiple concrete syntaxes. This, in turn, is crucial to ensure the language's quality and usability (according to the criteria mentioned in Section 2.3), and ultimately its acceptance by users [KT 08, BAGB 11].

To do so, the language designer must consider the target audience and its preferred manner of representing the desired system. The *audience's technical expertise* may also be an influencing factor in this decision: according to [KP 09], (1) a significant proportion of developers prefers text-based languages, because of their activities involving traditional programming and scripting languages, while (2) the general population typically prefers graphical languages, as long as they reflect the shapes of the real world entities that are being represented in the model. The *frequency* with which the language will be used should also be considered: a language that is used often should try and provide a concrete syntax that ensures its users can define models in a quick and error-free manner, while it is likely not problematic that a language which is only used occasionally (and briefly) not provide a syntax with which users can quickly define models.

Summary

In this chapter, we have presented our MDE-oriented approach for the development of CMS-based web applications. This approach differentiates itself from other existing approaches for web application development (some of which have already been analyzed in

Chapter 4) by (1) its usage of multiple modeling languages (instead of a single language, such as UML or WebML), and (2) its model synchronization mechanism.

More concretely, the approach defines the CMS-ML, CMS-IL, and MYNK languages. CMS-ML and CMS-IL are meant to be used by Business Designers (business stakeholders) and System Designers (programmers), respectively. On the other hand, the MYNK model synchronization language is meant to be used by *language developers* (e.g., DSL creators that wish to specify new CMS-oriented languages different from CMS-ML and CMS-IL) as a way to allow further kinds of stakeholder to participate in the web application's development process.

We have also presented a small set of guidelines for defining new modeling languages that can be used with MYNK. These guidelines enable the creation of further languages that better suit the needs of further kinds of stakeholder, which can prove useful in cases where CMS-ML and CMS-IL are not sufficiently adequate for creating the intended web application.

The following chapters are dedicated to presenting each of these languages in a greater level of detail (although not in an exhaustive manner), namely the rationale for the choices and compromises that were made when designing the languages. In particular, in the next chapter we present CMS-ML, this approach's modeling language for Business Designers to create a high-level specification of a CMS-based web application, which should provide a level of support for some (or all) of an organization's tasks.

Chapter 7

CMS-ML: CMS Modeling Language

A language that doesn't have everything is actually easier to program in than some that do.

DENNIS M. RITCHIE

A typical problem with current web application modeling languages is that their models are often computation-oriented (i.e., oriented toward programming-specific details), and are not understandable by non-technical stakeholders, in the sense that such people will not be able to look at a model and easily determine what information it conveys. This problem can be found in most web application modeling approaches (such as UWE [KK 08], XIS2 [SSSM 07], or WebML [BCFM 07, CFB⁺ 02]), some of which have already been analyzed in Chapter 4. Furthermore, this is an issue that can potentially impact our approach's usage in real-life web application development projects.

To address this problem, we propose **CMS-ML** (CMS Modeling Language) [SS 10-d], a graphical modeling language with the objective of allowing the specification of CMS-based web applications in a high-level and platform-independent manner. Furthermore, this language also allows for its extension, in order to address a stakeholder's specific needs and support the modeling of web applications with a higher degree of complexity.

The CMS-ML language is designed to enable non-technical stakeholders to quickly model a web application supported by a CMS system. More specifically, its main objectives are to allow stakeholders to:

1. *Read* a model, in the sense that a model should not be so complex that a stakeholder could easily get lost while looking at it;
2. *Understand* the model, i.e., correctly interpret the model's semantics; and
3. *Change* the model, so that it more accurately reflects the stakeholder's intent.

Although creating a CMS-ML model can be considered a particular case of changing a model (more specifically, a “blank” model), it is possible for CMS-ML-supporting modeling tools to provide a predefined model, with a few elements such as a basic web application with a page. However, because this chapter is dedicated to presenting the language (and not discussing such modeling tool decisions), we will not further elaborate on this topic.

CMS-ML is also designed with the intent of being a platform-independent language, and does not provide a large number of modeling elements with which to specify web application models. This limited number of elements is because the aforementioned intent would mean that the language could either (1) contain a superset of elements that can be found in one or more CMS systems, or (2) contain a subset of elements that can typically be found in various CMS systems. The first option would originate a language with a larger set of modeling elements, some of which may or may not have a platform-specific counterpart, while the second option would lead to a smaller set of elements that will probably be easy to map to a specific platform. However, considering that languages featuring a larger set of elements are likely to be harder to organize and remember (thus making the language more complex and harder to learn) [Har 07], the decision was made to define a smaller set of modeling elements.

Nevertheless, this limited set of elements could mean that CMS-ML is not expressive enough to model most kinds of CMS-based web applications (other than simple toy examples). In fact, when considering the elements available for modeling WebSite Templates (explained further down this text), it would not be difficult to define a case study that could not be modeled by CMS-ML. Although this lack of expressiveness is a result of a trade-off between language expressiveness and complexity, we have tried to mitigate it by (1) allowing some modeling elements to freely express a set of textual properties (instead of constraining those properties to sets of possible values), (2) defining a language extension mechanism, which we designated as *Toolkit*, and (3) defining a tag-like mechanism, by which elements can be *annotated* in a manner that will not make the model become platform-specific. These aspects will be explained further in this chapter.

In this chapter, we provide a brief description of CMS-ML. More specifically, this chapter describes: (1) the artifacts and modeling roles considered; (2) the language’s metalevels, establishing the relationship between the different models considered, and the underlying rationale; (3) the modeling elements available for specifying a web application; and finally (4) the modeling elements available for extending the language, which can be used afterward when modeling a web application. For additional information, we also advise readers to consult the “CMS-ML User’s Guide” [SS 10_c].

It should be noted beforehand that CMS-ML’s abstract syntax is described in this chapter using UML, because: (1) although the language does not use UML as a metamodel

(because of the reasons explained in Section 7.3), it is nevertheless possible to use UML to illustrate each of CMS-ML’s metalevels; and (2) it is expected that most readers are familiar with UML’s class diagrams, due to its widespread use nowadays.

Furthermore, as was already mentioned in Chapter 1, the terms “website” and “web application” will generally be used in an interchangeable manner throughout this chapter.

7.1 Guidelines

Before starting the description of the CMS-ML language, it is important to first establish some answers for the guidelines identified in Section 6.3 (see Chapter 6). The next paragraphs present the answers for these guidelines, which were used to steer the process of defining CMS-ML. It should be noted that these answers are presented in an abbreviated manner, as a complete answer would be considerably long and the information contained in those answers is presented in the remainder of this chapter anyway.

Identify the target audience for the language. The target audience for CMS-ML are non-technical stakeholders (sometimes also called *business users* [Gho 11]), with varying degrees of technical know-how. These stakeholders are aware (due to their daily activities when dealing with web browsers and CMS-based web applications) of: (1) basic CMS concepts, such as Website, Page, Role, and Component (also called Widget); (6) the basic HTML elements of the aforementioned Components, such as Link, Button, Text Box, and List; and (7) the specification of activities, modeled as flow-charts or similar diagrams. They are, however, not familiar with typical programming languages and activities, which often leads to a *semantic gap* between them and developers [Gho 11].

Identify the problem-domain that the language should address. The problem-domain in question is the universe of CMS systems and CMS-based web applications, which are analyzed in Chapter 5 and Appendix B. More specifically, this domain consists of CMS-based web applications aimed at supporting a variety of business-oriented tasks and objectives. These web applications, in turn, are obtained through: (1) the usage of common CMS elements, such as the aforementioned Website, Page, Role, and Component (also called Widget); and (6) the definition of CMS elements specifically oriented toward the aforementioned tasks. We will not expand on the identification of this problem-domain, for the reasons already mentioned at the beginning of this section.

Determine the degree of extensibility that the language should address. According to the problem-domain identified in the previous item, the stakeholders will need

to be able to define new kinds of CMS Component (which will be afterward instantiated in the modeled Website), with a layout of HTML components that will only be known at design-time. These new Components are particularly oriented toward supporting specific workflows, which are also known at design-time. Furthermore, these workflows also require that stakeholders be able to specify that only specific roles can perform certain actions in a workflow.

However, stakeholders will not need to change the Components that are already present in the CMS. This is because: (1) most CMS systems do not consider the customization of those out-of-the-box Components; and (2) such Components are typically created with a generic purpose in mind, instead of the purposes that the target audience aims for. These purposes, in turn, often require the creation of brand-new Components.

Considering the identified problem-domain, determine the language’s modeling levels and their hierarchy. Taking into consideration the answers obtained in previous items, the important composition relationships identified are: (1) between a Component and its HTML components (when defining new Components); (2) between a workflow and its actions; and (3) between a Website, its Pages, and their Components.

Furthermore, the following relevant instance-of relationships have been identified: (1) the modeled Website will contain instances of stakeholder-defined Components; and (2) some of the roles in the CMS will be classified as instances of roles that participate in the modeled workflows.

Aside from these relationships, and from the obvious instance-of relationships that will be created when possibly deploying a CMS-ML model to a CMS system (a deployment which, in turn, requires the execution of actions such as database configuration), no other relationships of interest have been identified. Thus, the CMS-ML language requires the existence of at least two modeling levels (excluding the aforementioned level containing the CMS instances): one for the stakeholders to model their own Components and workflows, and the other for modeling the Website while using instances of those Components.

Identify any constraints that may condition the choice of a metamodeling language. The only constraint worth mentioning is the one that is immediately derived from the previous answer: the chosen metamodeling language will have to support stakeholder modeling in at least two metalevels (not counting the “reality” metalevel, such as M0 in OMG’s metamodeling architecture). However, aside from the need to represent the relationship between the aforementioned metalevels, no particular constraints have been identified in each of the modeling levels that would make most metamodeling languages, such as MOF or even UML, inadequate for describing them.

Identify the kind of concrete syntax that the target audience is most comfortable with. The target audience does not consist of developers (who are more used to dealing with textual programming and scripting languages), and so a visual modeling language is likely preferable [KP 09]. Nevertheless, it is not necessary for this language to be purely graphical, as it may contain some snippets of text.

Additionally, *some* elements of the target audience may be somewhat familiar with visual modeling languages (for purposes such as domain modeling or workflow specification), namely UML and its class and activity diagrams.

After obtaining these guidelines, the CMS-ML language was defined according to them. The rest of this chapter is dedicated to the presentation of this language, and the explanation of some design decisions involved in this process.

7.2 Model Types and Modeling Roles

Modeling with CMS-ML is mainly focused on three different, and complementary, model types: (1) *WebSite Templates*, (2) *WebSite Annotations*, and (3) *Toolkits*. Figure 7.1 suggests how these models are related to each other: both WebSite Templates and Toolkits can reference other Toolkits (as discussed later in this chapter), but WebSite Annotations can *only* decorate WebSite Templates.

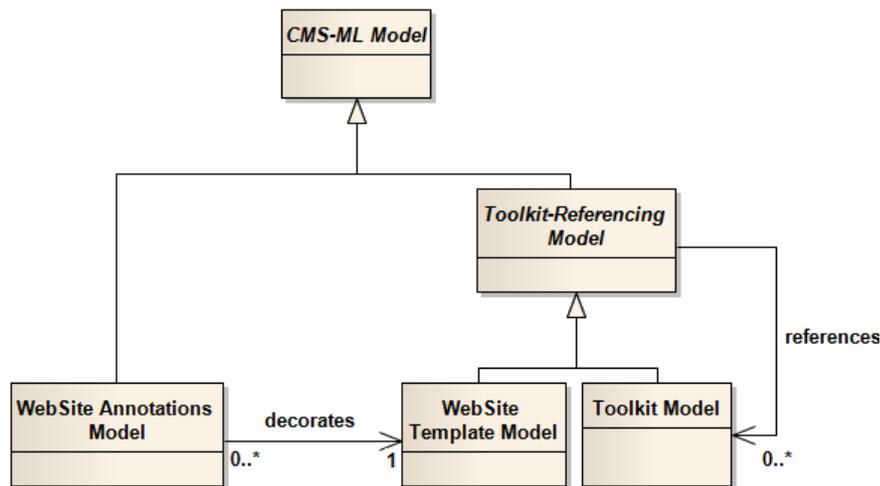


Figure 7.1: Relationship between the different CMS-ML models.

A **WebSite Template** (also just called *Template*, for simplicity) is a model that reflects the intended structure of the web application; this Template is modeled using *CMS elements* – such as `Role`, `Dynamic WebPage`, and `WebComponent` – that are provided by CMS-ML.

On the other hand, a **Toolkit** allows the definition of new CMS-oriented modeling elements, namely by specifying a domain model, user interface, and corresponding behavior. As previously mentioned, a WebSite Template can then reference a Toolkit (or a set of Toolkits), which in turn makes the Toolkit's elements available for use in the Template. Furthermore, a Toolkit can also reference other Toolkits, enabling the possibility of scenarios in which a Toolkit *A* refines and extends functionality that was previously defined in another Toolkit *B*.

Finally, the elements of a WebSite Template can be *annotated* by means of a **WebSite Annotations** model (or just Annotations). This model *decorates* a WebSite Template, allowing Template designers to specify CMS-specific properties (e.g., configuration settings) without polluting the Template itself with platform-specific details. Thus, from a practical perspective, CMS-ML WebSite Template designers do not view two different models – the Template and the Annotations – but rather a single model that results from combining those two models (i.e., a model that is actually the result of decorating the Template with the Annotations).

It is not mandatory that a single CMS-ML designer have the skills to create both WebSite Template and Toolkit models. Instead, we consider that CMS-ML development will often be performed according to the following modeling roles, as suggested in Figure 7.2:

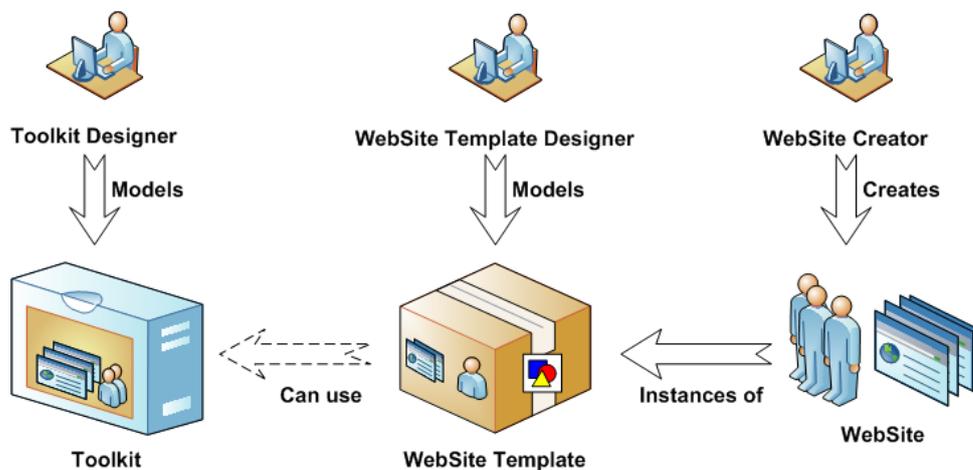


Figure 7.2: Modeling roles and artifacts considered by CMS-ML.

- The **Toolkit Designer**, who models Toolkits;
- The **WebSite Template Designer** (usually designated just as *Template Designer*, for text simplicity), who models a WebSite Template and, optionally, annotates it with a WebSite Annotations model; and
- The **WebSite Creator**, who instantiates the various elements defined in the WebSite Template.

This chapter will explain CMS-ML while taking these roles into consideration.

7.3 CMS-ML Architecture

Before starting the description of CMS-ML, it is important to highlight that WebSite Templates and Toolkits are located in *different metalevels*. While WebSite Templates are meant to create abstractions (i.e., models) of concrete web applications by using CMS-oriented elements, Toolkits use generic modeling elements to create new CMS-oriented modeling elements. Because some Toolkit concepts are also specializations of WebSite Template concepts (and so instances of those Toolkit concepts are automatically considered as instances of the corresponding WebSite Template's concepts), Template Designers can then use those Toolkit concepts to create WebSite Templates in the same manner as when using the predefined Template modeling elements.

Figure 7.3 depicts the metalevels that are considered by CMS-ML:

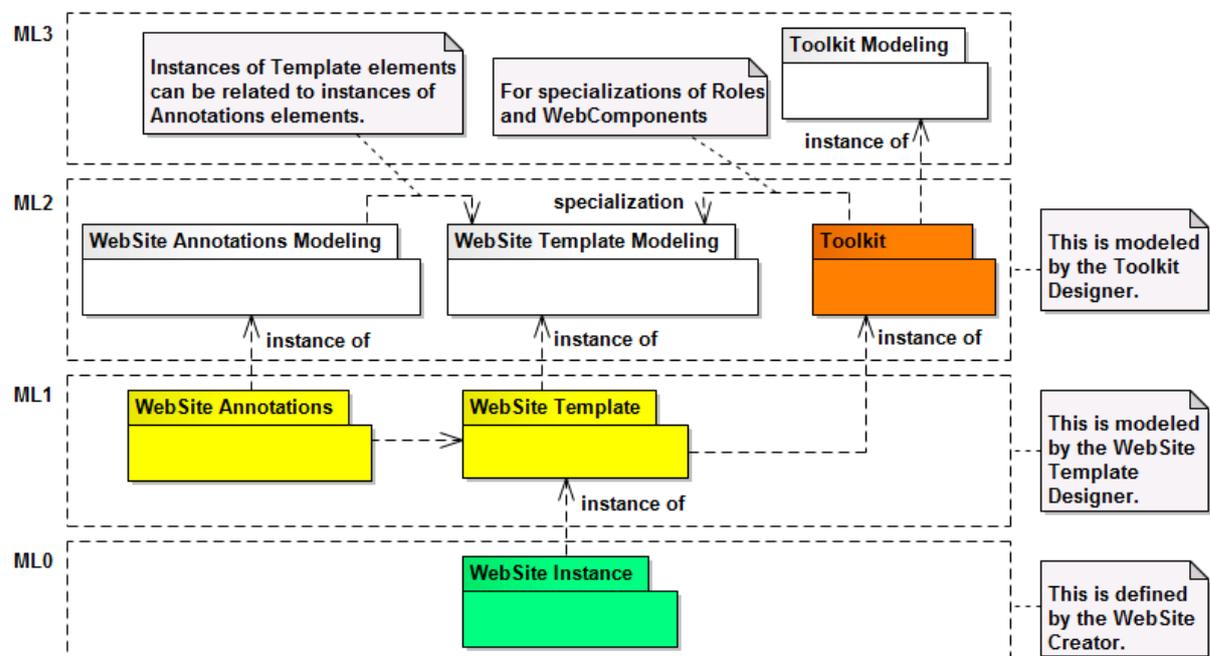


Figure 7.3: Metalevels considered by CMS-ML.

- Metalevel **ML3** contains the *Toolkit Modeling* model, which provides the definition of the generic Toolkit modeling elements that will be used to define Toolkit models. This metalevel cannot be changed by designers of any kind;
- The **ML2** metalevel contains the *WebSite Template Modeling* and *WebSite Annotations Modeling* models, which provide the modeling elements that will be used to define WebSite Template and WebSite Annotations models. Furthermore, Toolkit Designers can create instances of the generic modeling elements located in ML3, in order to define new elements that specialize WebSite Template Modeling elements. However, like the Toolkit Modeling models in ML3, the WebSite Template Modeling

and WebSite Annotations Modeling models are fixed and cannot be changed by anyone;

- In metalevel **ML1**, WebSite Template Designers can create *WebSite Template* and *WebSite Annotations* models, by using the modeling elements defined in ML2. These elements include not only those provided by the WebSite Template Modeling and WebSite Annotations Modeling models, but also the elements defined by any Toolkit model(s) that are made available to the WebSite Template, via the `Toolkit Import` concept (explained in Section 7.8);
- Finally, in metalevel **ML0**, the WebSite Creator (not the Template Designer) uses the elements defined in the WebSite Template model (along with its decorating WebSite Annotations model, if any) to configure a particular CMS installation. This will usually require some CMS-specific mechanism that establishes a mapping between an instance and a model element (e.g., a column in a Users database table that, for each row/CMS user, identifies the corresponding Template `User`).

Note that there is some similarity between the ML1 and ML0 metalevels and the M1 metalevel found in OMG's specification of UML [OMG 11_e], because their purpose is practically the same. The main difference is that, while in UML instances of `Class` and `Object` are located in the same metalevel (M1), in CMS-ML they are located in the ML1 and ML0 metalevels, respectively.

The rationale for this metalevel architecture was: (1) to address the issue of language extension in a simple, yet elegant, manner; (2) to reduce the accidental complexity [AK 08] that is usually derived from using modeling patterns similar to type–instance in the same metalevel; and (3) to obey the strict metamodeling doctrine [AK 02, Küh 09], in which there should be no instance-of relationships crossing more than one metalevel boundary.

7.4 WebSite Template Modeling

The CMS-ML language (more specifically, the WebSite Template Modeling metamodel, as illustrated in Figure 7.3) provides a set of CMS-oriented modeling elements – typically called *CMS elements*, because CMS systems often provide some support for these elements – that WebSite Template Designers can use to define their Templates for CMS-based web applications. A WebSite Template model, consisting of instances of those CMS elements, is defined according to the following set of views (illustrated in Figure 7.4):

- The *Structure view*, which specifies the web application's structural components;
- The *Roles view*, which deals with the set of responsibilities that the web application expects its users to assume; and

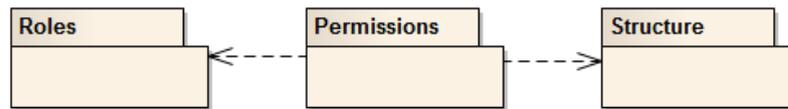


Figure 7.4: Views involved in the definition of a WebSite Template.

- The *Permissions view*, specifying which roles have access to the web application’s structural components.

The focus of these views shows that a WebSite Template deals with the *structural* concerns of the web application, and so it will be used mainly to *configure* the CMS system when the web application is deployed on it (e.g., create new pages and roles if they do not exist). The *behavioral* concerns, on the other hand, are only specified in Toolkits (described in the next section) because (1) a CMS-based web application’s behavior is usually defined by the **WebComponents** available in the CMS (e.g., an **HTML WebComponent** will behave differently than a **Forum WebComponent**), and (2) even CMS administrators are typically unable to change the CMS’s behavior itself (unless they possess programming skills *and* have access to the source code of the CMS’s platform), and can only change some of its parameters.

7.4.1 Structure View

The Structure view is the most important, as it immediately conveys the web application’s page structure, by using a set of CMS-oriented concepts: (1) **WebSite**, which represents instances of the web application, and serves both as a container for **Dynamic WebPages** and as the element that will import Toolkits (explained further down this text); (2) **Dynamic WebPage**, representing the dynamically generated pages (in the sense that their contents can be changed through the CMS interface) that users will access; (3) **Container**, which is modeled within a specific area of a **Dynamic WebPage** and holds a set of **WebComponents**; and (4) **WebComponent**, representing the functionality units (e.g., a Blog or a Forum) with which the user will interact.

This view is further divided into two smaller views, the *Macro Structure view* and the *Micro Structure view*. The Macro Structure view specifies an overview of the web application, modeling only **Dynamic WebPages** and the relationships between them, while the Micro Structure view is where each **Dynamic WebPage**’s internal structure is specified (i.e., what **WebComponents** are in each **Dynamic WebPage**, their location, and their order relative to one another). Figure 7.5 presents the abstract syntax for the Structure view, where the Macro Structure and Micro Structure views’ concepts (and the relationships between them) can be observed.

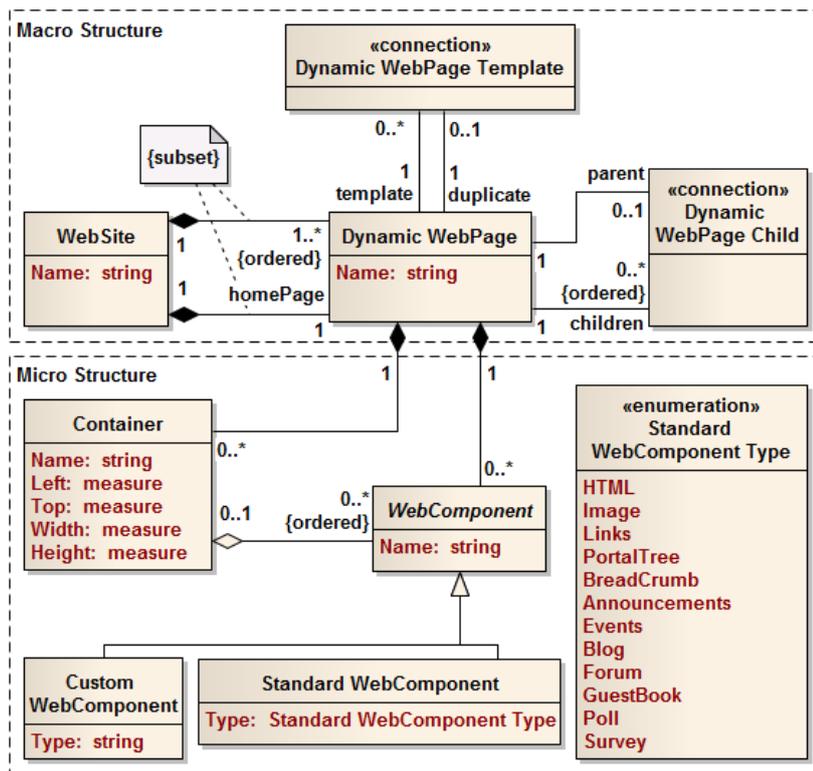


Figure 7.5: Abstract syntax for the WebSite Template’s Structure view.

On the other hand, Figure 7.6 depicts two examples of the Structure view’s concrete syntax: Figure 7.6a illustrates the Macro Structure view, namely a simple `WebSite`, `My Personal WebSite`, that contains two `Dynamic WebPages` – `Home` and `About me` –, while Figure 7.6b shows the Micro Structure view’s definition of the aforementioned `Dynamic WebPage Home`, namely its three `Containers` – `Banner`, `Body`, and `Navigation Bar` – and two `WebComponents` – `My Blog` and `My TV Viewer`. These examples also show that CMS-ML’s concrete syntax was defined with the purpose of being easy to understand and to draw manually, without requiring the explicit use of specialized modeling tools to create models.

7.4.2 Roles View

The Roles view describes the various kinds of expected responsibilities that the CMS-based web application’s users are expected to have when interacting with it. This view defines two concepts, **Role** and **Role Delegation**: the former models those expected responsibilities, while the latter specifies whether such responsibilities can also be played out by other Roles (or, in other words, if a Role will share its responsibilities with other Roles). Figure 7.7 illustrates the abstract syntax for the Roles view.

Regarding the graphical representation of this view, Figure 7.8 depicts the concrete syntax for each of these concepts: Figures 7.8a, 7.8b, and 7.8c illustrate regular, ad-

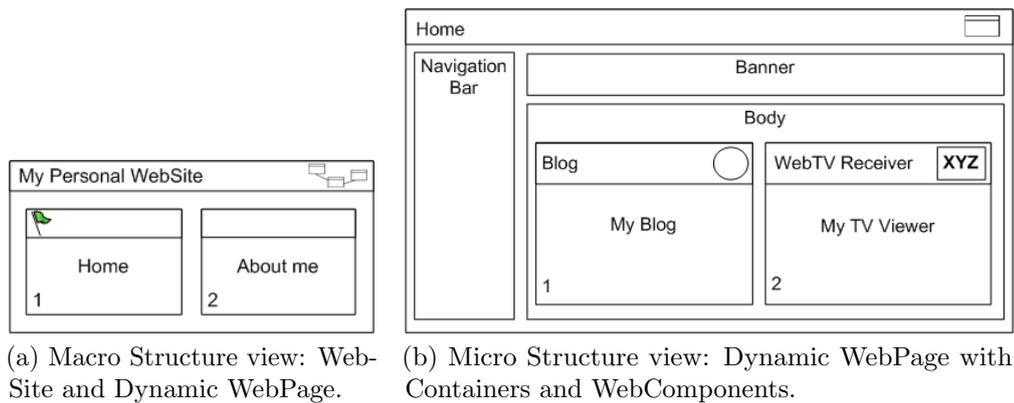


Figure 7.6: Concrete syntax for the Website Template's Structure view.

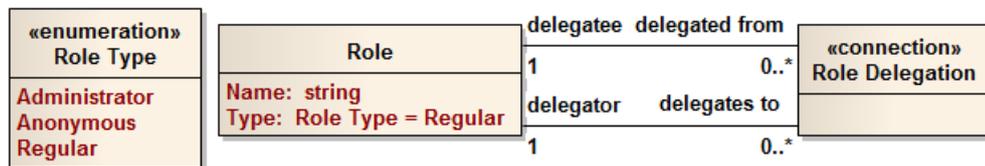


Figure 7.7: Abstract syntax for the Website Template's Roles view.

administrator, and anonymous Roles (respectively), while Figure 7.8d illustrates a Role Delegation relationship between two regular Roles (in which the Role Manager shares its responsibilities with the Role Secretary, and so any Secretary will effectively have the same authority to perform actions as any Manager).

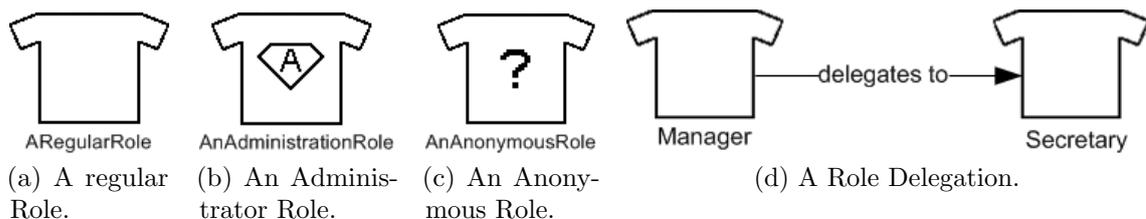


Figure 7.8: Concrete syntax for the Website Template's Roles view.

Some notes should be made about the Role Delegation concept, namely regarding its relationship to UML's Generalization concept (which is also used in Use Case diagrams), and the delegation of responsibilities between Roles.

The first note is that Role Delegation is *not equivalent* to UML's Generalization. This is because Generalization typically has the semantics of an element *inheriting characteristics* (e.g., element attributes) from another element *and potentially refining them*. On the other hand, the only purpose of Role Delegation is to indicate that a Role can have the same responsibilities of another Role (which is different from saying

that “a Role *A* is a Role *B*”), and not the refinement of a Role’s characteristics. An example of this difference can be found in Figure 7.8d, as it does not go against common sense to believe that a Role **Manager** can delegate (most of) its responsibilities to a Role **Secretary**, but it would be wrong to infer that a **Secretary** is a **Manager**. Because of this semantical issue, we opted to not use a **Generalization**-like concept and instead define a more adequate one.

Furthermore, Role **Delegation** does not provide the ability to make a *partial* delegation of responsibilities. This is because, if such a delegation is necessary, it can be easily achieved by (re)defining the existing Roles as a result of delegations by “smaller” Roles (in the sense that those smaller Roles have less responsibilities than the existing ones). More specifically, this is done by: (1) creating multiple smaller Roles, each representing a responsibility to be delegated; (2) having those new Roles delegate to already existing Roles; and (3) also having some of those new Roles delegate to other Roles, according to the desired partial delegation of responsibilities.

7.4.3 Permissions View

The Permissions view establishes a mapping between the web application’s structure and its roles, namely to specify *who can do what*. It defines two concepts, **Dynamic WebPage Permission** and **WebComponent Permission**, which respectively enable the creation of Role–**Dynamic WebPage** and Role–**WebComponent** links. Figure 7.9 provides an illustration of these concepts.

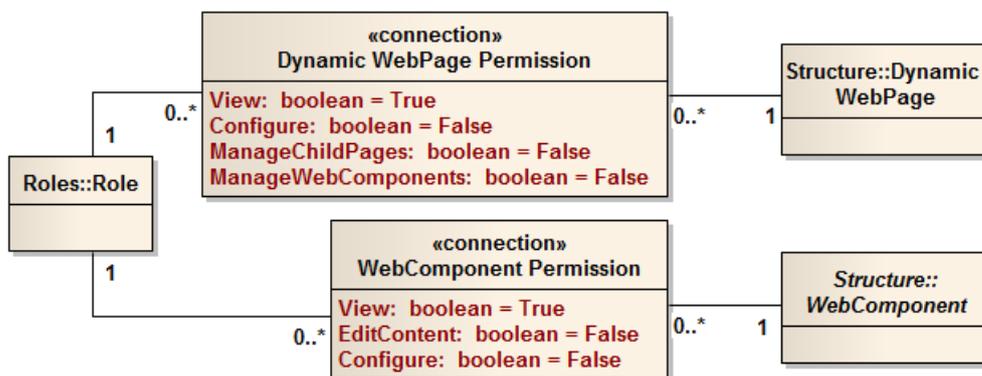


Figure 7.9: Abstract syntax for the Website Template’s Permissions view.

Before proceeding with the description of the Permissions view, it is important to make a distinction between the terms “configuration” and “management” in the context of this view. This distinction is as follows: (1) **configuring** a **Dynamic WebPage** or a **WebComponent** consists of setting its *internal* characteristics (i.e., *its properties*), such as its name or its friendly URL, to new values; and (2) **managing** a **Dynamic WebPage** does

not include configuring it but rather changing some/all of its *external* characteristics (i.e., *its associations to other elements*), such as the **WebComponents** that are included within the **Dynamic WebPage**.

A **Dynamic WebPage Permission** determines the actions that a **Role** can perform over a **Dynamic WebPage**. It defines permissions for viewing and configuring the **Dynamic WebPage**, as well as managing its **WebComponents** and child pages. On the other hand, a **WebComponent Permission** determines the actions that a **Role** can perform over a **WebComponent**. It also defines permissions for viewing the **WebComponent**, as well as configuring it and editing its contents.

Each of the defined permissions has a default value, obtained from what is typically allowed for any user that browses a website (namely, viewing permissions default to **True**, while editing, configuration, and management permissions are **False** by default).

The Permissions view can be represented by two alternative manners, either a graphical manner or a set of permission matrices. The graphical manner is more adequate for conveying simpler sets of permissions, while the matrix representation enables the representation of larger sets of permissions in a more compact and efficient manner.

Regarding the representation of the Permissions view via a set of permission matrices, this view can be represented by: (1) a matrix containing the permission mappings between the **Roles** and **Dynamic WebPages**, which is called *Page Permission Matrix*; and (2) for each **Dynamic WebPage**, a matrix containing the permission mappings between the **Roles** and the page's **WebComponents**, called *WebComponent Permission Matrix*.

The permission values are specified with checkmarks or crosses, depending on whether the value is **True** or **False**, respectively. The names and values of the Permission's attributes (e.g., **View**, **Configure**) need to be explicitly specified only if they are not set to their default value, otherwise they can be omitted (i.e., left "blank").

Figure 7.10 depicts some examples of the Permissions view's concrete syntax:

- in Figure 7.10a, a graphical representation of a **Dynamic WebPage Permission** with some non-default values;
- in Figure 7.10b, a graphical representation of a **WebComponent Permission** (also with some non-default values);
- in Figure 7.10c, a matrix representation of a set of **Dynamic WebPage Permissions** (once again, the values that are not represented assume their default value); and
- in Figure 7.10d, a matrix representation of a set of **WebComponent Permissions**.

In this view, **Dynamic WebPages** are represented with the same syntax as in the Macro Structure view, while **WebComponents** are represented as in the Micro Structure view (although it is not necessary to represent their details, such as order within the parent), in order to facilitate the reading of the view.

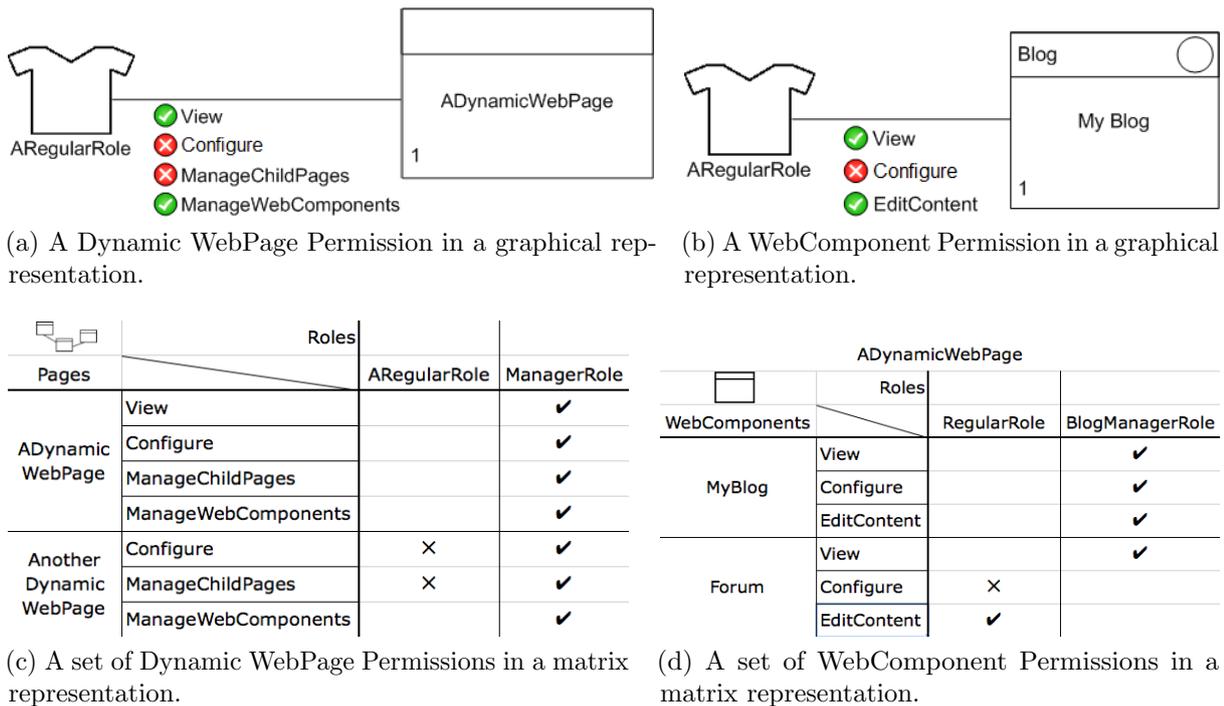


Figure 7.10: Concrete syntax for the Website Template's Permissions view.

7.5 Toolkit Modeling

A Toolkit can be regarded as a *task-oriented extension of CMS elements*, as it enables the addition of new CMS-related concepts (namely **Roles** and **WebComponents**) that are oriented toward supporting a particular set of tasks and the corresponding domain model. A Toolkit is defined according to the following set of views (which are illustrated in Figure 7.11):

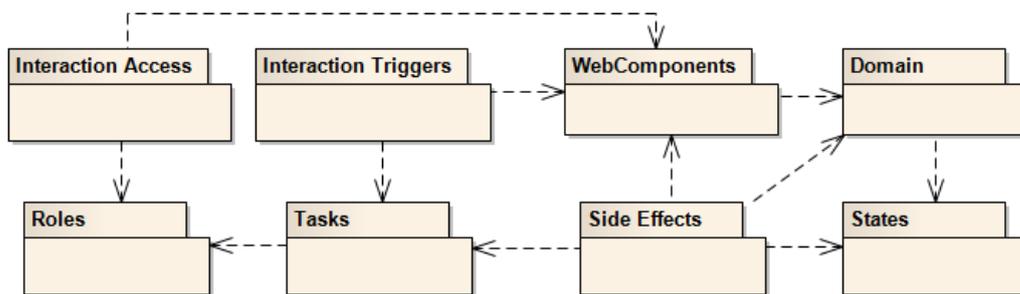


Figure 7.11: Views involved in the definition of a Toolkit.

- The *Roles view*, specifying the roles that will be responsible for performing the Toolkit's tasks;
- The *Tasks view*, representing the user tasks to be supported by the Toolkit;
- The *Domain view*, which specifies the domain model underlying those tasks;

- The *States view*, dealing with the lifecycle of the entities that the tasks are to manipulate;
- The *WebComponents view*, specifying the **WebComponents** to support the tasks;
- The *Side Effects view*, which enables the definition of desired side effects (regarding the defined domain model) for the modeled user tasks and **WebComponents**;
- The *Interaction Access view*, which establishes mappings between the defined **Roles** and **WebComponents** (i.e., what **Roles** can access what **WebComponent** elements); and
- The *Interaction Triggers view*, which establishes triggers between **WebComponents** and user tasks (i.e., what **WebComponent** elements will trigger the execution of user tasks or other transitions).

The Tasks, Roles, and WebComponents views are the most important in a Toolkit. The Tasks view allows the Toolkit Designer to define user tasks as orchestrations of steps which may involve user interaction (in a manner similar to UML’s Activity diagrams). The Roles view (not directly related to CMS **Roles**) models the different kinds of responsibilities (i.e., roles) that the web application should consider. Finally, the WebComponents view is where the Toolkit’s UI is specified, by creating complex UI structures from simpler ones (e.g., button, image). This variety of web interface elements allows the modeling of relatively complex web interfaces using CMS-ML.

7.5.1 Roles View

In a manner very similar to the WebSite Template’s Roles view, this view describes the various kinds of behavior that are expected by the *tasks* defined in the Toolkit, as well as the relationships between those behaviors. These are modeled by the **Role** and **Role Specialization** concepts, respectively. Figure 7.12 illustrates this view’s abstract syntax.

Unlike the Template’s **Role Delegation** concept, **Role Specialization** models specialization relationships between **Roles**, as the name indicates. In other words, it is possible to specify that a **Role A** is a *particular case* of a **Role B**, although the reverse is not true. Note that this is not the same as delegating responsibilities (which is the purpose of **Role Delegation**): a specialization is typically meant as a *permanent* relationship between **Roles** (if a **Role R_B** is a specialization of a **Role R_A**, and *b* is an instance of **R_B** – and thus of **R_A** –, it would make no sense for *b* to *not* be an instance of **Role R_A** at another point in time), while a delegation often expresses a *temporary* relationship.

It is important to remember that Toolkit **Roles** are not modeled in the same manner as WebSite Template **Roles**, because these two concepts are located in different metalevels (as was explained in Section 7.3). The only relationship between the Template **Role** and Toolkit **Role** concepts, indicated in Figure 7.12, is that each modeled Toolkit **Role** will

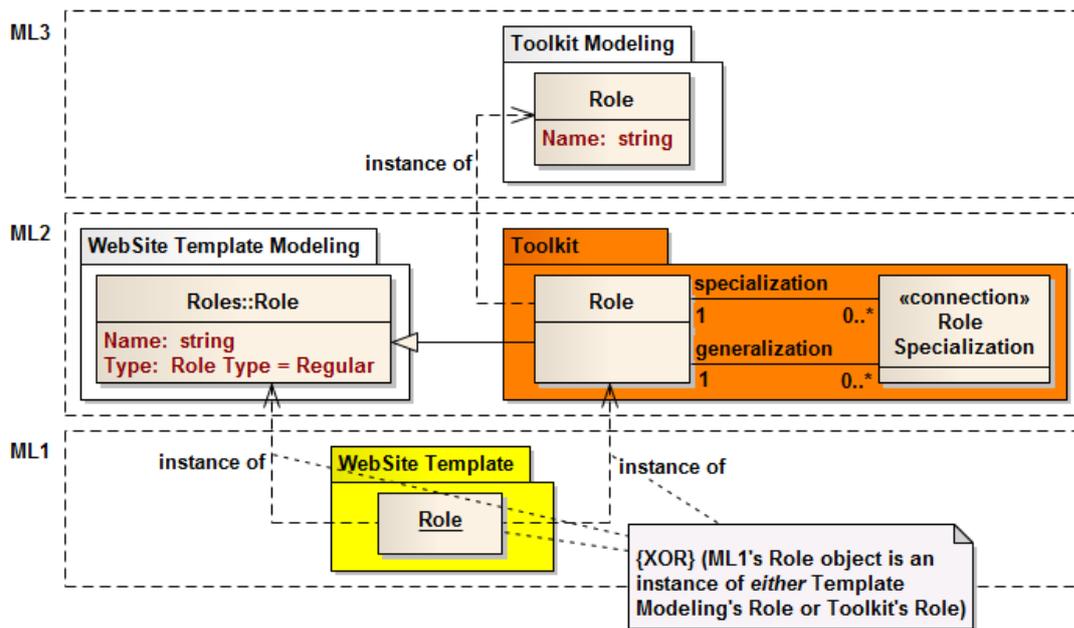


Figure 7.12: Abstract syntax for the Toolkit's Roles view.

actually be a specialization of the Template `Role` concept (which, in turn, will be modeled as a concrete `Role` in the WebSite Template). Thus, when modeling the WebSite Template Roles view, each `Role` will always be considered as an instance of the WebSite Template's `Role` concept, but it may actually be an instance of a Toolkit `Role` that will participate in some of the Toolkit's tasks. However, because the Toolkit is modeled in a different metalevel than the WebSite Template, it is not possible to use Toolkit `Role` instances and WebSite Template `Role` instances in the same model, nor is it possible to model `Role Delegations` in the Toolkit Roles view (this concept would not make sense in Toolkit modeling anyway, as the set of tasks to perform should depend only on `Roles` and not on any hierarchical delegation relationships between them).

Figure 7.13a depicts the concrete syntax for the `Role` concept: its representation is very similar to a Template `Role`'s but, because of the aforementioned conceptual difference between these concepts, there is no danger of designers erroneously using one concept instead of the other. On the other hand, Figure 7.13b illustrates the concrete syntax for the `Role Specialization` concept (it is represented in the same manner as UML's `Generalization` concept, for obvious reasons).

7.5.2 Tasks View

This view is fundamental for the Toolkit Designer (and is perhaps the most important in a Toolkit model), because it allows the specification of the various tasks that the Toolkit should support (which, in turn, are the Toolkit's *raison d'être*). The most important

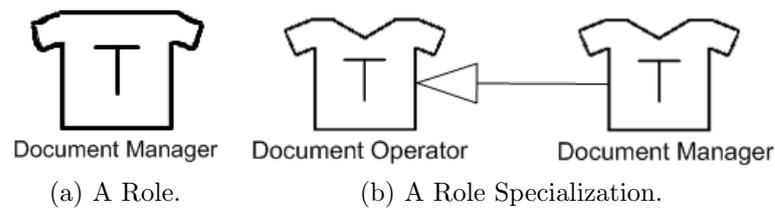


Figure 7.13: Concrete syntax for the Toolkit's Roles view.

concepts in this view are **Task** and **Action**, which represent the task to be performed, and the various actions that will be necessary to perform that task, respectively. Figure 7.14 depicts the abstract syntax for the Tasks view.

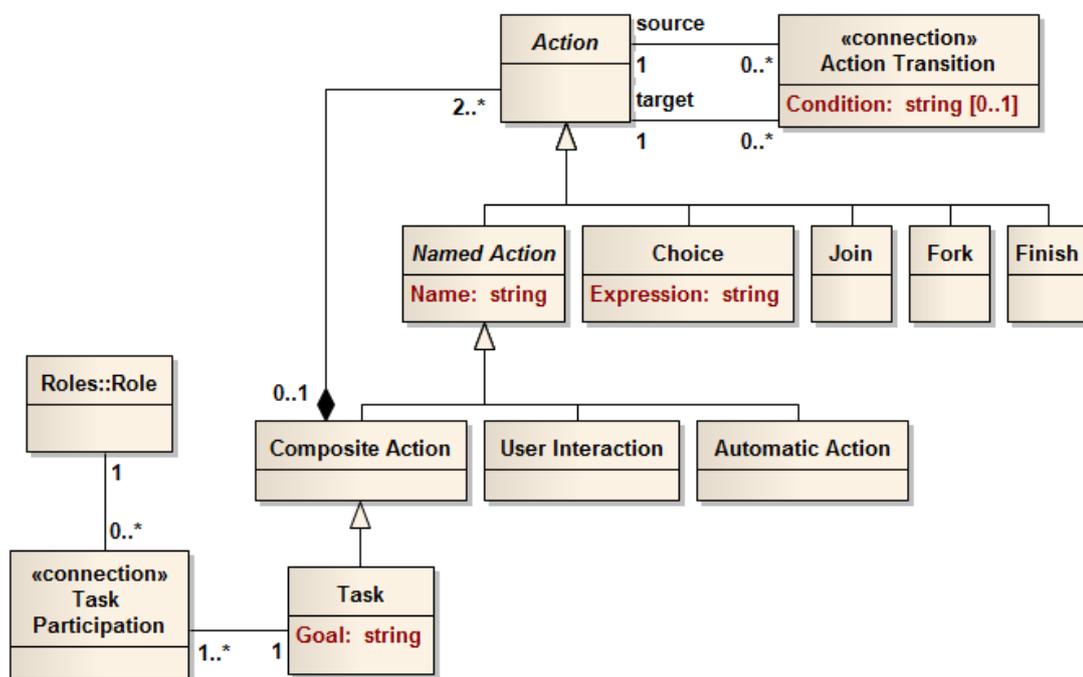


Figure 7.14: Abstract syntax for the Toolkit's Tasks view.

The Toolkit Designer starts modeling the Toolkit by defining its supported tasks; this is to try and ensure some alignment between the Toolkit itself and the *reason* for creating the Toolkit. A **Task** is used to describe its goal and the manner in which the interaction between a user and the Toolkit should be performed. Furthermore, a **Task** must be performed by someone, a fact modeled by the **Task Participation** concept, which establishes a relationship between the **Task** to be performed and each of the **Roles** that are to play some part in performing that **Task** (i.e., the **Roles** that participate in it).

Nevertheless, it is important to emphasize that a **Task** is not just as a structured collection of **Actions**. In fact, a **Task** can be regarded as a holistic entity, not only composed of a sequence of steps that will be executed by the user(s) performing the **Task**,

but also serving as the main driver for the interaction between the web application itself and its users.

An **Action** can be considered as a *unit of work* that is necessary to complete a **Task** (e.g., one of the **Actions** in a **Task View Document Contents** can be **Select Document to view**). There are different types of **Action** available in CMS-ML, namely:

User Interaction An **Action**, identified by the symbol , that requires some kind of interaction with the user (e.g., show a message to a user, requiring that the user press an OK button).

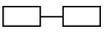
Automatic Action An **Action**, represented by the symbol , that is performed automatically by the system, without requiring any interaction with the user. An example can be the sending of an e-mail (assuming that the e-mail's parameters – sender, destination, contents – have already been specified in previous **Actions**).

Choice An **Action** expressing a set of alternatives regarding what **Actions** to perform. This path is determined by the attribute **Expression**, which can be defined in any matter (e.g., a natural language statement – *if we have a good enough search match* – or a logical expression – $\text{searchResultRanking} \leq 10$ – are valid expressions).

Fork An **Action** that splits the **Task's** flow into several other flows (which, in turn, can be performed in a concurrent manner).

Join An **Action** that receives multiple **Action** flows (which are being performed concurrently) and joins them into a single flow. In practice, a **Join** means “wait for all the selected concurrent **Actions** to finish, and then proceed”.

Finish An **Action**, represented as , that specifies the *ending* of the **Task**, although it does not necessarily mean a successful ending (i.e., the **Task** may have ended prematurely because of an error). The usage of the **Finish** element indicates that the **Task** is over, and thus no further **Actions** will be performed in the context of that **Task**.

Composite An **Action**, identified by the symbol , with the sole purpose of aggregating other **Actions**. It is typically used as a way to organize **Actions**, by grouping related finer-grained **Actions** into a coarser-grained **Action** (e.g., a set of **Actions**, **Enter Author Name** and **Enter Author Address**, can be grouped into a **Composite Action Specify Author**). Aside from this grouping usage, the **Composite Action** presents no functional added-value whatsoever, and so it is possible to define any **Task** without using them.

Some of these are also **Named Actions** because they have a name (a small statement regarding *what* the **Action** will do); this name is used at a later stage, when CMS-IL designers are editing the corresponding CMS-IL model (according to the approach described

in Chapter 6), as those designers will then use that name – describing the work that must be done – to specify *how* the **Action** is performed.

Actions are linked to each other by an **Action Transition**, a directed link between two **Actions** – the source and the target – that indicates the control flow between them.

Figure 7.15 depicts an example of the concrete syntax for the Tasks view (for simplicity, only some modeling elements are used in this example). More specifically, the figure presents a very simple Task, named **Mark Document as Invalid**. This Task, which is performed by the **Document Manager Role**, contains the following **Actions** (in order, from left to right and top to bottom):

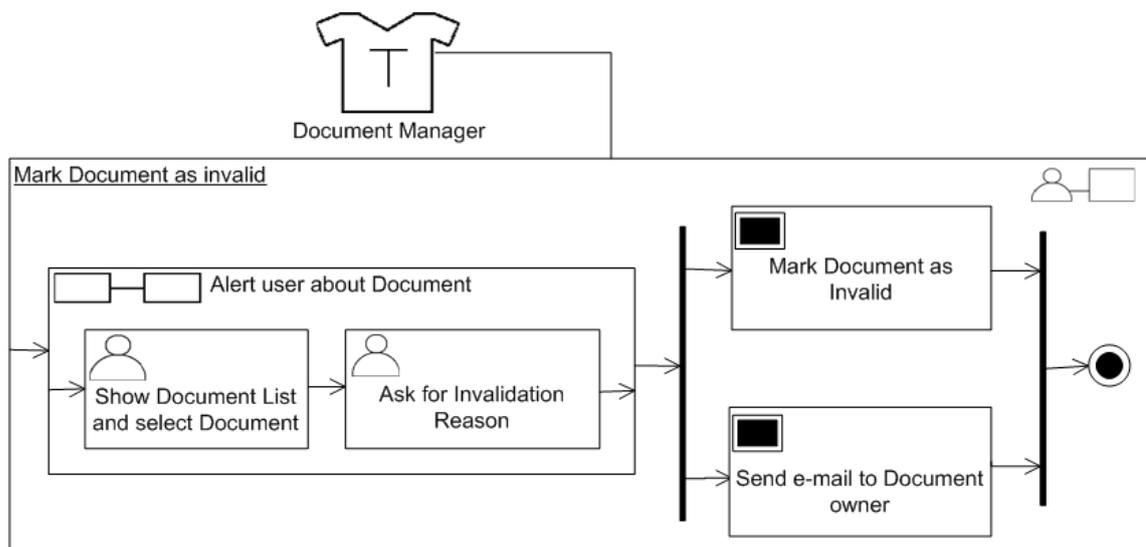


Figure 7.15: Concrete syntax for the Toolkit's Tasks view.

- The Task begins with a **Composite Action**, **Alert user about Document**, which will group a set of **Actions** that are related to the intent expressed in its name;
- This **Composite Action** itself begins with a **User Interaction** element, called **Show Document List and select Document**, which indicates (through its Name) that a list of documents is to be displayed to the user and that the user should select a document from the list;
- When **Show Document List and select Document** completes, it proceeds to another **User Interaction** called **Ask for Invalidation Reason**, in which the system asks the user for an invalidation reason. The **Composite Action** itself is completed when **Ask for Invalidation Reason** is completed;
- The **Composite Action** is followed by a **Fork**, indicating that the execution of the following **Actions** will be performed in parallel;
- After the **Fork**, we have an **Automatic Action** called **Mark Document as Invalid**, in which the system will mark the selected document as invalid;

- *While* the previous **Action** is being performed, another **Automatic Action** (**Send e-mail to Document owner**) will send an e-mail to the document's owner;
- After the previous two **Actions** are completed, a **Join Action** indicates that any following **Actions** will *not* be performed in parallel;
- The **Task** is completed when it reaches the **Finish Action**.

It should be noted that the Tasks view was heavily based on UML's Activity diagrams [OMG 11_e], and it is also loosely based on some concepts defined by ConcurTask Trees [PMM 97, MPS 02, Pat 03, GSSF 04]. The UML Activity diagram already defines most of the modeling elements necessary to specify flowchart-like sets of operations (a perspective that most non-technical stakeholders typically consider to be intuitive). On the other hand, ConcurTask Trees define elements that enable the modeling of human-computer interaction, namely with the specification of collaborative actions (e.g., *user interaction* actions that involve interaction between a user and a computer, and *fully automated* actions that are carried out entirely by the computer). We consider that this Tasks view, based on the control flow modeling of UML Activity diagrams and including a few task-oriented elements of ConcurTask Trees, allows Toolkit Designers to better model the desired Toolkit tasks.

Also, the Tasks view does not provide a mechanism for error-handling. This is deliberate because, from our own observations while building the language: (1) most stakeholders are usually not concerned about errors at this stage of the modeling process; (2) when modeling an error-handling mechanism, and in order to streamline the modeling process, stakeholders typically choose to *show the problems that occurred, and go back to the beginning of the task*); and (3) adding error-handling operators to this view would potentially complicate Task models more than necessary.

7.5.3 Domain View

The Domain view is where the Toolkit's underlying domain model is specified. It is very similar to UML's Class diagrams [OMG 11_e] and WebML's Data Model [CFB⁺ 02], and consists of specifying entities and relationships between them. Figure 7.16 depicts this view's abstract syntax.

The most important concept in the Domain view is the **Entity** concept, which is used to model the entities (e.g., **Person**, **Credit Card**, **Document**) that will be manipulated by the users of web applications where the Toolkit is used. An **Entity** can be *abstract*, in which case it is not possible for a CMS system to create runtime instances of that **Entity**; it should be noted that these semantics are also found in regular Object-Oriented Programming (OOP) languages, namely regarding the definition of *abstract classes*.

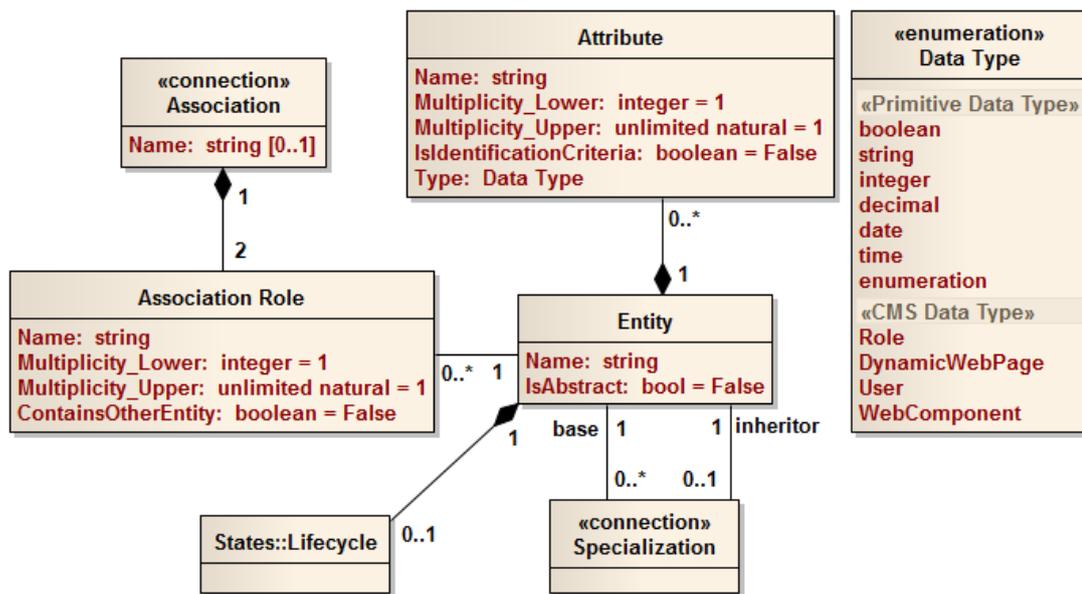


Figure 7.16: Abstract syntax for the Toolkit's Domain view.

An **Entity** can have a set of fields, called **Attributes**. Whenever an instance of an **Entity** is created, corresponding instances for its **Attributes** will also be created, used to store values that constitute the instance's state (e.g., if a person has paid its monthly subscription for a magazine). If we consider that an example of **Entity** could be **Person**, then typical **Attributes** could be **Name**, **Date of birth**, or **Address**.

Furthermore, each **Attribute** must have a certain **Data Type**, which is an indication of the kind of value to be stored. CMS-ML defines two different kinds of **Data Type**, (1) **Primitive Data Types** and (2) **CMS Data Types**. A **Primitive Data Type** is a kind of element that is usually provided by any software system (e.g., boolean values, strings, integers, dates). On the other hand, a **CMS Data Type** is used to represent the identification criteria for a *concrete instance of a CMS concept* (e.g., the **User Data Type** can be used to store whatever information is necessary to uniquely identify a concrete user in the CMS system). An easy way to distinguish between these two kinds of **Data Types** is that the names of **CMS Data Type** always start with an uppercase character (e.g., **User**), while **Primitive Data Type** names always start with a lowercase character (e.g., **string**).

On the other hand, the **Association** concept enables the modeling of relationships between **Entities**. CMS-ML only supports *binary* **Associations** (i.e., **Associations** between two **Entities**), although the two related **Entities** can actually be the same, enabling reflexive relationships. An **Association** contains exactly two **Association Roles**, one for each of the associated **Entities** E_1 and E_2 . The **Association Role** is what actually links each **Entity** to the **Association**, and determines the part that the

linked Entity will play in the Association – or, from Entity E_1 's perspective, the role of Entity E_2 in relation to itself (and vice versa). Each Association Role also defines multiplicity, and whether it contains composition semantics (i.e., if its Entity is a container for the Entity of the opposite Association Role).

Another important concept, **Specialization**, allows Toolkit Designers to specify *inheritance* – generalization or specialization, depending on the point of view – between two Entities, the base and the inheritor. The specification of an inheritance hierarchy between Entities allows (1) the inheritor Entity to be considered as a particular case (or *specialization*) of the base Entity, and (2) the Attributes of the base Entity to also be available to the inheritor Entity. The Specialization concept does not support multiple inheritance (i.e., for an inheritor Entity to have more than one base Entity), in order to avoid possible conflicts coming from Attributes or Association Roles with the same name in different base Entities.

An Entity can also have a Lifecycle (a possible set of stages through which it will pass during its lifetime) associated with it, although this is optional. An Entity can inherit a lifecycle from its base Entity, although it can also override the inherited lifecycle by defining another one. CMS-ML enables the explicit definition of an Entity's lifecycle via the States view, which is described next.

Figure 7.17 provides some examples of the Domain view's concrete syntax, namely of Entities and the possible relationships between them. Entities that contain the symbol  have a Lifecycle defined, and abstract Entities are identified by the letter A on their top-right corner.

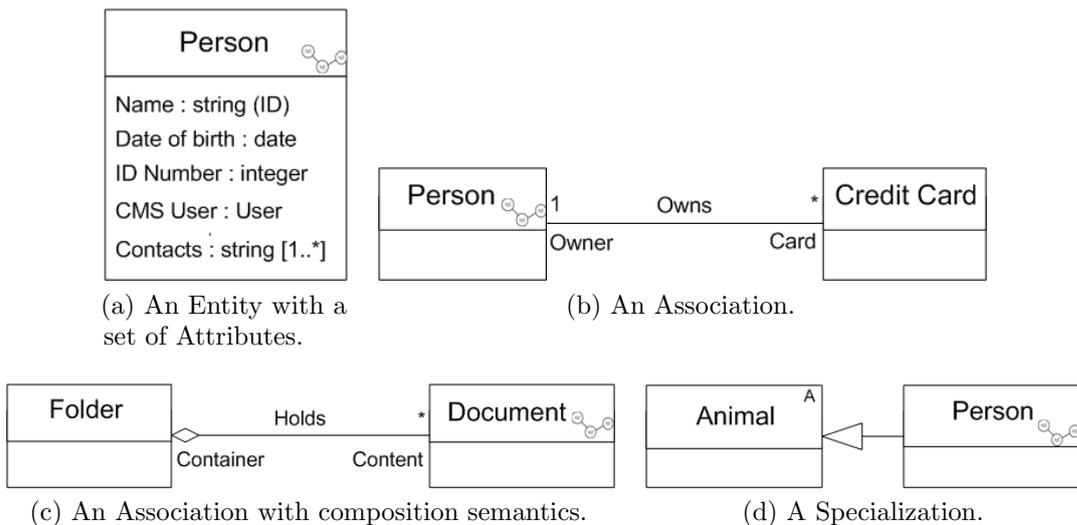


Figure 7.17: Concrete syntax for the Toolkit's Domain view.

As previously mentioned, the Domain view is similar to UML's Class diagram and WebML's Data Model. However, the reason why this view defines so few modeling

elements – much like WebML and unlike UML – is that most of UML’s class diagram modeling elements (e.g., **Dependency**, **Realization**), although helpful in *describing* domain models, do not provide enough added-value to justify their inclusion in this language (which is not intended to have a great number of modeling elements).

CMS-ML does impose some semantic restrictions on how domain modeling should be performed (e.g., there cannot be two **Attributes** with the same name within the same **Entity**), in order to ensure the structural and semantical correctness of those models. However, we do not elaborate on such restrictions in this dissertation, because of their extensive length; interested readers are nevertheless invited to consult the “User’s Guide” [SS 10_c] for further information.

7.5.4 States View

The States view enables the specification of the lifecycles for the Domain view’s **Entities**. More specifically, these are defined by specifying a **Lifecycle** for each **Entity** that has a lifecycle. A **Lifecycle** consists of a set of **States** and **State Transitions** between them, and is modeled in the same manner as a state machine is traditionally modeled (e.g., like UML’s **State Machines**, albeit CMS-ML uses a smaller palette of modeling elements). Figure 7.18 illustrates the abstract syntax of this view.

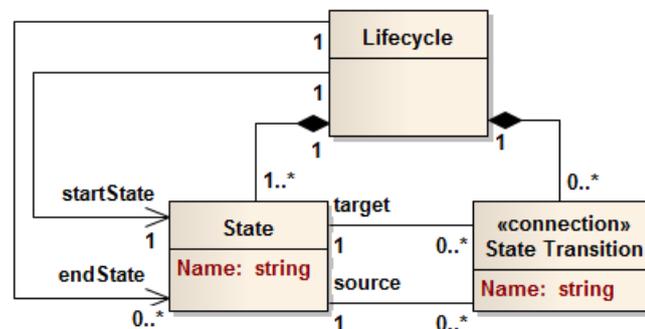


Figure 7.18: Abstract syntax for the Toolkit’s States view.

A **Lifecycle** can be considered as an aggregator for a set of **States** and **State Transitions**. A **State** represents a specific stage in the **Entity**’s lifecycle (e.g., the **State Awaiting approval** can be used to represent the point in a **Document**’s lifecycle in which it is awaiting approval by some moderator). On the other hand, a **State Transition** establishes a possible flow between two **States**, the source and the target. This enables the specification of what **States** in the **Entity**’s **Lifecycle** can be reached from a particular **State**.

A **Lifecycle** has a start **State** and a set of end **States**, identified by the symbols ● and ○, respectively. When an instance of the corresponding **Entity** is created, it will be

in the start **State**. End **States** indicate that the **Entity**'s instance is no longer important to the system, and so it may be discarded. Because it may be possible for an **Entity** to never become unnecessary (e.g., for accountability purposes), specifying the ending **State(s)** is optional.

Figure 7.19 provides two examples of the concrete syntax for this view. More specifically, Figure 7.19a shows the **Lifecycle** for an **Entity Folder**, containing two **States** (**Exists** is the start **State** and **Deleted** is an end **State**) and a **State Transition** between them. On the other hand, Figure 7.19b illustrates the **Lifecycle** for an **Entity Document** containing three **States** (including the start **State** **Awaiting Approval** and an end **State** **Deleted**), and some **State Transitions** between them.

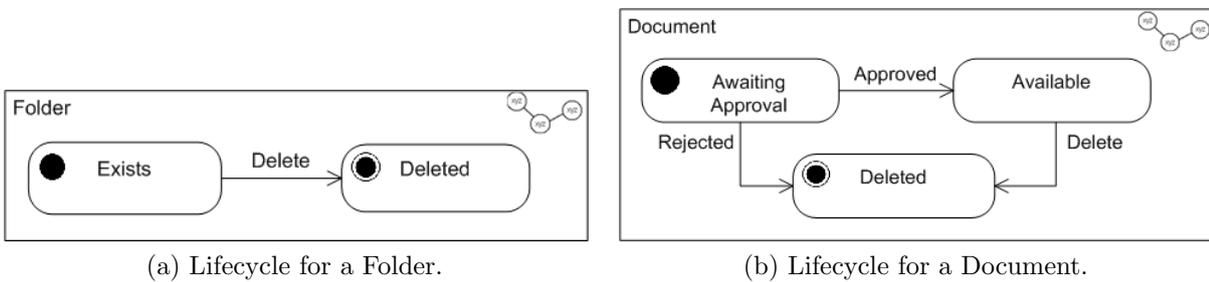


Figure 7.19: Concrete syntax for the Toolkit's States view.

It is important to emphasize that it is not necessary for an **Entity** to have a **Lifecycle** defined. If no **Lifecycle** is specified for an **Entity**, then the **Entity**'s instances will simply continue existing (one of the consequences of this is that they will never be discarded).

7.5.5 WebComponents View

One of the main goals of Toolkit modeling is to allow Toolkit Designers to define new kinds of **WebComponent** that support the user tasks defined in the **Tasks** view. The **WebComponents** view enables this goal, by providing Toolkit Designers with the possibility of modeling **WebComponents** and **Support WebPages**. Figure 7.20 illustrates this view's abstract syntax¹. As depicted in the figure, the most important concepts in this view are **WebElement**, **WebComponent**, and **Support WebPage** (for simplicity, we will often refer to these elements just as *visual elements*, because they convey elements that users will actually see and interact with).

Modeling this view starts with the definition of a **WebComponent** (sometimes called *Toolkit WebComponent* in this dissertation, to remove possible ambiguity between the

¹Some of the concepts shown, such as **Binding** and **Expected Entity**, will not be explained in this dissertation. However, interested readers are invited to consult the "User's Guide" [SS 10_c].

Toolkit **WebComponent** will be a specialization of CMS **WebComponent**. Thus, when modeling the WebSite Template's Structure view, each **WebComponent** may be an instance of a Toolkit **WebComponent**, but it will always be an instance of the WebSite Template's **WebComponent** concept (because the specialization relationship is transitive). However, because the Toolkit is modeled in a different metalevel than the WebSite Template, it is not correct to use instances of Toolkit **WebComponent** and instances of WebSite Template **WebComponent** in the same model, nor is it correct to model (in this view) relationships taking place between Toolkit **WebComponent** instances and elements of a Template's Structure view.

On the other hand, a **Support WebPage** is used to model a web page (not to be confused with a **Dynamic WebPage**) that supports a specific **WebComponent** in parts of the **Task(s)** that it addresses; a typical example of a **Support WebPage** would be a page to edit details regarding a certain **Entity**. A **Support WebPage** is modeled as an orchestration of simpler **WebElements** (explained next), and will typically correspond to a dynamic HTML page that receives some parameters and performs a corresponding set of actions.

In order to support user tasks, **WebComponents** and **Support WebPages** must contain *elements* (e.g., buttons, images, text boxes) that users will be able to see and interact with. These are captured by the **WebElement** abstract concept, which represents something that will be shown in the web browser and with which the user may be able to interact.

Some **WebElements** are simple (i.e., they have relatively simple semantics) and are frequently used when designing web applications. These simple elements are captured in CMS-ML by the **Simple WebElement** concept, which consists of a **WebElement** with a predefined type, namely: (1) **Button**; (2) **Link**; (3) **Image**; (4) **Text**; (5) **Text Input Box**; (6) **List Box**; and (7) **Selection Box**.

Another element frequently used in web applications is the HTML string, with purposes such as labeling other elements (e.g., input elements) or displaying relevant text on the web browser. This leads to the definition of the **HTML WebElement**, which is a **WebElement** corresponding to some HTML manually specified by the Toolkit Designer.

Finally, it is possible to define **WebElements** that *contain* other **WebElements**. These are modeled with the **WebElement Container** concept, a **WebElement** container that can also be used to organize semantically related **WebElements**. CMS-ML defines the following types of **WebElement Container**: (1) **Normal**, corresponding to a typical section of a page (and typically represented using an HTML `div` element); (2) **Popup**, which specifies a container that will appear only as a *popup window* when the user performs some action that should make the popup visible; (3) **List**, which enables the display of a set of **Entity** instances as a non-tabular list, according to the arrangement of its contained **WebElements**; (4) **Table**, which displays a set of **Entity** instances in a tabular manner;

and (5) **Binding**, a container that is used *solely* for the purpose of creating or refining **Bindings**, a topic that will not be discussed in this dissertation (due to text length).

Figure 7.21 provides two examples of the concrete syntax for this view, namely of a **WebComponent** and a **Support Page**, respectively identified by the symbols  and . Specifically, Figure 7.21a represents a **WebComponent** **Manage Documents**, composed of: (1) a **List WebContainer**, **Document List**, containing only a **WebElement** **Name** of type **Text**; (2) a **Create Document Link**; (3) an **Edit Document Button**; and (4) a **Delete Document Button**. On the other hand, Figure 7.21b represents a **Support WebPage** **Edit Document**, which supports the **Manage Documents WebComponent**, and is composed of: (1) a **Name Text WebElement** and a corresponding **NameValue Text Input Box**; (2) a **Description Text WebElement** and a **DescriptionValue Text Input Box**; (3) a **Cancel Button**; and (4) a **Confirm Button**.

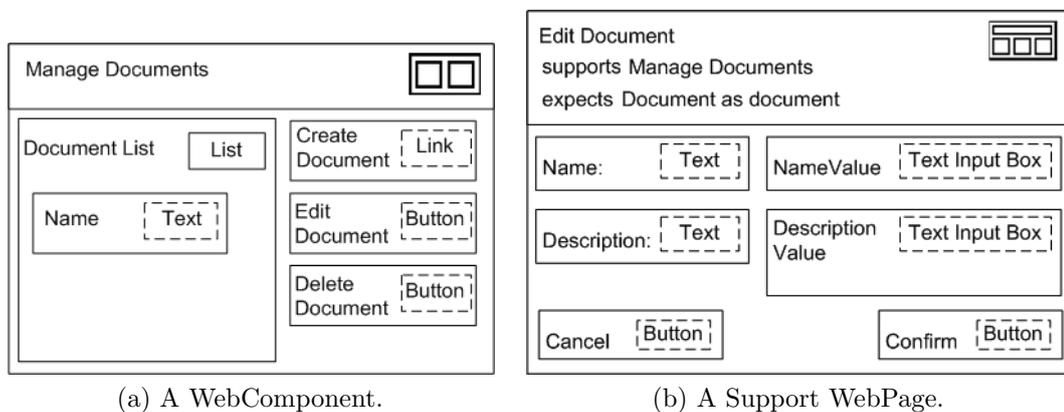


Figure 7.21: Concrete syntax for the Toolkit's WebComponents view.

As Figure 7.21 shows, the orchestration of **WebElements** is done in a graphical manner: **WebComponent** and **Support WebPage** provide a *canvas* (like the **Dynamic WebPage** in the **WebSite Template's Micro Structure** view) in which the contained **WebElements** are represented. This allows non-technical stakeholders to design **WebComponents** and **Support Pages** in a WYSIWYG manner.

7.5.6 Side Effects View

This view allows Toolkit designers to specify *side effects* that will take place when some event occurs. Such events can be **Action Transitions** or even clicks on a **WebElement**. Figure 7.22 depicts the abstract syntax of this view.

The main concept in this view is the **Side Effect**, consisting of an ordered set of effects (called **Operations**) that take place when a specified event occurs. Possible events that can trigger side effects are: (1) occurrence of an **Action Transition**; (2) starting an **Action**;

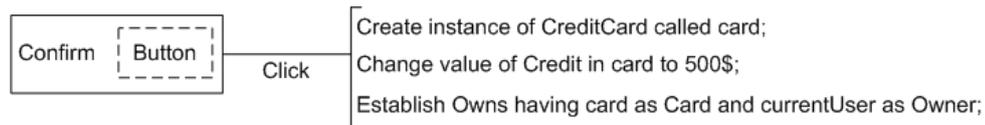


Figure 7.23: Concrete syntax for the Toolkit's Side Effects view.

2. A **Change Value Operation** that takes the `card` instance (created in the previous **Operation**) and changes the value of its `Credit` **Attribute** to 500\$; and
3. An **Establish Association Operation** that creates an instance of the **Association Owns**, between the aforementioned `card` (an instance of `CreditCard`) and `currentUser` (an instance of `Person`, and in this particular case assumed to be provided by the context in which the **Side Effect** occurs).

7.5.7 Interaction Access View

The Interaction Access view establishes access mappings between the **WebComponents** and **Roles** views, namely between **Roles** and the visual elements (**WebElements**, **WebComponents**, and **Support WebPages**) with which they can interact, in a manner similar to the **WebSite Template's Permissions** view. Figure 7.24 depicts the abstract syntax for this view.

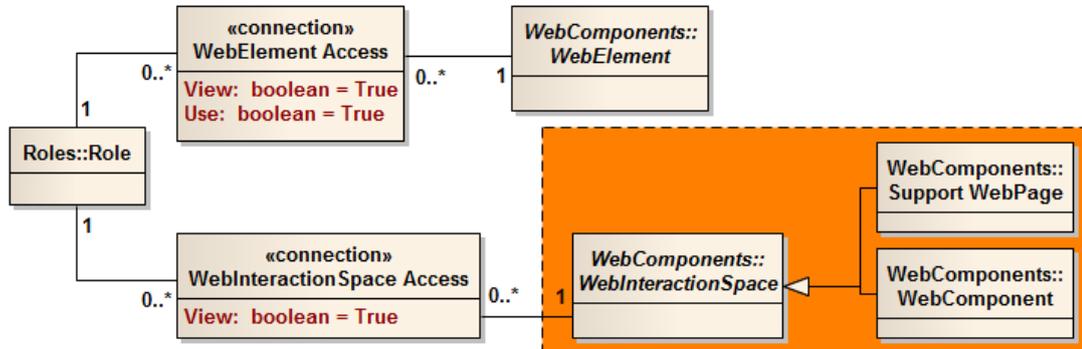


Figure 7.24: Abstract syntax for the Toolkit's Interaction Access view.

Access to a **WebElement** is established via the **WebElement Access** concept, which is used to specify Toolkit **Role-WebElement** access permissions and indicate the **WebElements** which the **Role** can (or cannot) interact with. More concretely, it defines access permissions for *viewing* and *using* the **WebElement**, each of which has a default value (**True**) that only requires Toolkit Designers to specify what **Roles** are *not allowed* to view or interact with the element.

On the other hand, access to **WebComponents** and **Support WebPages** is set with the **WebInteractionSpace Access** concept. Unlike **WebElement Access**, this concept defines only access permissions for *viewing* the **WebComponent** or **Support WebPage**; if

any user without this access permission attempts to view the `WebComponent` or `Support WebPage`, the CMS should return an “Access denied”-like message (or an empty response, depending on the CMS implementation itself).

Figure 7.25 depicts some examples of the concrete syntax for the `WebInteractionSpace Access` element (the `WebElement Access` element is represented in the same manner). Figure 7.25a and 7.25b also illustrate the fact that, much like the `WebSite Template’s Permissions` view, the `WebElement Access` and `WebInteractionSpace Access` elements can be represented either graphically or by means of matrices.

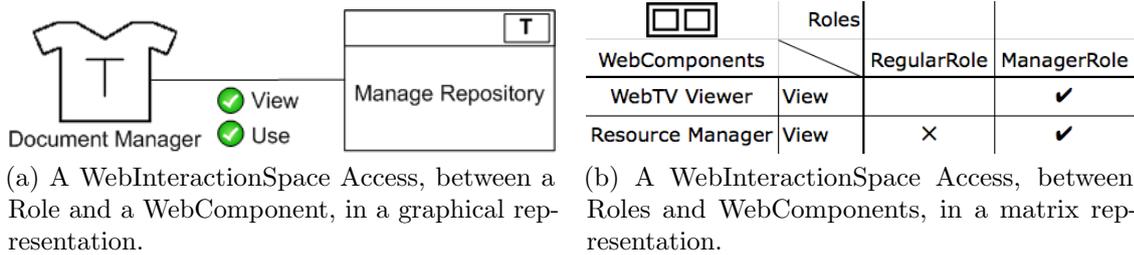


Figure 7.25: Concrete syntax for the Toolkit’s Interaction Access view.

7.5.8 Interaction Triggers View

The Interaction Triggers view is used to establish mappings between the `WebComponents` and `Tasks` views. The considered mappings take place between: (1) `WebElements` and the `Tasks` that they will initiate, or the `Action Transitions` that they will trigger; and (2) `Task Actions` and the visual elements that they should be displayed on. Figure 7.26 depicts the abstract syntax for this view.

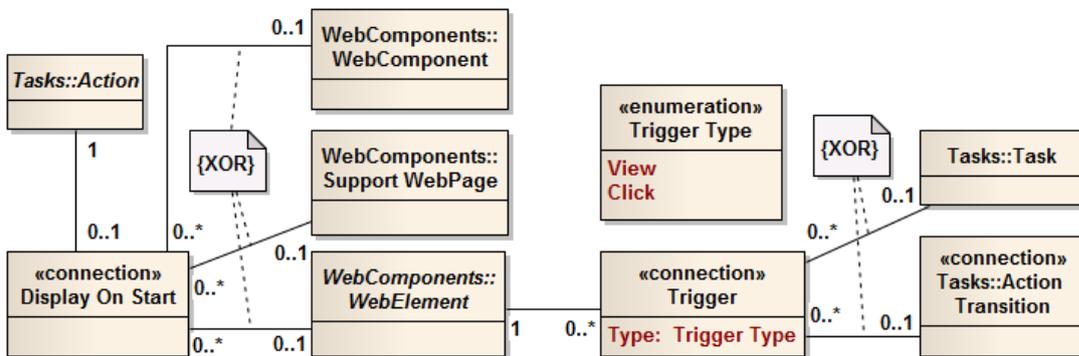


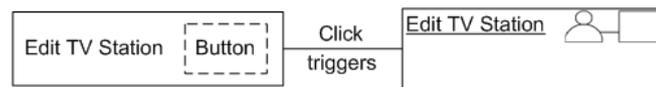
Figure 7.26: Abstract syntax for the Toolkit’s Interaction Triggers view.

The **Trigger** concept consists of a relationship between a `WebElement` and either a `Task` or an `Action Transition`. It allows Toolkit Designers to specify that interaction

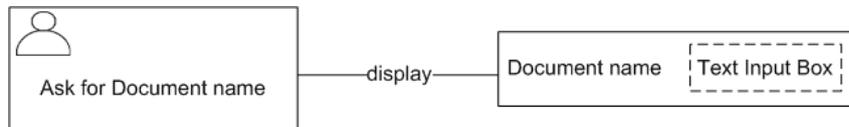
with the `WebElement` will *trigger* some event (depending on the element to which the `Trigger` is connecting the `WebElement`): (1) if the `WebElement` is connected to a `Task`, interaction will initiate the `Task`; (2) otherwise, if the `WebElement` is connected to an `Action Transition`, interaction will cause the `Action Transition` to be triggered (and thus the `Task`'s current `Action` will flow to the next `Action`). A `Trigger` has a certain type, defined by the `Trigger Type` enumeration, indicating the kind of interaction (between the user and the `WebElement`) that must occur in order to trigger the event. There are two kinds of trigger type: (1) `View`, indicating that merely viewing the `WebElement` is enough to trigger the event; and (2) `Click`: indicating that the user must click on the `WebElement` (or take a similar action) to trigger the event.

On the other hand, the `Display On Start` concept is used to indicate that, when its associated `Action` is started, a certain visual element must be displayed to the user (specific details, such as the `WebComponent` instance to display, are left for the CMS system to decide).

Figure 7.27 depicts some examples of this view's concrete syntax.



(a) A Trigger.



(b) A Display On Start.

Figure 7.27: Concrete syntax for the Toolkit's Interaction Triggers view.

7.6 Website Annotations Modeling

As was mentioned earlier, the Website Annotations model (which uses the concepts defined in Website Annotations Modeling metamodel) allows Website Template Designers to “add” **Annotations** (representing tags, properties, or general constraints) to a Website Template. These annotations can convey any kind of information, such as content configuration options (e.g., a tag `Allows Content Subscription` applied to a `WebComponent` can indicate that it should provide readers with an RSS feed or similar mechanism) or deployment platform-specific data, such as a canonical home address for the `Website`. Unlike **Additional Features** (which are explained in Section 7.7), these annotations are meant to be *interpreted by the CMS system* in which the model is deployed. Figure 7.28 illustrates the concepts defined by the Website Annotations Modeling metamodel.

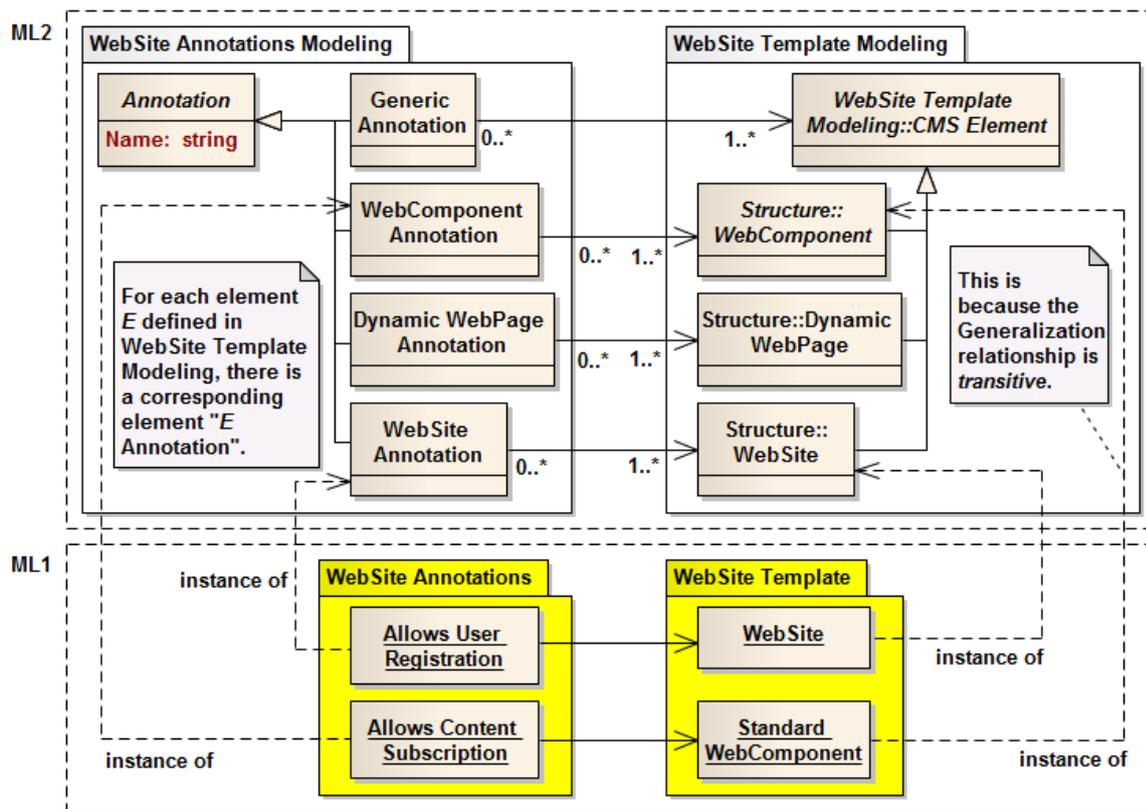


Figure 7.28: Abstract syntax for the WebSite Annotations model.

The **Annotation** concept is specialized by the other concepts in this metamodel. The most basic is the **Generic Annotation**, which can be associated with any **CMS Element** (from which all **WebSite Template Modeling** elements inherit, as described in Section 7.7). Furthermore, for each **WebSite Template Modeling** element *E* defined (e.g., **Role**, **WebComponent**, **DynamicWebPage**), CMS-ML also defines an annotation element called *E* **Annotation**, which can only be associated with instances of *E*. In other words, a **Role Annotation** can only be applied to a **Template Role**, but not to a **WebComponent**; the only annotation that can be applied to elements of different kinds is **Generic Annotation**.

Figure 7.29 provides some examples of **Annotation** elements applied to **WebSite Template** model elements, namely:

- Figure 7.29a illustrates a **WebComponent Annotation**, designated **Allows Content Subscription**, that is applied to a **CMS Standard WebComponent** of type **Forum** and indicates that the **WebComponent** should allow users to subscribe to updates to its contents (via a mechanism such as **RSS**, depending on the **CMS**). Note that **WebComponent Annotation** can be applied to a **Standard WebComponent**, because the latter is a specialization of **WebComponent** and the generalization/specialization relationship is transitive; and

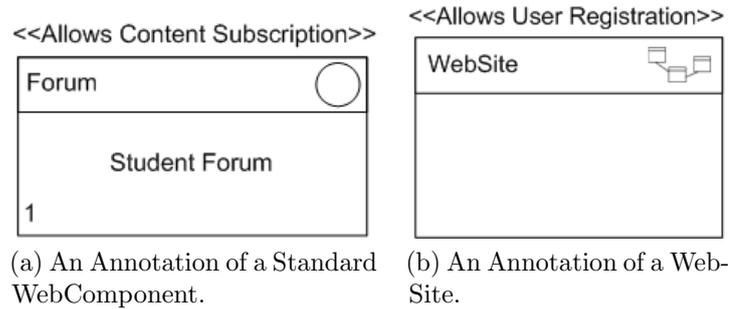


Figure 7.29: Concrete syntax for Annotations.

- Figure 7.29b depicts a **WebSite Annotation**, **Allows User Registration**, applied to a **WebSite** and indicating that the CMS (in which the CMS-ML model is deployed) should allow users to register with it (in order to become regular users, instead of anonymous ones).

It should be mentioned that a **WebSite Annotation** model *decorates* a **WebSite Template** model, allowing Designers to specify CMS-specific properties without polluting the Template model with platform-specific details. Nevertheless, from a practical perspective, we expect **WebSite Template** modeling to be performed in an integrated manner: a modeling tool would not explicitly present the Template Designer with two different working models (the Template and the Annotations), but rather a single model that results from combining those two models (i.e., from decorating the Template with the Annotations). In fact, this is the perspective that is also conveyed by the examples of Figure 7.29.

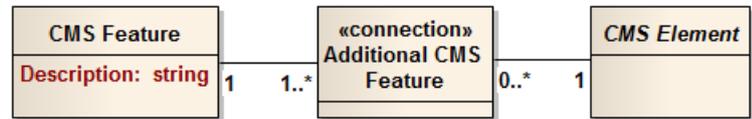
The reader may notice that an **Annotation**'s concrete syntax is similar to the representation of a UML **Stereotype** [OMG 11.e]. In fact, this representation was chosen because, from the perspective of a designer with a UML modeling tool, this Annotation mechanism is very similar to the UML stereotypes mechanism. Nevertheless, from a conceptual point of view, these two mechanisms are actually very different: UML **Stereotypes** are meant to *extend* the UML metaclasses (e.g., **Class**, **Association**) – and so their instances are defined in the M2 metalevel [OMG 11.e], where the UML metaclasses are located – while CMS-ML **Annotations** are meant to *decorate* the **WebSite Template** metaclasses (e.g., **WebComponent**, **Role**), and thus their instances are defined in the same metalevel as the instances of the **WebSite Template** concepts (see Figure 7.3).

7.7 Additional Features

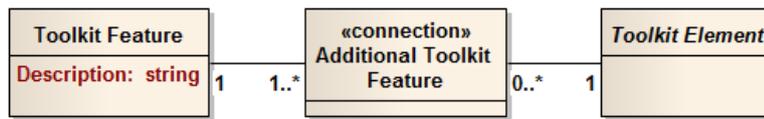
Although we consider that CMS-ML elements enable the modeling of most kinds of CMS-based web applications, this language is deliberately high-level and lacks the expressiveness

to address particular issues, such as the implementation of algorithms (e.g., how to choose an advertisement to display in a `WebComponent`). This shortcoming is expected, as CMS-ML’s objective is not to model such details, but nevertheless such desired features should not simply be ignored.

To address this issue, CMS-ML provides the **CMS Feature** and **Toolkit Feature** concepts, which consist of small textual representations of desired features or requirements (e.g., `The advertisement should be changed every 60 seconds`) that the Template Designer or Toolkit Designer (respectively) cannot model directly with CMS-ML. These can be associated with any `WebSite Template Modeling` and `Toolkit Modeling` element, with the exception of other `CMS Feature` or `Toolkit Feature` elements. Figure 7.30 illustrates the abstract syntax for these concepts.



(a) The Additional CMS Feature concept.



(b) The Additional Toolkit Feature concept.

Figure 7.30: Abstract syntax for CMS-ML Additional Features.

The **Additional CMS Feature** concept is used to establish a relationship between a `CMS Feature` and a `CMS Element`, of which *all* `WebSite Template modeling elements` are specializations. Thus, it is possible for any `WebSite Template element` to be associated with one or more additional `CMS Features`. The `Toolkit Feature` and `Additional Toolkit Feature` concepts are very similar to `CMS Feature` and `Additional CMS Feature` (respectively), as they have nearly the same meaning. Nevertheless, they are located on different metalevels (as previously explained in Section 7.3). Furthermore, an **Additional Toolkit Feature** establishes a relationship between a `Toolkit Feature` and a `Toolkit Element`, from which *all* `Toolkit modeling elements` inherit.

Figure 7.31 provides an example of the concrete syntax for each of these concepts: Figure 7.31a specifies a feature for a `Custom WebComponent` (and defines some particular details of how the `WebComponent` should behave), while Figure 7.31b defines a particular feature that should be considered when creating an instance of that `Toolkit Role`.

The main motivation for providing these **Additional Feature** concepts is to allow `Template Designers` and `Toolkit Designers` to follow a model specification workflow in which: (1) they design the model and make it as detailed and correct as allowed by

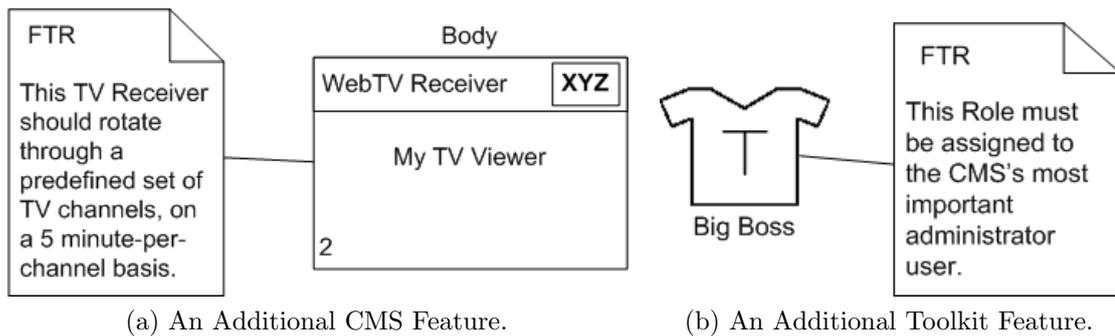


Figure 7.31: Concrete syntax for CMS-ML Additional Features.

the language; and (2) when CMS-ML is not expressive enough to specify some aspect(s) of the intended web application, they add new **Additional Features** that should be addressed, in order to comply with the web application’s requirements (which should be established *before* modeling the target system). These **Additional Feature** concepts could also be compared to Scrum’s Product Backlog items [SB 01, SS 11_d], which are the set of all features, requirements, and fixes that must be made to future versions of the product [SS 11_d].

Furthermore, although it might seem that introducing such feature-request concepts into the model is useless (because they cannot be addressed), the idea for these elements is to serve as “reminders” in other (semantically related) models. Such models will likely be specified using other modeling languages, and thus may be able to address those features; an example of such a language is CMS-IL (which will be presented in Chapter 8).

Finally, it should be mentioned that the term “Feature”, present in these concepts’ names, is derived from IEEE’s definition of *software feature* [IEEE 90]: “a distinguishing characteristic of a software item (for example, performance, portability, or functionality)”.

7.8 Importing Toolkits

Toolkits can be used in WebSite Templates, or even in other Toolkits, by means of the **Toolkit Import** modeling element, a relationship between a Toolkit (the imported element) and either a WebSite Template or a Toolkit (the importer). This relationship is transitive, which means that importing a Toolkit T_1 will automatically import all Toolkits that have been imported by T_1 . Also, it is possible to import more than one Toolkit into a WebSite Template or Toolkit, enabling the composition of Toolkit functionalities in a simple manner.

The **Toolkit Import** concept actually consists of *two concepts*: one of those concepts is located at the ML2 WebSite Template metalevel (connecting a **WebSite** to a **Toolkit**,

which is identified by *name*), and the other is located at the ML3 Toolkit metalevel (connecting a `Toolkit` to another `Toolkit`, also by name). Figure 7.32 illustrates how these concepts are related to `WebSite Templates` and `Toolkits`.

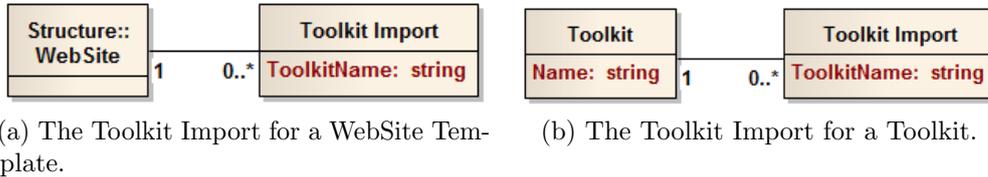


Figure 7.32: Abstract syntax for the CMS-ML Toolkit Import mechanism.

The reason why `Toolkits` are imported by means of their `Name`, instead of using a relationship to the `Toolkit` itself, is because:

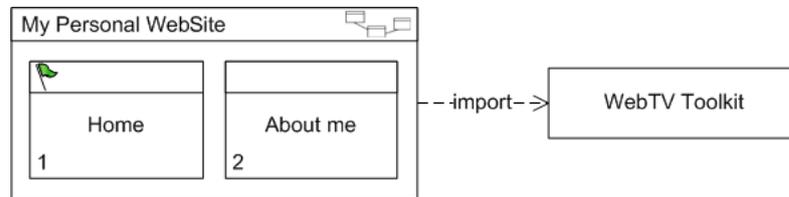
- A `Template–Toolkit` relationship would be conceptually incorrect (from the perspective of strict metamodeling), as these two concepts are located in two different metalevels, ML1 and ML2 (see Section 7.3), and the only relationship that can cross metalevel boundaries is the instance-of relationship; and
- A `Toolkit–Toolkit` relationship would require that all imported `Toolkits` also be modeled in the CMS-ML model, thus excluding possible scenarios in which the imported `Toolkit` was developed in another CMS-specific manner and is already deployed in the target CMS.

Thus, this import mechanism establishes a convenient *indirection* between `Templates` and `Toolkits`, an indirection which (1) solves the conceptual problem that would occur from a possible relationship between `Templates` and `Toolkit`, and (2) potentially reduces the amount of effort necessary to model a `Toolkit`, as it does not require that all imported `Toolkits` also be defined using CMS-ML.

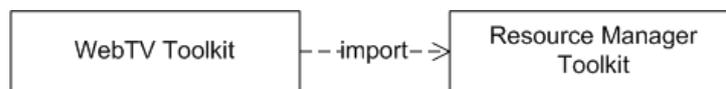
When importing a `Toolkit` into a `Template`, the elements defined in the `Toolkit`'s *Roles and WebComponents views* become available as new `Template` modeling elements; in other words, the `Template Designer` is *allowed* to use those `Toolkit`-defined `Roles` and `WebComponents` as `WebSite Template` modeling elements (otherwise, the CMS-ML model would be invalid).

On the other hand, when importing a `Toolkit` into another `Toolkit`, the elements in the imported `Toolkit`'s *Roles, Tasks, Domain, and States views* (but *not the WebComponents views*) can be used or specialized by the importer `Toolkit`. The reason why the `WebComponents view`'s elements (of the imported `Toolkit`) cannot be used in the importer is that CMS systems typically do not allow their extensions to change – or even reference – the HTML elements (e.g., buttons, input text boxes) of their components; at most, they may allow extensions to change the rendered HTML before it is sent to the user's web browser.

Figure 7.33 illustrates the concrete syntax for both `Toolkit Import` elements. It should be noted that the concrete syntax for these elements consists of a box (containing the imported `Toolkit`'s name) *and* a dashed line connecting the `WebSite/Toolkit` to that box, which explains why there is no “connection” element defined in the abstract syntax.



(a) A `WebSite Template` importing a `Toolkit`.



(b) A `Toolkit` importing another `Toolkit`.

Figure 7.33: Concrete syntax for `Toolkit Import` elements.

Regarding the representation of imported `Toolkit` elements, Figure 7.34 depicts some examples of their representation:

- Figure 7.34a illustrates a `Toolkit WebComponent` instance, `My Favorite TV`, that is used of a `WebSite Template` (more specifically, in a `Dynamic WebPage Home`). This element is an instance of a `TV Receiver WebComponent`, which was made available when the `WebTV Toolkit` was imported;
- Figure 7.34b depicts the representation of a `Toolkit Role`, called `MySite Document Manager`, used in a `WebSite Template`. This element is an instance of the `Role Document Manager`, available in an imported `Toolkit` called `DMS` (which stands for “Document Management System”);
- Figure 7.34c illustrates a simple `Domain view` for a `Toolkit` that imports other `Toolkits` (`Resources` and `Entities`). In the figure, the `Document Entity` – which is defined in the importer `Toolkit` – is a specialization of an `Entity Resource` (imported from the `Resources Toolkit`), and it is also associated with the `Person Entity` (imported from the `Entities Toolkit`).

7.9 Language Design Considerations

In addition to the CMS-ML description presented in the previous sections, it is important to provide further insight into some language design considerations that, in turn, are the motivating factor for some of the design decisions that led to the existence – or lack thereof – of some modeling elements. These considerations concern not only the metalevel

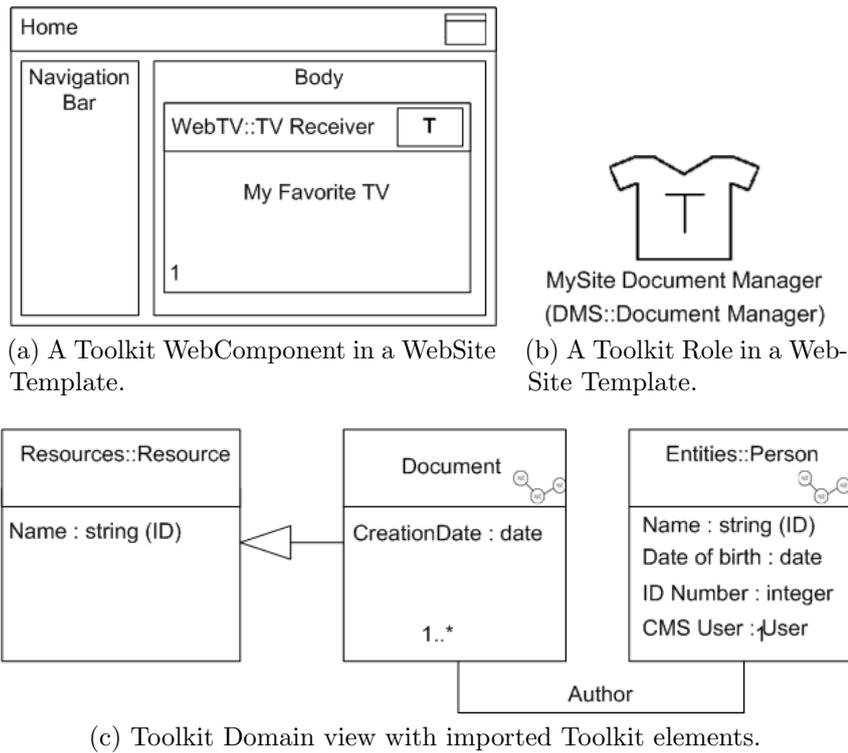


Figure 7.34: Concrete syntax for imported Toolkit elements.

structure presented in Section 7.3, but also the question of which modeling elements are relevant to the language and which are not.

The first consideration is that the tactic used for CMS-ML’s architecture – splitting the language into different metalevels, each of which with a set of hardcoded modeling elements – is actually an application of the *Language metaphor* (described in Chapter 2), as each of the unchangeable models can be considered as being hardcoded into an orthogonal metamodel. This usage of the Language metaphor – instead of the (more extensible) Library metaphor – was due to our decision to follow the simplest approach when defining the first iterations of CMS-ML. Nevertheless, CMS-ML presents no issues that would make it inadequate for the Library metaphor, although this would involve the extra work of defining more basic elements (for the hardcoded metamodel) and defining libraries for the various models considered.

Another consideration concerns CMS-ML’s extensibility, as we have opted for a limited application of *orthophrase* [Sta 75], because it is clearly more useful for extending a modeling language, as it allows the designer to add truly new elements to the language. The other approaches [Sta 75] were also considered, but discarded: (1) the paraphrase approach would serve little purpose other than supporting design patterns, such as those defined in [GHJV 95], by providing new elements that are just syntactic sugar for elements that already exist in the language; and (2) the metaphrase approach would likely be more

harmful than helpful, as the same WebSite Template model could have different meanings depending on its imported Toolkits, thus introducing a potential source of confusion for a stakeholder when looking at a model and interpreting it.

Furthermore, although CMS-ML provides a WebSite Annotations mechanism, it does not provide a corresponding Toolkit Annotations mechanism (in a manner similar to the WebSite Annotations Modeling model defined in ML2). This is because the WebSite Annotations mechanism is meant to allow the Template Designer to configure a Template's deployment with additional CMS-specific details that could not be conveyed via its elements alone (e.g., the aforementioned example specifying that a `WebComponent` should allow users to subscribe to content updates). However, considering that a Toolkit model does not define any elements that are immediately displayed to the web application's users (i.e., a Toolkit model does not configure the WebSite Template model, but rather only endows it with additional building blocks), it would not make sense to define a Toolkit Annotations mechanism to configure a Toolkit's deployment in a CMS-specific manner.

An additional consideration regards the Structure view's usage of a page-centric approach, as opposed to a content-centric one. The rationale behind the usage of a page-centric approach is that most users are accustomed to it, namely when using a web browser to navigate the World Wide Web. This activity consists of a user being presented with *pages*, which in turn contain *content* and *links to other pages*. On the other hand, content-centric approaches are typically used in a *CMS's back office*, where the definition of the web application's content before structure is often a best practice to avoid the copy-paste of contents when restructuring the web application at a later point in time. However, since CMS-ML does not consider the definition of the web application's content, a page-centric approach is the one that makes most sense.

It should also be noted that some WebSite Template modeling elements are based on similar elements from the CMS reference model presented in [Car 06]. Nevertheless, our work regarding CMS-ML goes beyond that reference model, namely by (1) addressing the behavior of CMS systems, (2) enabling the possibility of adding new elements to the language (by using the Toolkit mechanism), and (3) refining existing modeling elements (e.g., specifying their attributes) and adding new elements (such as `Content`).

The final consideration that we highlight regards the usage of side effects. It should be noted that, from a software development perspective, the use of side effects has its advantages and disadvantages [Hug 90, AS 96], of which we respectively highlight: side effects provide a way to specify a sequence of steps that can modify the program's state in some manner (e.g., change a bank account's balance); and they can make a program harder to understand and debug (because these activities require knowledge about the program's context and possible history). Nevertheless, the reason why CMS-ML includes

a Toolkit view dedicated to the definition of side effects is that Toolkit Designers can use this view to specify changes (as **Side Effects**) that will occur over the Domain view's **Entity** and **Association** instances. Those changes, in turn, can result from a variety of factors (such as *business rules*), and will often be implied by the Tasks view's **Action** that ended up resulting in performing the **Side Effect**.

Summary

In this chapter, we have presented CMS-ML, a high-level and platform-independent modeling language that aims to endow non-technical stakeholders with the necessary elements to model a CMS-based web application according to their intent. Unlike other modeling languages, CMS-ML provides a mechanism for its extension, called Toolkit, which allows designers to add new modeling elements (**Roles** and **WebComponents**) to the language in a controlled manner.

However, CMS-ML presents a lack of expressiveness (considering its target domain, CMS-based web applications) that is the result of a trade-off between language learnability and the number of modeling elements provided by the language. This, in turn, makes CMS-ML unable to address particular features (e.g., algorithm specifications) that are expected of some CMS-based web applications.

In the next chapter, we present the CMS-IL modeling language. Unlike CMS-ML, CMS-IL provides a low level of abstraction over computation concepts (in the sense that it is similar to a programming language), although it is still platform-independent. The objective of this language is to provide an implementation-independent language that can be used to (1) address low-level computation aspects that could not be handled by CMS-ML, and (2) deploy a web application model in any CMS platform (assuming, of course, that the platform can interpret CMS-IL models).

Chapter 8

CMS-IL: CMS Intermediate Language

The complexity of software is an essential property, not an accidental one. Hence, descriptions of a software entity that abstract away its complexity often abstract away its essence.

No Silver Bullet: Essence and Accidents of Software Engineering
FREDERICK P. BROOKS, JR.

The CMS-ML language, described in Chapter 7, allows non-technical stakeholders to model a CMS-based web application according to their intent, in a high-level and platform-independent manner. Unlike other modeling languages that have been analyzed in this dissertation, CMS-ML also provides the Toolkit mechanism, which enables its extension in a controlled manner.

However, the language is the result of a trade-off between learnability and the number of modeling elements provided. The main premise for this decision is CMS-ML's main objective to allow non-technical stakeholders (i.e., people without expertise in the development and maintenance of web applications and underlying technology) to quickly make correct models of CMS-based web applications. Nevertheless, this trade-off makes CMS-ML unable to address particular requirements that are typically expected of web applications, such as algorithm specifications (e.g., show an advertisement banner that chooses ads based on how much each advertiser pays) or integration with external web functionality, such as web services.

To address this problem, we propose **CMS-IL** (CMS Intermediate Language), a textual language which provides a low level of abstraction over computational concepts (in the sense that it is similar to a programming language), although it is still CMS-oriented and platform-independent. This language's objectives are:

- To provide a mechanism, independent of any particular CMS implementation, that can be used by technical stakeholders to (1) address low-level computation aspects that could not be handled by CMS-ML, and (2) deploy a web application model in any CMS platform (assuming, of course, that the platform can interpret CMS-IL models); and
- To establish a common ground for the specification of CMS-based web applications (this is, in fact, the ultimate objective of CMS-IL).

CMS-IL provides structure models that are very similar to CMS-ML. In fact, most of the structural views are identical, although some modeling elements define additional attributes that technical stakeholders typically find useful when defining CMS-based web applications. Furthermore, like CMS-ML, the language also allows for its extension, in order to address a stakeholder's specific requirements and to support the modeling of more complex web applications. Nevertheless, the greatest differences between these two languages lie in *behavior specification*, which in CMS-IL is of a much more low-level nature.

The CMS-IL language also provides a significant number of modeling elements with which to specify web application models. This is due to the aforementioned compromise between language learnability and number of elements: while CMS-ML attempts to improve learnability at the expense of having a moderate number of modeling elements (and a relatively low degree of expressiveness), CMS-IL strives to improve the degree of expressiveness, although this may make it harder to learn even by technical stakeholders.

In this chapter, we provide a general description of CMS-IL. More specifically, this chapter – which is structured in a manner similar to Chapter 7 – describes: (1) the artifacts and modeling roles considered for CMS-IL modeling; (2) the metalevels that are the basis for CMS-IL modeling, as well as the underlying rationale; (3) the modeling elements available for specifying a web application; and finally (4) the modeling elements available for extending this language with elements that are better adjusted to the web application's specific purpose. The “CMS-IL User's Guide” [SS 11.b] describes this language with a greater level of detail than the presentation that is provided in this chapter.

Before proceeding with the presentation of CMS-IL, there are some important points that should be taken into consideration by the reader.

The first point is that CMS-IL's abstract syntax is described in the present chapter using UML. Although CMS-IL has a textual concrete syntax, its abstract syntax does not provide any limitations (besides the ones regarding its multiple metalevels, as was

the case with CMS-ML) that prevent its representation using simple UML elements, such as **Class**, **Association**, and **Generalization**. Furthermore, it is possible to depict the concepts of textual languages using UML class diagrams, as long as those languages can be described in a formal manner (e.g., using EBNF [EA 06]). Thus, just like in Chapter 7, we have opted to use UML in this chapter (although not strictly following the translation rules described in [EA 06], for simplicity), in order to facilitate the reading of this chapter.

Another point is that some of the concrete syntax examples provided in this chapter contain line breaks, and some lines are indented. These breaks and line indentations are included *only* to facilitate reading, and are considered by CMS-IL as whitespace, which is ignored. The indentations, in particular, are used to indicate that the current line continues the element declaration that was present in the previous line(s).

Finally, most of the concrete syntax examples are usually presented in their simplest form (i.e., without representing other contained elements), also in order to facilitate reading. An example of such a simplified form is the following (extracted from Listing 8.8):

```
1 | CSS Class "OceanBlue Component" is applied to WebComponent "My Blog"
```

There could be a number of **CSS Classes** and **WebComponents** with such names, according to the Visual Themes and Structure views that are presented later in this chapter. Thus, if a CMS-IL model is being defined, then the names of their containing entities should also be specified – to remove any possible ambiguity – and the previous declaration should be as follows (again, the line break is only used to improve readability):

```
1 | CSS Class "OceanColors"."OceanBlue Component"  
2 | is applied to WebComponent "My WebPage"."My Blog"
```

The only exception to this rule would be if there was no possible ambiguity (e.g., if there was only one **WebComponent** named **My Blog**).

8.1 Guidelines

Like in the previous chapter, it is important to provide some answers for the guidelines identified in Section 6.3 (see Chapter 6), before starting the definition of CMS-IL. These answers, used to steer the CMS-IL definition process, are provided in the next paragraphs. Like in the CMS-ML guidelines presentation of the previous chapter, these answers are presented in an abbreviated manner, for the same reasons.

Identify the target audience for the language. The target audience for CMS-IL are technical stakeholders (namely *developers*) that are well-versed in the area of CMS-based web applications. These stakeholders are aware of the CMS website-oriented and HTML

concepts (which were presented in Chapter 7) that are employed by CMS-ML business users. However, while those business users define a new CMS Component by specifying *what* they want the Component to do, CMS-IL stakeholders define that Component by specifying the low-level details of *how* the Component will accomplish its objective. Furthermore, because of their web application development background, these stakeholders are also aware of typical programming language constructs, such as the declaration of variables and the assignment of elements to those variables.

Identify the problem-domain that the language should address. Like the case of CMS-ML, the problem-domain for this language is the universe of CMS systems and CMS-based web applications. However, because of its target audience, CMS-IL is oriented toward the specification of *how* to address the statements that have been modeled in CMS-ML; in turn, this will require that the former provide technical concepts (namely to represent *instructions* to be run by a computer) that are not present in the latter. Other aspects to consider include the specification of the web application's users, contents, and look-and-feel, as well as the localization and structuring of the aforementioned contents. Once again, we will not expand on the identification of this problem-domain right now, for the same reasons as in the previous chapter.

Determine the degree of extensibility that the language should address. As in CMS-ML, and according to the identified problem-domain, CMS-IL stakeholders need the ability to add new kinds of CMS Component, which will support the tasks (if any) that were previously identified in CMS-ML. Due to those tasks, stakeholders will also need to specify new kinds of role that will participate in the aforementioned Components.

CMS-IL stakeholders will occasionally need to add snippets of source code (written in a CMS-specific programming language), namely calls and auxiliary functions that will interact directly with the underlying system.

Furthermore, stakeholders should also be able to define source code (in either CMS-IL or some CMS-specific language) that can extend the CMS's behavior, namely with the ability to: (1) intercept certain events that occur when a web request takes place (e.g., the request's user is authenticated with the CMS); (2) implement functionality for an **Action** defined in the corresponding CMS-ML model; and (3) implement functionality corresponding to a CMS-ML **CMS Feature** or **Toolkit Feature**. The functionalities available to this code should include the ability to interact with the elements that have been defined in the CMS-IL model.

Finally, stakeholders will not need to modify the Components that are available out-of-the-box in the CMS, for the same reasons that have been presented in CMS-ML.

Considering the identified problem-domain, determine the language’s modeling levels and their hierarchy. Considering the previous answers, the important *composition relationships* identified are: (1) between a Component and its HTML parts (when defining new Components); (2) between a Website, its Pages, and their Components; and (3) between a snippet of source code and its containing entity (typically a Website, Page, or Component). Some of these relationships are also present in CMS-ML, which can be considered as a motivation to reuse some of its concepts in CMS-IL (albeit with some changes when deemed necessary).

We have identified the same relevant *instance-of relationships* as in CMS-ML, namely: (1) a Website will contain instances of user-defined Components; and (2) some CMS roles will be instances of roles that were designed to interact with those Components.

Taking these relationships into consideration, the CMS-IL language requires the existence of at least two metalevels (in addition to the metalevel that will contain the CMS instances): one metalevel for stakeholder-defined Components and CMS extensions, and the other for specifying the Website itself (with instances of those Components).

Identify any constraints that may condition the choice of a metamodeling language. As was the case with CMS-ML, the only noteworthy constraint detected was the need to allow stakeholder modeling in two metalevels (in addition to the “reality” metalevel). Nevertheless, none of the identified metalevels revealed any particular constraint that would make us consider most current metamodeling languages as unsuitable for describing it.

Identify the kind of concrete syntax that the target audience is most comfortable with. Unlike CMS-ML, which features a target audience of business users that typically favor visual modeling languages of a simpler nature, CMS-IL’s target audience – technical stakeholders – tend to prefer text-based languages, because their activities often include the usage of textual programming and scripting languages [KP 09].

These guidelines are the motivating factor behind most of the language design decisions taken during the definition of CMS-IL. The remainder of this chapter is dedicated to presenting the CMS-IL language, and explaining some of the design decisions involved in this process.

8.2 Model Types and Modeling Roles

Overall, the CMS-IL modeling process is very similar to CMS-ML’s, presenting only some differences in specific stages of the process. CMS-IL modeling is mainly focused on three

different model types: (1) WebSite Templates, (2) WebSite Annotations, and (3) Toolkits. These models have the same names as in CMS-ML because they play the same overall roles, the main difference between them lying in the scope and detail of those models. Figure 8.1 illustrates the relationships that take place between these CMS-IL model types.

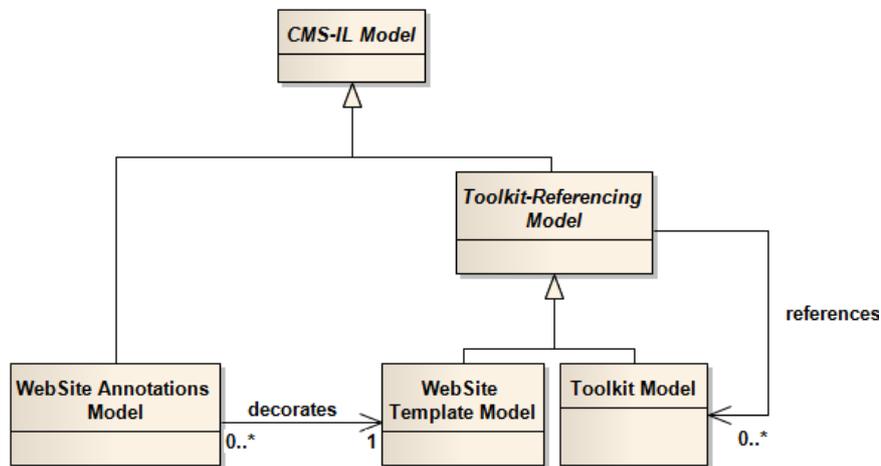


Figure 8.1: Relationship between the different CMS-IL models.

A **WebSite Template** (or just *Template*) is a model that reflects the intended web application’s *structure*; it is modeled using CMS elements that are provided by CMS-IL, such as **WebSite** or **Dynamic WebPage**. Overall, it can be considered as a refinement of the WebSite Template in CMS-ML, although the only formal relationship between these elements lies in the model synchronization mechanism that is presented in Chapter 9.

Furthermore, the elements of a WebSite Template model can be annotated with elements defined in a **WebSite Annotations** model (or just *Annotations*). The elements of a WebSite Annotations model consist simply of tags (strings) that can be “attached” to a Template element (i.e., the tags *decorate* Template elements). These tags, in turn, are expected to be interpreted by the CMS in which the CMS-IL model is deployed; if the CMS does not recognize a certain tag, then the Template elements to which it is attached will be interpreted according to their default semantics.

On the other hand, the **Toolkit** model allows the definition of new modeling elements that can be used in WebSite Templates. It is mainly focused on domain and user interface modeling, as well as defining behavior (in a programming-like manner) for various event-driven aspects of the Toolkit. Like in CMS-ML, a Toolkit can be referenced by a WebSite or by other Toolkits.

Of course, as in CMS-ML, it is not required that a single CMS-IL stakeholder should have the skills to create all kinds of CMS-IL models. Thus, we consider that the development of CMS-IL models will typically be performed according to the following roles (depicted in Figure 8.2):

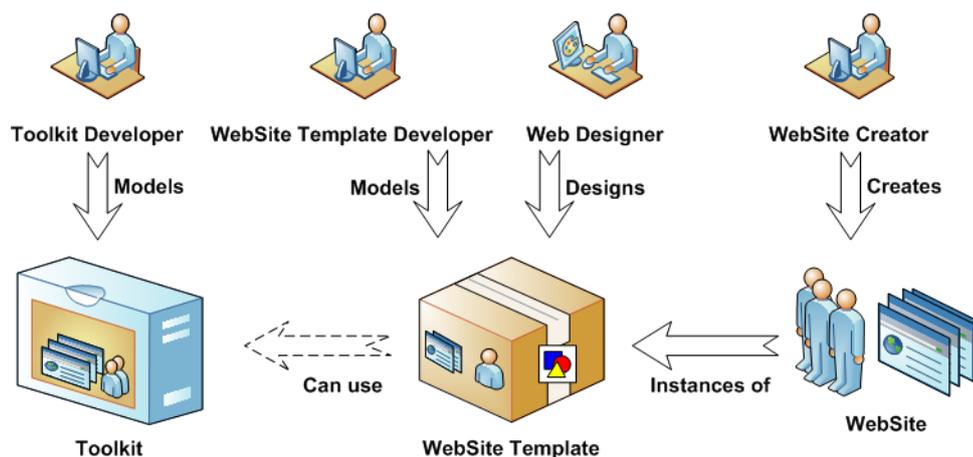


Figure 8.2: Modeling roles and artifacts considered by CMS-IL.

- The **Toolkit Developer**, who specifies Toolkits using programming language-like elements, but still in a platform-independent manner;
- The **WebSite Template Developer** (also usually designated as *Template Developer*), who models a WebSite Template and, optionally, annotates it with a WebSite Annotations model;
- The **Web Designer**, who defines the visual themes, graphics, and layouts for the Template; and
- The **WebSite Creator**, who instantiates the elements defined in the Template.

The remainder of this chapter will explain the CMS-IL language while taking these roles into account.

Figure 8.2, when compared to Figure 7.2, also shows that the main differences between CMS-ML and CMS-IL modeling are: (1) the Toolkit Designer role of CMS-ML is replaced by the Toolkit Developer role, as they do not have the same know-how requirements; (2) the WebSite Template Designer role of CMS-ML is replaced by the WebSite Template Developer role, for the same reasons; and (3) the addition of the Web Designer role, who is tasked with the configuration of the web application’s look-and-feel by defining CSS (Cascading StyleSheet) classes.

On the other hand, the WebSite Creator role in both CMS-IL and CMS-ML can be performed by the same kind of stakeholder (e.g., CMS administrators). This is because the skill set required for the role in these two languages are actually the same: technical expertise in the configuration and day-to-day maintenance of a CMS system. Although at first sight this might seem like a duplication of responsibilities (e.g., why would there be a need for a WebSite Creator to deal with CMS-ML and CMS-IL models?), it is expected that these languages are used in a workflow like the one previously depicted in Figure 6.1, in which the WebSite Creator will only have to deal with CMS-IL models.

8.3 CMS-IL Architecture

The metamodel architecture of CMS-IL follows the same tactic as CMS-ML, namely the definition of a set of models located at different metalevels. We also consider that this strategy’s advantages – previously enumerated in Section 7.3, namely (1) addressing language extension in a simple manner, (2) reducing accidental complexity [AK 08], and (3) obeying the strict metamodeling doctrine [AK 02, Küh 09] – make it adequate for the first iterations of CMS-IL. Furthermore, this kind of Language metaphor-based architecture (described in Chapter 2) is relatively easy to implement, which is particularly helpful in these first iterations.

The architecture of CMS-IL – illustrated in Figure 8.3 – is very similar to CMS-ML’s, and considers the following metalevels:

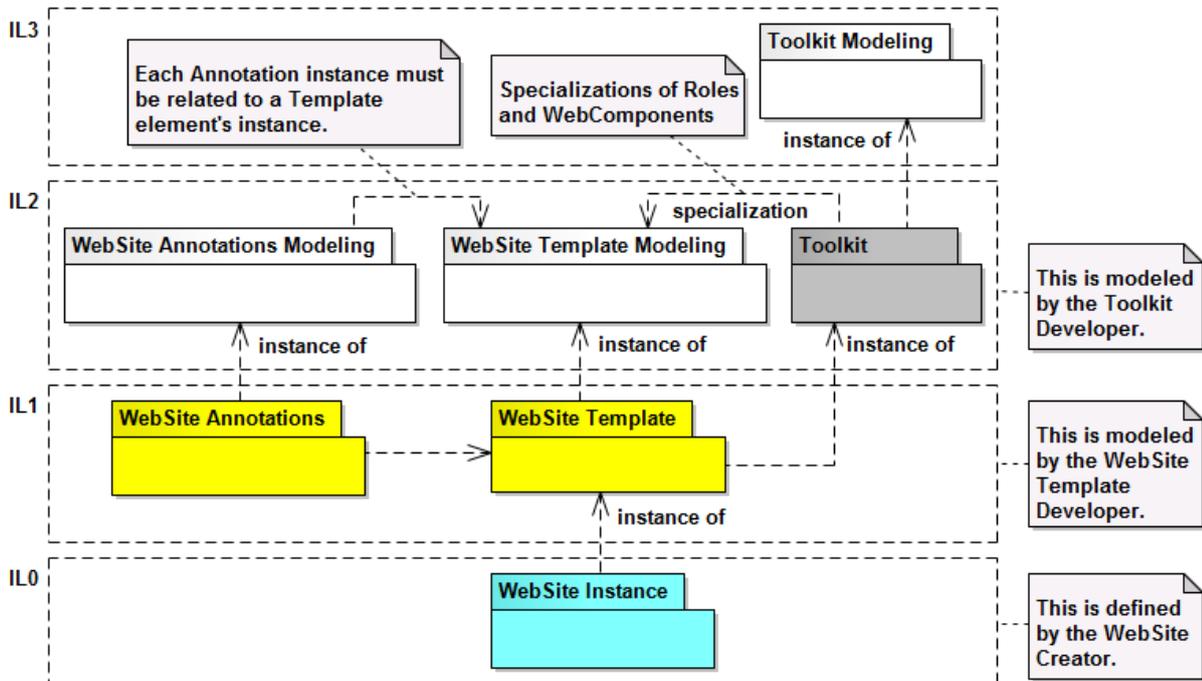


Figure 8.3: Metalevels considered by CMS-IL.

- The metalevel **IL3** defines all of CMS-IL’s Toolkit modeling concepts, in the *Toolkit Modeling* model (which, in turn, is the metamodel for the Toolkit model in IL2). Like in CMS-ML, stakeholders cannot make any changes on this metalevel;
- **IL2** is the metalevel that provides the *WebSite Template Modeling* and *WebSite Annotations Modeling* models, which are used as the metamodels for WebSite Template and WebSite Annotation models, respectively. Furthermore, it is in IL2 that Toolkit Developers can specify their Toolkit models, by creating instances of

the modeling concepts defined in IL3. Of course, the WebSite Template Modeling and WebSite Annotations Modeling models cannot be changed by anyone;

- The **IL1** metalevel is where WebSite Template Developers and Web Designers create *WebSite Template* and *WebSite Annotations* models. These models are specified by using the WebSite Template Modeling and WebSite Annotations Modeling concepts defined in IL2, as well as the Toolkit model elements defined in IL2 (which, in turn, are themselves instances of the concepts defined in IL3);
- **ILO** is the metalevel in which the WebSite Creator operates, namely by creating instances – in a particular CMS installation – of the elements that were defined in IL1’s WebSite Template and WebSite Annotations models. This will typically be performed in an automated manner, by using a CMS-specific mechanism that can map model elements to instances, although it should be considered that the WebSite Creator may have to manually adjust some CMS-specific details that are not contemplated by the mapping operation.

Concerning *extensibility*, we have opted for an approach very much like the one in CMS-ML, namely the use of *orthophrase* [Sta 75] to allow for the addition of new modeling elements to the language. Likewise, the rationale for why not to use other extensibility approaches [Sta 75] is the same as that provided in the previous chapter.

8.4 WebSite Template Modeling

The CMS-IL WebSite Template model can be considered as a superset of the CMS-ML WebSite Template described in Chapter 7. This model allows the Template Developer to configure the structure of the CMS-based web application (and, to a limited extent, its behavior regarding the permitted actions for each of the roles considered by it).

To support this model, the CMS-IL language defines a set of modeling elements (to which we again call *CMS elements*) with which Template Developers can define Templates. As was the case for CMS-ML WebSite Templates, a CMS-IL Template is defined according to a set of views (illustrated in Figure 8.4), namely:

- The *Structure view*, which defines the structure of the web application, namely its pages and their components;
- The *Roles view*, specifying the responsibilities that the web application expects of its users;
- The *Permissions view*, modeling what the web application’s roles are allowed to do;
- The *Users view*, which defines particular users that are considered important to the modeled web application (and thus should be available immediately after deploying the model to the CMS);

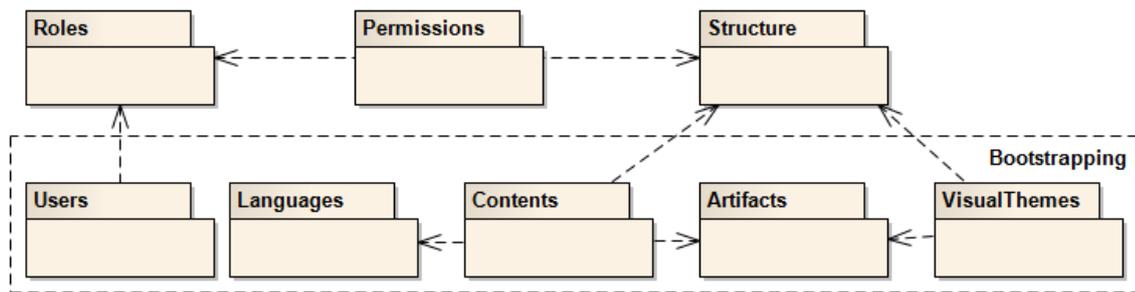


Figure 8.4: Views involved in the definition of a WebSite Template.

- The *Languages view*, which deals with localization and the languages that the web application should have available;
- The *Artifacts view*, where the Developer can define a set of language-agnostic artifacts (strings and files) that will be available in the web application;
- The *Contents view*, which specifies content to be displayed by the web application’s structural elements (namely content provided by the Artifacts view); and
- The *Visual Themes view*, which specifies the graphical layout and properties of the web application’s various structural elements.

The *bootstrapping views* are not mandatory for the modeling of a Template, and should only be defined when Template Developers have *a priori* content that should be available in any web application that is an instance of the modeled Template.

Once again, the WebSite Template deals mainly with the web application’s *structure*. On the other hand, its *behavior* is specified in a Toolkit (which is described further down this chapter), because (1) behavior is typically defined by each CMS **WebComponent** and not by the CMS itself, and (2) CMS administrators are usually able to only change a specific set of parameters regarding the system’s behavior. However, unlike CMS-ML, this Toolkit behavior will be specified using programming language-like concepts, instead of a graphical modeling language.

8.4.1 Structure View

The Structure view is one of the cornerstones of the WebSite Template. This view defines a set of concepts – **WebSite**, **Dynamic WebPage**, **Container**, and **WebComponent** – which can be considered as nearly equivalent to those in CMS-ML’s Structure view (because they have the same responsibilities). Figure 8.5 presents the abstract syntax for the Structure view.

It should be mentioned that there are some differences between the Structure views of CMS-ML and CMS-IL, namely:

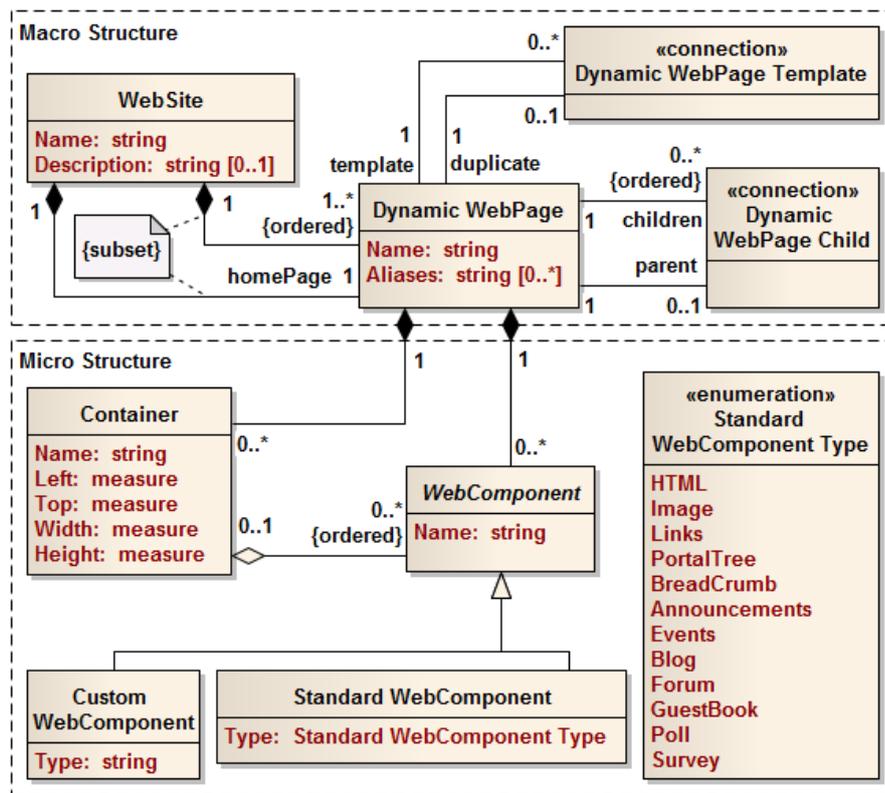


Figure 8.5: Abstract syntax for the Website Template’s Structure view.

- `WebSite` defines an additional property, `Description`, which is optional and provides a description of the web application. Although it would be possible to also include this property in CMS-ML, we have found that most stakeholders typically consider it as irrelevant (and leave it empty), making its inclusion not worthwhile. However, it should be mentioned that this attribute *is* important, namely because most web search engines use it when displaying search results;
- `Dynamic WebPage` defines another property, `Aliases`, consisting of a set of strings that provide *alternative names* for the page (e.g., an `About` page can also have an alias `Contacts`). The rationale for not merging this property with `Name` is that `Name` has the responsibility of being the page’s *canonical name* (i.e., its authoritative name); and
- The `WebSite`, `Dynamic WebPage`, `WebComponent`, and `Container` concepts all inherit from another abstract concept, `Structural Element`. This concept is not represented in Figure 8.5 for simplicity, but is referenced in other views to enable relationships with *structural* elements (i.e., elements that inherit from `Structural Element`).

Listing 8.1 provides a simple example of the Structure view’s concrete syntax. This example is the CMS-IL equivalent to the CMS-ML `WebSite` (and its contained `Dynamic`

WebPages and WebComponents) that was previously represented in Figure 7.6. It is important to note that: (1) the Macro and Micro Structure sub-views are not differentiated in the concrete syntax; (2) the order of elements is not explicitly represented in the concrete syntax, but is instead determined by the order in which each element – namely instances of Dynamic WebPages and WebComponents – is declared within its parent element; and (3) for illustrative purposes, line 8 includes a Dynamic WebPage Template relationship between the About Me and Home Pages (this relationship is not included in Figure 7.6).

Listing 8.1: Concrete syntax for the WebSite Template’s Structure view.

```

1 WebSite "My Personal WebSite" has
2   HomePage "Home" with
3     Container "Banner" at (22%, 1%, 77%, 10%)
4     Container "Navigation Bar" at (1%, 1%, 20%, 98%)
5     Container "Body" at (22%, 12%, 77%, 87%) with
6       WebComponent "My Blog" of Standard type "Blog"
7       WebComponent "My TV Viewer" of Custom type "WebTV Receiver"
8   Page "About Me" follows layout of Page "Home" (...)

```

8.4.2 Roles View

The Roles view, which is identical to the WebSite Template’s Roles view in CMS-ML, describes the responsibilities that the modeled CMS-based web application expects its users to assume. Like CMS-ML’s view, it defines two concepts, **Role** and **Role Delegation**, which can be used to model those responsibilities and specify whether they can also be played out by other Roles, respectively. Figure 8.6 illustrates the abstract syntax for the Roles view.

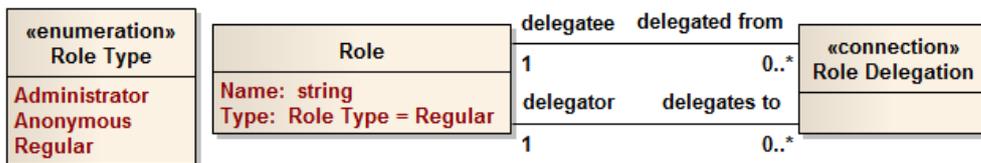


Figure 8.6: Abstract syntax for the WebSite Template’s Roles view.

As was already mentioned, this view is identical to the one in CMS-ML. The reason for this (instead of the CMS-IL Role defining more attributes, for example) is that, from the previously obtained guidelines (presented in Section 8.1), we considered there were no further details that would be useful to CMS-IL modelers other than those that were already present in CMS-ML.

Listing 8.2 provides some examples of the concrete syntax for this view. These examples are semantically equivalent to the CMS-ML Role and Role Delegation examples that

can be found in Figure 7.8; more specifically, (1) line 1 corresponds to Figure 7.8a, (2) line 3 corresponds to Figure 7.8b, (3) line 5 corresponds to Figure 7.8c, and (4) line 7 corresponds to Figure 7.8d.

Listing 8.2: Concrete syntax for the WebSite Template’s Roles view.

```

1 Role "ARole"
2
3 Role "AnAdministrationRole" is Administrator
4
5 Role "AnAnonymousRole" is Anonymous
6
7 Role "Manager" delegates to "Secretary"

```

8.4.3 Permissions View

The Permissions view, which is very similar to its CMS-ML-homonym view, is responsible for establishing a correspondence between Roles and the structural elements of the website, namely its Dynamic WebPages and WebComponents. It defines two concepts, also called Dynamic WebPage Permission and WebComponent Permission. These concepts have the same name as those of the CMS-ML Permissions view because they have same function: the creation of Role–Dynamic WebPage and Role–WebComponent relationships. Figure 8.7 provides an illustration of these concepts. Like in CMS-ML, the terms “configuration” and “management” respectively consist of (1) changing the values of its properties, and (2) changing some/all of its associations to other elements.

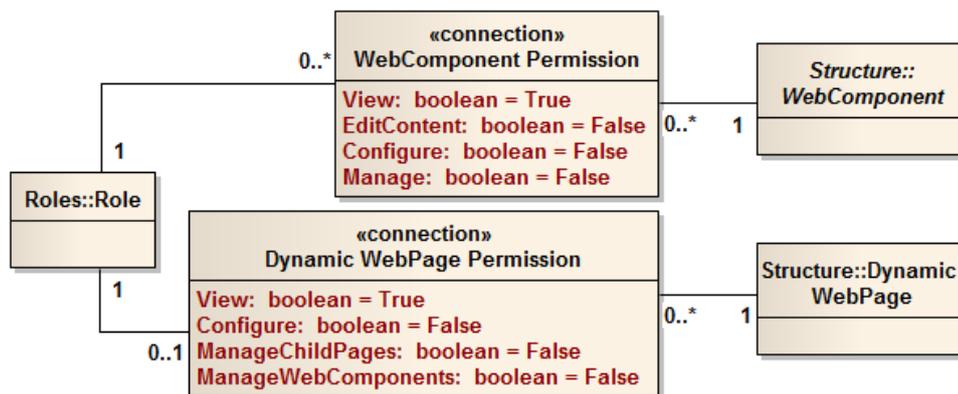


Figure 8.7: Abstract syntax for the WebSite Template’s Permissions view.

A **Dynamic WebPage Permission**, as implied by its name, specifies what a Role can do regarding a certain Dynamic WebPage. It defines the same permissions as in CMS-ML: (1) viewing and (2) configuring the page, as well as managing the page’s (3) child pages and (4) WebComponents.

On the other hand, a **WebComponent Permission** determines what a **Role** can do regarding a **WebComponent**. It defines permissions for (1) viewing, (2) editing, and (3) configuring a **WebComponent**. However, it also adds another permission, **Manage**, that allows a **Role** to manage the **WebComponent** (i.e., change its relationships to other elements, such as moving the **WebComponent** to a different **Container**, or changing its order within the parent **Dynamic WebPage**) without the requirement of being able to configure it (e.g., set the number of posts to show in a blog).

Like in CMS-ML, each of these permissions has a default value: viewing permissions assume the default value **True**, while editing, configuration, and management permissions default to **False**.

Listing 8.3 depicts the concrete syntax for this view. Although there are some differences from the CMS-ML example presented in Figure 7.10 (because of the differences in the metamodels), these two examples are considered to be semantically equivalent: (1) lines 1–2 correspond to the **Dynamic WebPage Permission** example of Figure 7.10a; (2) lines 4–5 specify the same as the matrix in Figure 7.10b; (3) lines 7–9 are equivalent to the permissions in the matrix of Figure 7.10c; and (4) lines 11–14 correspond to the matrix of Figure 7.10d. Again, and as in Figure 7.10, permissions that are not explicitly specified assume their default value.

Listing 8.3: Concrete syntax for the **WebSite Template’s Permissions** view.

```
1 Role "ARegularRole" can (view, manage WebComponents of) Page "ADynamicWebPage"
2 Role "ARegularRole" cannot (configure, manage child pages of) Page "ADynamicWebPage"
3
4 Role "ARegularRole" can (view, edit content of) WebComponent "My Blog"
5 Role "ARegularRole" cannot configure WebComponent "My Blog"
6
7 Role "ARegularRole" cannot (configure, manage child pages of) Page "Another Dynamic WebPage"
8 Role "ManagerRole" can (view, configure, manage child pages of, manage WebComponents of) Page
  "ADynamicWebPage"
9 Role "ManagerRole" can (configure, manage child pages of, manage WebComponents of) Page
  "AnotherDynamicWebPage"
10
11 Role "RegularRole" can edit content of WebComponent "Forum"
12 Role "RegularRole" cannot configure WebComponent "Forum"
13 Role "BlogManagerRole" can (view, configure, edit content of) WebComponent "MyBlog"
14 Role "BlogManagerRole" can view WebComponent "Forum"
```

8.4.4 Users View

The **Users** view is responsible for identifying any relevant CMS users that should be available whenever the CMS-IL model is deployed, as well as their **Role** assignments. A typical example could be an administrator user other than the default one that most CMS

systems define out-of-the-box. It should be noted that this view is optional, and it is useful only for bootstrapping the web application’s operation after the CMS-IL model is deployed. This view defines two concepts, **User** and **User Assignment**: the former specifies a CMS user itself (a specific *person* that will interact with the modeled web application), while the latter models the assignment between a **User** and the **Role(s)** – i.e., the expected responsibilities within the web application – to which the **User** is assigned. Figure 8.8 illustrates the abstract syntax for the Users view.

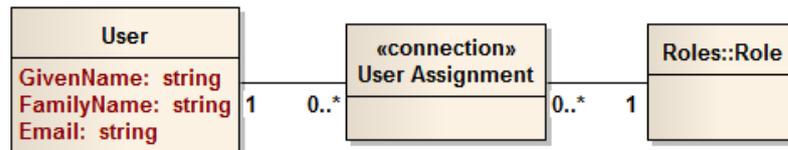


Figure 8.8: Abstract syntax for the WebSite Template’s Users view.

Listing 8.4 provides an example of the concrete syntax for these two concepts. More specifically, (1) the example in line 1 illustrates a **User** named **John Doe** (**Given Name** is **John** and **Family Name** is **Doe**), while (2) line 3 represents a **User Assignment** relationship between the **User John Doe** and one of the **Roles** (in this case, **Manager**) that the **User** is to perform.

Listing 8.4: Concrete syntax for the WebSite Template’s Users view.

```

1 User " John Doe" ("john.doe@company.com")
2
3 User " John Doe" has the Role " Manager"
  
```

The reason why **User** does not define a **Name** attribute, but rather two attributes **Given Name** and **Family Name**, is that some CMS systems (e.g., DotNetNuke, which is analyzed in Appendix B) actually differentiate between these two kinds of name, most likely so that they can provide a more familiar environment to the user (e.g., by displaying a message “Hello, John!”, which feels more personal and familiar – because it treats the user on a first name basis – than the message “Hello, John Doe.”).

Furthermore, **Given Name** and **Family Name** are named as such in order to make the **User** concept adequate for non-western cultures, in which the **Family Name** usually comes before the **Given Name**. This explicit separation allows the CMS to correctly identify the person’s family and given names (e.g., for the Chinese name “Yao Ming”, the family name is “Yao”, and not “Ming” as most western cultures would assume). To address this issue, the **User**’s concrete syntax supports the same rules as L^AT_EX’s B_IB_TE_X, namely (a) the usage of western rules – **Given Name** followed by **Family Name** – to represent the **User**’s name, unless (b) a comma (“,”) is used, in which case the **Family Name** comes before

the comma and the **Given Name** comes afterward [SS 11_b]. These rules are especially important when parsing a CMS-IL model (to obtain the corresponding abstract syntax elements), as they remove the ambiguity that underlies typical name parsing.

It should be noted that the **User Assignment** represented in line 3 does not explicitly represent the **Email** property, because in this example we assume that there is only one **User** with **Given Name** and **Family Name** as **John** and **Doe**, respectively. If this was not the case, and there was more than one such **User** (a situation that, in practice, is not uncommon), then their respective **Email** addresses would have to be represented, in order to remove any possible ambiguity between those **Users** (as an e-mail address is typically used only by a single person). Another way to solve this problem would be to assign the **User** to a CMS-IL alias (as explained further down this chapter, in Section 8.7), and then use that alias to assign the **User** to the **Role**. Interested readers may consult the “CMS-IL User’s Guide” [SS 11_b] for further details.

8.4.5 Languages View

The Languages view addresses a part of CMS-IL’s content localization issues, by enabling the specification of what languages will be supported in the web application. It defines a single concept, **Language**, which is used to model the various languages to be considered (e.g., English, Portuguese, Spanish). Figure 8.9 provides a simple illustration of the abstract syntax for this view.

Language
Name: string
ISO_Name: string
Order: int

Figure 8.9: Abstract syntax for the WebSite Template’s Languages view.

Of the attributes defined by the **Language** concept, **ISO Name** and **Order** warrant further explanation:

- The **ISO Name** attribute consists of the language identification code (according to Best Current Practice 47¹), which in turn is also typically used by web browsers when making requests to the web application (by specifying the HTTP header **Accept-Language**²); and
- **Order** consists of the **Language**’s order in the context of the web application. More specifically, this establishes an ordered set of **Languages** for the web application,

¹<http://www.rfc-editor.org/rfc/bcp/bcp47.txt> (accessed on February 18th, 2012)

²<http://www.w3.org/International/questions/qa-accept-lang-locales> (accessed on February 18th, 2012)

which in turn can be used by the supporting CMS system to choose the language in which the web application's elements (and content) will be displayed to the user.

Listing 8.5 illustrates two examples of this view's concrete syntax: (1) the example in line 1 defines the **English Language**, while (2) line 3 corresponds to the **Portuguese Language**. It should be noted that, like in the Roles view, the order of each **Language** is implicitly specified by the order in which it is declared in the model; in other words, if **English** was declared *after* **Portuguese**, then their **Order** attributes would assume the values 2 and 1, respectively.

Listing 8.5: Concrete syntax for the WebSite Template's Languages view.

```

1 Language "English" ("en-uk")
2
3 Language "Portuguese" ("pt-pt")

```

The reason why this view is so simple (i.e., it defines just a single concept) is that it is used only to *identify* the regions/cultures that are considered relevant for the web application's purpose (namely by defining their names and ISO codes). It will be up to the CMS system itself to provide the corresponding UI text translations, and to the Contents view (explained further down this chapter in Subsection 8.4.7) to provide the localized contents.

8.4.6 Artifacts View

The Artifacts view allows the Template Developer to provide some *a priori* artifacts (as the name suggests), namely strings and files. These artifacts can then be used in the Contents and Visual Themes views, described in the next subsections. Figure 8.10 illustrates the abstract syntax for the Artifacts view.

This view is modeled by specifying **Artifacts**. An **Artifact** is an object with a **Name** and representing some content (e.g., the string "Hello!"). Furthermore, each **Artifact** is categorized as either a **String** or a **File**.

A **String** is a simple concept that consists of an ordered set of characters, just like strings in typical programming languages. However, CMS-IL contemplates two different kinds of **String**, (1) **Placeholder Strings** and (2) **Absolute Strings**. Both kinds of **String** support the usage of *escape characters* (i.e., characters that change the meaning of the characters that follow). To facilitate the specification of **Strings** by Template Developers, CMS-IL adopts the escape semantics that are typically found in programming languages: the character `\` is used to indicate that the following character should be *escaped* (i.e., not interpreted as-is), and any occurrence of `\\` is interpreted as regarding the character `\` itself.

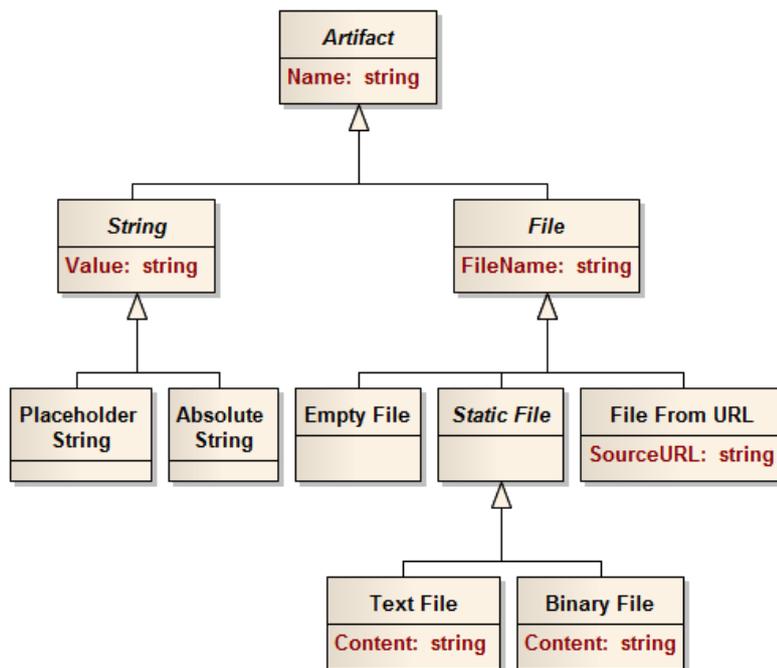


Figure 8.10: Abstract syntax for the WebSite Template’s Artifacts view.

An **Absolute String** is a **String** that can contain *only* certain escape character sequences (namely the ones to include string terminator characters: `\\`, `\"`, and `\'`), and all other characters are interpreted as-is. An example of this difference in character sequences is that `“Hello\n”` contains 7 characters (`\n` counts as 2 characters), but `“\’Hello\’!”` contains 8 characters (as `\’` counts only as a single character, `’`).

On the other hand, a **Placeholder String** not only supports the escape character sequences of **Absolute String**, but also increases the range of supported sequences (e.g., with `\t`, `\n`, `\b`, and other sequences typically supported by programming languages). Furthermore, it enables the usage of **named placeholders**, which are special markers, with a name (surrounded by `#{` and `}`), that are used to pinpoint particular positions in a **String**. These placeholders can then be used to dynamically include text in the corresponding string, whenever the **String** is referenced in other CMS-IL views. A typical example of this feature would be the replacement of the placeholder `user`, in the string `“Welcome, #{user}!”`, with the given name of the CMS user that is currently authenticated (if any), in order to obtain a string like `“Welcome, Jack!”`.

As for the remaining kinds of **Artifact**, a **File** represents a file in the storage medium in which the CMS system operates; even if the storage medium does not support the concept of files (e.g., a cloud-based environment such as Microsoft’s Azure which uses binary large objects, or *blobs*), it is customary for such environments to provide some similar data-storage mechanism. CMS-IL contemplates the following kinds of **File**: (1) the

Empty File, as the name indicates, consists of a file without any content (which can be useful to flag certain conditions, such as *Application successfully installed and configured*); (2) the **File From URL** consists of a file whose initial contents (i.e., its contents when the CMS-IL model is deployed to a CMS) should be obtained from the specified **SourceURL**; (3) the **Text File** is a file with text content (the content is specified in the same manner as a **Placeholder String**); and (4) the **Binary File** is a file, similar to **Text File**, but it has binary contents, which are specified in Base64 encoding (in order to be readable and writable as text in a CMS-IL model, without requiring encoding schemes such as UTF-8).

It should be noted that, although there is a **File** concept, CMS-IL does not provide the concept of *directory*. This is because the manner in which files (or other content) are *persisted* should typically be considered as an internal detail of the CMS system (in order to promote abstraction from implementation details). Furthermore, most CMS systems provide some mechanism that supports the storage of key-value pairs (e.g., a NoSQL implementation such as Apache CouchDB³, to which the **File** concept can be mapped by storing the pair <file name, file content>. However, it is not as common for such mechanisms to support namespaces (or similar), which would be necessary to support directories; although it would be possible to specify the key as a concatenation of the directory name and the file name, that solution could then lead to other problems, such as complicating index or search operations. Thus, by not providing a *directory* concept, it becomes easier to implement a CMS-IL model interpreter mechanism if the target CMS does not provide a traditional file system-based storage medium.

Listing 8.6 provides some examples of the concrete syntax for the elements provided by this view:

Listing 8.6: Concrete syntax for the WebSite Template's Artifacts view.

```

1 String "WelcomeMessage" is "Welcome, #{user}!\n\nPlease select an action:"
2
3 String "PasswordRequirements" is absolute "Your password must include the characters # and \\"
4 String "PasswordForbiddenChars" is absolute "Your password must NOT include the characters \" or \"
5
6 File "website_has_been_configured.txt"
7
8 File "license.txt" from "http://www.mycmsdomain.com/eula.txt"
9
10 File "text_file.txt" with text content "Hello"
11
12 File "binary_file.txt" with binary content "0a5b="

```

- Line 1 illustrates a **Placeholder String** named `WelcomeMessage`, which contains a placeholder, `#{user}`, and two `\n` escape characters;

³<http://couchdb.apache.org> (accessed on February 18th, 2012)

- Line 3 illustrates an **Absolute String**, which contains a **#** (a placeholder character) that is interpreted as-is, and an escape sequence **** that is interpreted as ****;
- Line 4 illustrates another **Absolute String** that contains two escape sequences, **\"** and **\'**, which are respectively interpreted as **"** and **'**;
- Line 6 depicts an **Empty File**, whose **Name** indicates that the website represented in the CMS-IL model has been configured;
- Line 8 represents a **File From URL** that fetches its initial contents from the specified URL (in this case, a file containing the text for an end-user license agreement);
- Line 10 presents a **Text File** containing a simple string “Hello”; and
- Line 12 presents a **Binary File** that contains a Base64-encoded set of octets.

Lines 1–4 of Listing 8.6 also illustrate that the main reason for two different kinds of **String**, **Placeholder String** and **Absolute String**, is to respectively facilitate the creation of (1) *language-specific strings* and (2) strings with a number of typical escape sequences that are *not* meant to be interpreted as such (and thus to make these **Strings easier to read**). These two kinds of **String** are inspired by the Python [Lut 09] and Ruby [FM 08] programming languages’ strings: (1) **Placeholder String** is similar to Python’s normal strings and Ruby’s double-quote strings, while (2) **Absolute String** is similar to Python’s raw strings and Ruby’s single-quote strings.

It should also be mentioned that the **Placeholder String** feature – called *string interpolation* in programming languages – is again inspired by the Python and Ruby languages, which also provide it (albeit with a slightly different syntax, in Python’s case). However, unlike those languages, CMS-IL does not provide a string interpolation mechanism based on the *order* of elements in a string, as such mechanisms are usually error-prone (because it is easy for a developer to make a mistake and interpolate the string with a set of elements in the wrong order).

8.4.7 Contents View

The Contents view can be used to provide the initial contents for the website, namely through the usage of artifacts (defined in the Artifacts view). The abstract syntax for this view is illustrated in Figure 8.11.

Modeling in this view starts with defining instances of the **Content** concept, which represents some string that will be shown to a user browsing the website. A typical example is the text in a website’s banner (e.g., **My Website**) or the welcome message that appears on the website’s starting page (e.g., **Welcome to my homepage!**).

On the other hand, **Content Assignment** allows Template Developers to specify the contents to be shown in (1) **WebComponents**, or in the **Website**’s (2) banner or (3) footer.

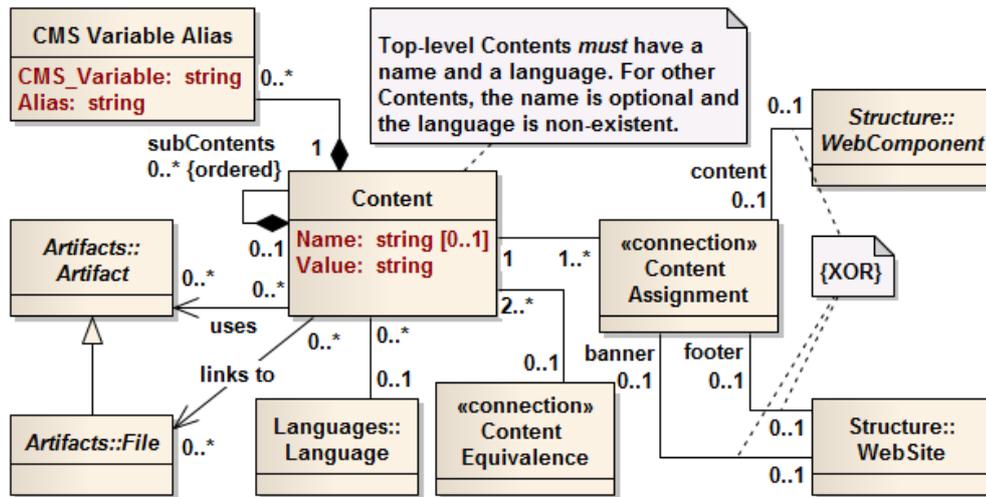


Figure 8.11: Abstract syntax for the Website Template's Contents view.

The reason for supporting these last two assignments – considering that the Structure view does not contemplate the existence of the Website's banner or footer – is that, although CMS administrators are usually unable to change the website's banner or footer itself (unless they have some technical know-how regarding the system's programming infrastructure), typically they are able to change the contents of the banner and footer (i.e., the text to be displayed in these sections of the Dynamic WebPage). Of course, it is important to keep in mind that the interpretation of an assigned Content actually depends on the assignment's target (e.g., a Content assigned to a Forum WebComponent will be interpreted in a different manner than if it were assigned to an HTML WebComponent).

A Content can itself be a container for other Contents. A typical example would be a Forum Thread, with a Content corresponding to a Post and the contained Contents corresponding to replies to that Post. Note that there is no predefined structure for a Content element, and so its interpretation will depend on the structural element(s) to which the Content is assigned.

It is also possible to establish bidirectional Content Equivalence relationships between two or more Contents. This equivalence relationship is important when Contents are localized (i.e., they are written in specific Languages), as it becomes necessary to indicate that a set of Contents, although written in different Languages (such as Portuguese and English), are semantically equivalent to one other.

Furthermore, a CMS-IL Content can also use any number of Artifacts that were defined in the Artifacts view. Contents can use an artifact in one of the following manners: (1) if the Artifact is a String, then its Value is included into the Content's Value; (2) otherwise, the Artifact should be a File and it can either a) be *linked to*, or b) have its *contents* included into the Content's Value (just like a String).

Finally, a **Content** can declare *aliases* for variables that are provided by the CMS, in order to then include the values of those variables into the **Content's Value** (in a manner similar to a **Content's** usage of **String Artifacts**). This is done via the **CMS Variable Alias** concept, which assigns an **Alias** to a **CMS Variable**; the **CMS Variable**, in turn, depends: (1) on the target(s) to which the **Content** is meant to be assigned (e.g., the **WebSite's** banner may provide **CMS Variables**, such as the current user's **GivenName**, that the footer does not); and (2) on the target CMS (e.g., the banner for a specific CMS may provide **CMS Variables** that another CMS does not). Although this dependence on the target CMS could make a CMS-IL model be considered as not being platform-independent (because the **Template Developer** would have to be aware of what is the model's target CMS, and the variables that it provides), this problem can be addressed by using **Annotations** (which are explained in Section 8.6) to specify that the **Content's** target (e.g., the **WebSite's** banner) *must* provide those **CMS Variables**, thus ensuring that the CMS-IL model is still platform-independent.

Listing 8.7 presents some examples of this view's concrete syntax, in particular:

- Line 1 illustrates a simple English **Content** that contains no other **Contents**;
- Lines 3–11 also depicts an English **Content**, but now containing a small set of simple **Contents**;
- Lines 13–16 illustrates a **Content Equivalence** relationship between two **Contents**, in Portuguese and English;
- Lines 18 and 19 show **Content Assignments** that take place between a **Content** and an **HTML WebComponent** and between a **Content** and the **WebSite's** banner, respectively (the definition of **Contents Lorem Ipsum 4** and **Lorem Ipsum 5** is not provided for simplicity);
- Lines 21–25 illustrate a **Content** that references two **Artifacts**, **WelcomeMessage** and **license.txt**; and
- Lines 27–31 illustrate the definition and usage of **CMS Variable Aliases**: (1) line 28 declares an alias, **userFirstName**, for the **CMS Variable** **CurrentUser.GivenName**; (2) line 30 directly outputs the value of the **userFirstName** **Alias**, which in turn outputs the value of the **CurrentUser.GivenName** **CMS Variable**; and (3) line 31 uses the value of the **Alias** **userFirstName** as input for the **user** placeholder in a **WelcomeMessage** **Artifact**.

Finally, we consider it important to elaborate on some particular issues regarding the examples presented in Listing 8.7.

The first issue regards the **Content Value** strings – defined after the **is** token and delimited by the **"** token on both sides of the string – which are present in most of the examples provided. A **Content's Value** is specified as a regular text string (written as typ-

Listing 8.7: Concrete syntax for the WebSite Template's Contents view.

```

1 Content "Lorem Ipsum 1" in "English" is "Consulted perpetual of pronounce me delivered. Too months
   may end change relied who beauty wishes matter."
2
3 Content "Lorem Ipsum 2" in "English" is "
4   You folly taste hoped their above are and but.
5   Dwelling and speedily ignorant any steepest.
6   "
7   contains (
8     Content is "Drawings me opinions returned absolute in."
9     contains Content is "Early to weeks we could."
10    Content is "Cheerful but whatever ladyship disposed yet judgment."
11  )
12
13 Equivalence between
14 Content "Lorem Ipsum 3en" in "English" is "I asked God for a bike, but I know God doesn't work that
   way. So I stole a bike and asked for forgiveness."
15 and
16 Content "Lorem Ipsum 3pt" in "Portuguese" is "Pedi a Deus uma bicicleta, mas sei que Ele não funciona
   assim. Então roubei uma bicicleta e pedi perdão."
17
18 Content "Lorem Ipsum 4" is the Content for WebComponent "Introduction Text"
19 Content "Lorem Ipsum 5" is the Banner Content
20
21 Content "License" in "English" is "
22   >= WelcomeMessage user:'Jack' <
23   License follows:
24   >= license.txt <
25   The full text of this license is also available at: > license.txt <"
26
27 Content "WelcomeMsg" in "English"
28   has variables ("CurrentUser.GivenName" as userFirstName)
29   is "
30   Welcome, >= userFirstName <!
31   >= WelcomeMessage user:userFirstName <"

```

ical Placeholder Strings) in the aforementioned manner; everything (even whitespace) included between the " tokens is a part of the **Value**. The only caveat is that the " token itself cannot be included in the string (although it can be escaped, by writing \").

Another issue concerns the >, >=, and < tokens in lines 21, 24–25, and 30–31. These tokens, unless immediately preceded by the escape character \, indicate the usage of **Artifacts** or **CMS Variables** and are applied as follows:

- The > and < tokens can *only* be used with **Files** (as shown in line 25), and indicate that the corresponding segment will be replaced with a *hyperlink* (or similar mechanism) to allow users to access and download the corresponding file;
- On the other hand, the >= and < tokens – which can be used with any kind of **Artifact**, as depicted in lines 22 and 24 – indicate that the placeholder should be replaced with the *contents* of the used **Artifact**. The manner in which those

contents should be included is CMS-specific, and depends on the nature of the artifact (e.g., a **Binary File** with an image should be included as an HTML image tag that points to the image, and not as a Base64-encoded string).

If the artifact is a **Placeholder String**, then it is possible to provide values for the various placeholders (if any) that have been defined in it, as illustrated in line 22. In the same manner, the contents of **Files** will also be treated as **Placeholder Strings**, as long as the CMS can detect that they contain text; this can be done, for example, by scanning the file to check if it contains only valid ASCII characters, or detecting the existence of an UTF BOM (Byte Order Mark) at the start of the file. If a **File** does not pass that test, then it will be considered as containing binary content and handled in the previously described manner;

- However, the `>=` and `<` tokens do not indicate the usage of **Artifacts** exclusively, as these tokens can also be used to include the value of a **CMS Variable** (as shown in line 30), in the same manner that an **Artifact**'s contents are included in the **Content**. Furthermore, it is possible to use **CMS Variables** as *input for the placeholders* in **Artifacts**, as illustrated in line 31.

It should be mentioned that the semantics for these tokens are inspired by web frameworks that rely on the processing of server-side pages, like Microsoft ASP.NET⁴ or Java Server Pages⁵. In such frameworks, a web page is specified by using HTML in which placeholder markups – typically called *tags* – are included. Each tag can specify that it either (1) contains code to execute (without returning any output value), or (2) contains an invocation of a method or a variable, and the return value of this execution should be included in the page.

The final issue, which is also related to the `"` token delimiting the **Content**'s **Value**, regards the indentation used in the value depicted in lines 3–6. The indentation of each line in a **Content** is determined by using the same rules as *Python's multi-line docstrings*⁶. More specifically, this is done by the following process: (1) considering the set of *all non-blank lines* in the **Content**, determine its *minimum* indentation; and (2) remove that same indentation from all lines in the **Content**. Thus, lines 4–5 are not really indented, because their minimum indentation (two spaces) is removed from *every* line in the **Content**. However, because each line has the same indentation as the others, they are all considered to be starting at the beginning of the line (i.e., with no indentation whatsoever).

⁴<http://www.asp.net> (accessed on February 18th, 2012)

⁵<http://www.oracle.com/technetwork/java/javaee/jsp> (accessed on February 18th, 2012)

⁶Python Documentation, “PEP 257: Docstring Conventions”, <<http://www.python.org/dev/peps/pep-0257/#handling-docstring-indentation>> (accessed on March 15th, 2012)

8.4.8 Visual Themes View

The Visual Themes view – meant for the Web Designer role – determines how each of the web application’s structural elements should be viewed by users, in terms of visual elements such as color, width, or line thickness. This is done by defining CSS (Cascading StyleSheet) classes and specific inline styles, in accordance with the best practices of web design [Mey 06]. The abstract syntax of this view is depicted in Figure 8.12.

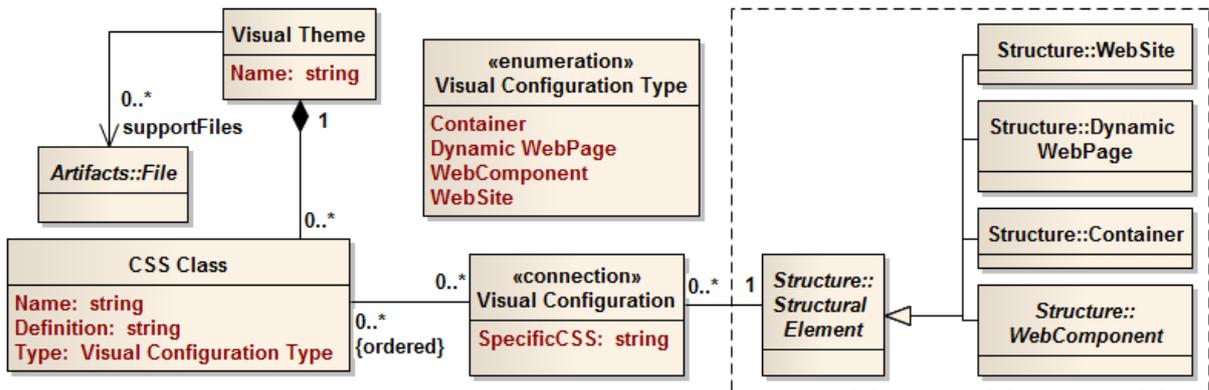


Figure 8.12: Abstract syntax for the WebSite Template’s Visual Themes view.

A **Visual Theme** can be considered as an aggregator for a set of related CSS classes. A **CSS Class** is a concept that maps to a concrete CSS class, which will be used to adjust the rendering of an HTML element in a web browser or web-enabled device. Additionally, a **CSS Class** must be defined for a certain kind of Structural element (to only instances of which it may be applied), which must be one of the values in the **Visual Configuration Type** enumeration: (1) **WebComponent**; (2) **Dynamic WebPage Container**; (3) **Dynamic WebPage**; or (4) **WebSite**.

On the other hand, a **Visual Configuration** can be considered as a specification of CSS visual properties for a certain structural element (e.g., a specific **WebComponent**). It is an association between a structural element and a set of **CSS Classes**, and can be viewed by modelers as *applying the CSS classes to the element* (with eventual customizations to the element, by using the **Visual Configuration**’s **Specific CSS** property to include style information – like background color or strong text – that makes sense in that element but not in the **CSS Class** itself).

Additionally, a **Visual Theme** can reference a set of **Files**, which will be used to support the web application’s visual presentation. Such support **Files** are likely to be images and videos, although they can also be CSS files, Javascript files (e.g., to enable showing elements with rounded corners in older web browsers), or any other files that the Web Designer considers relevant for the web application’s visual.

Listing 8.8 provides some examples of the concrete syntax for the Visual Themes view:

Listing 8.8: Concrete syntax for the WebSite Template's Visual Themes view.

```
1 Visual Theme "Ocean Colors" with
2   CSS Class "Nemo Writing" for WebComponents
3     as "fore-color: orange;"
4   CSS Class "OceanBlue Component" for WebComponents
5     as "width: 100%; back-color: blue;"
6
7   CSS Class "OceanBlue Component"
8     is applied to WebComponent "My Blog"
9     with inline "width: 33%; fore-color: navy;"
10  CSS Class ("OceanBlue Component", "Nemo Writing") is applied to WebComponent "My Blog"
11
12  Supported by file "my_company_logo.jpg"
```

- Lines 2–3 and lines 4–5 depict the definition of two CSS Classes, `Nemo Writing` and `OceanBlue Component`, respectively;
- Lines 7–9 illustrates a Visual Configuration that also contains some Specific CSS (with value `width:33%;fore-color:navy;`);
- Line 10 also illustrates a Visual Configuration, now taking place between a set of CSS Classes and a WebComponent, in such a way that `OceanBlue Component` is applied before `Nemo Writing` (of course, it is possible to provide specific CSS by using the same syntax as in line 9); and
- Line 12 provides an example of the usage of a File as support for the Visual Theme (in this case, it consists of an image with a company's logo, as indicated by the File's name).

It should be noted that the examples in lines 7–10 do not contain the Definition of the CSS Classes `OceanBlue Component` and `Nemo Writing`, as they were already provided in lines 5 and 3, respectively.

As was previously mentioned, a Visual Configuration association can *only* take place between CSS Classes and specific Structure view elements:

- Container-typed CSS Classes (i.e., with the `Type` attribute assuming the value `Container`) and Containers;
- Dynamic `WebPage`-typed CSS Classes and Dynamic WebPages;
- `WebComponent`-typed CSS Classes and WebComponents;
- `WebSite`-typed CSS Classes and the modeled `WebSite` itself.

Thus, non-matching elements (e.g., a `WebComponent` and a CSS Class with `Type` assuming the value `Dynamic WebPage`) cannot be associated by a Visual Configuration element. Although at first this may seem like a senseless restriction (especially to someone with a background in web design, in which the same CSS class is often applied to various types of HTML elements), the rationale for this choice is to *strong-type* CSS classes, in order to

avoid some typical errors (e.g., a CSS class trying to set a property value for an HTML element that doesn't contain that property). Although most web browsers do not alert the user about errors when this kind of mismatch problem occurs (the browser typically just logs the issue and ignores the style assignment), it is a bad practice nonetheless, because different web browsers may interpret the style assignment in different manners, which in turn will often lead to unexpected results.

8.5 Toolkit Modeling

The Toolkit mechanism allows Toolkit Developers to extend a CMS system with additional functionality, namely `WebComponents` and source code (CMS-specific or otherwise) to handle particular events that can happen during the runtime of a CMS system. Unlike the `WebSite Template`, a CMS-IL Toolkit model presents only some similarities to its homonym CMS-ML model; this is because a Toolkit is meant to address behavior, and the target audiences for these two languages typically use different concepts to specify the system's desired behavior (leading to the so-called *semantic gap* [Gho 11]).

A Toolkit model is specified according to a set of views, illustrated in Figure 8.13:

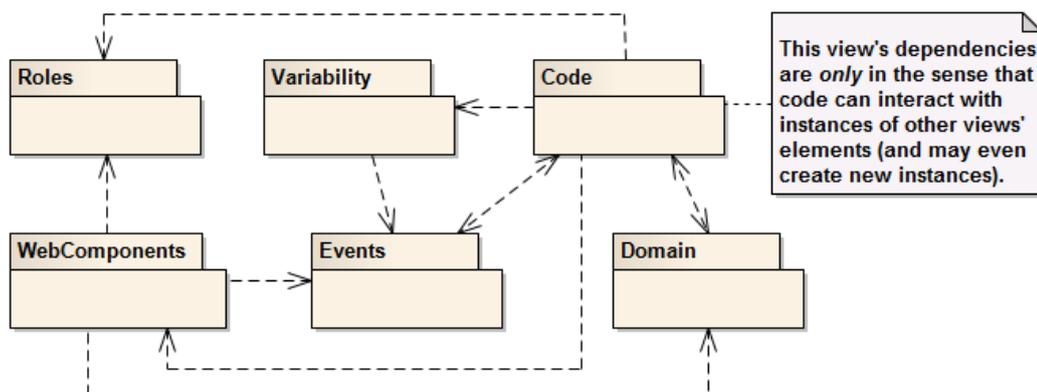


Figure 8.13: Modeling views involved in the definition of a Toolkit.

- The *Roles view*, which defines the user roles that will have the responsibility of performing actions in the context of the Toolkit's `WebComponents`;
- The *Code view*, which defines source code (of either CMS-IL or a CMS-specific programming language) that can be used by other views;
- The *Events view*, in which it is possible to specify additional commands that will be performed whenever specific CMS-related events occur;
- The *Variability view*, which specifies the variability points that will configure the Toolkit's operation;
- The *Domain view*, defining the Toolkit's domain model and its data structure; and

- The *WebComponents view*, specifying new kinds of `WebComponent` (namely their internal elements, and the commands to run when those elements are used).

8.5.1 Roles View

The Toolkit’s Roles view is defined exactly in the same manner as its homonym CMS-ML view, namely by defining two concepts, **Role** and **Role Specialization**. The former is used to specify the expected kinds of participation (and responsibilities) that will take place in the context of the additional functionality provided by the Toolkit, while the latter is used to determine the *specialization* relationships between those kinds of participation. Figure 8.14 depicts the abstract syntax for the Roles view.

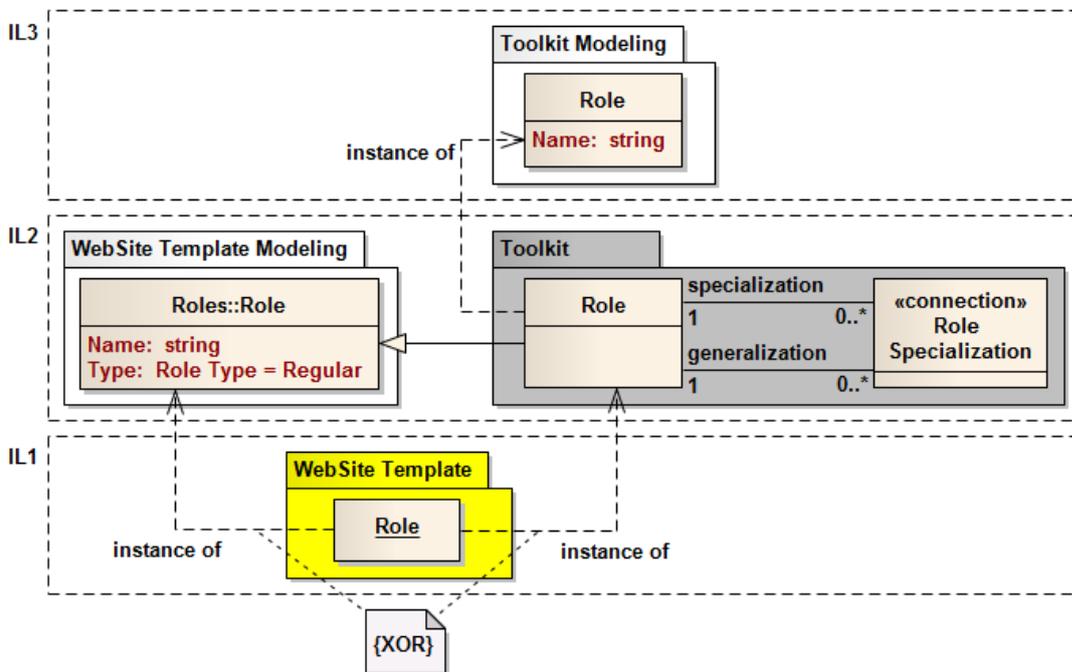


Figure 8.14: Abstract syntax for the Toolkit’s Roles view.

The reason for providing such a simple view is that the `Role` concept is used solely to indicate the *existence* of a kind of participation that the Toolkit will be expecting when someone uses its functionality. The actual specification of what the `Role` can do will be addressed afterward in the Toolkit’s remaining views.

Furthermore, like in CMS-ML, any instance of a Toolkit `Role` is automatically considered a specialization of the `WebSite Template` `Role` concept. Thus, when defining a `WebSite Template`, each modeled `Role` is always an instance of the `WebSite Template`’s `Role` concept, but it may actually be an instance of a Toolkit `Role`.

Listing 8.9 illustrates the concrete syntax for this view: (1) line 1 represents a `Role` that is semantically equivalent to the CMS-ML Toolkit `Role` depicted in Figure 7.13a;

and (2) line 3, semantically equivalent to the example in Figure 7.13b, depicts a `Role Specialization` relationship between two `Roles`, `Document Manager` (the specialization) and `Document Operator` (the generalization). As the example in line 1 shows, a `Toolkit Role` is represented in the same manner as a `WebSite Template's Role`, although it should be noted that these two concepts will be contained within different contexts (a `Toolkit` and a `WebSite Template`, respectively), which removes the possibility of erroneously using one concept instead of the other.

Listing 8.9: Concrete syntax for the Toolkit's Roles view.

```

1 Role "Document Manager"
2
3 Role "Document Manager" specializes Role "Document Operator"

```

8.5.2 Code View

The Code view is what effectively gives CMS-IL its *programming language* characteristics that are familiar to most web application developers (which is CMS-IL's target audience). It can be considered as a set of *function declarations* in conjunction with a *subset of the Ruby* programming language's abstract syntax [FM 08], which in turn makes this view (arguably) the most complex one in CMS-IL. Figure 8.15 presents a simplified representation of the abstract syntax for this view (the `Statement` concept is not further developed because of the similarity with Ruby's own abstract syntax).

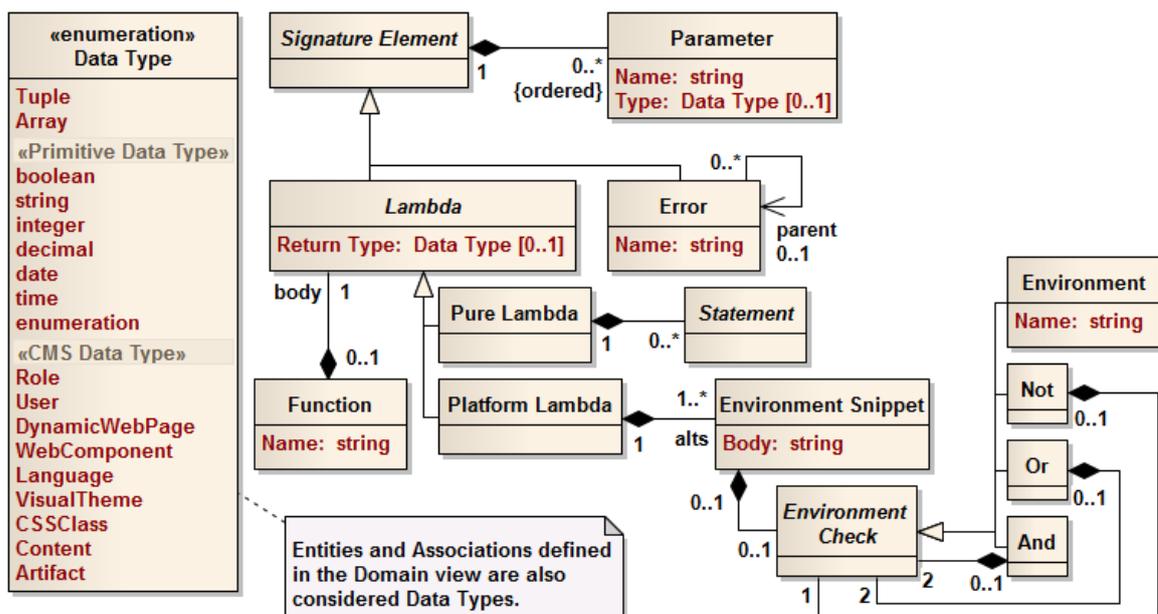


Figure 8.15: Abstract syntax for the Toolkit's Code view.

Although the Code view is presented separately from the other Toolkit views, it is nevertheless the cornerstone for most of those views, as it is possible for elements in them (e.g., **Methods** in the Domain view) to reference or contain elements from the Code view.

This view provides some concepts, of which the most relevant are **Function** and **Lambda**. These concepts can also typically be found – in *some* form – in other programming languages (imperative or otherwise), such as C [KR 88], Java [Sch 11], or Python [Lut 09].

This view starts by defining a set of **Data Types**, which expand the set of CMS-ML’s Domain view **Data Types** with values such as **User**, **Content**, **Tuple**, and **Array**, allowing the Toolkit Developer to reference more kinds of element. Each **Data Type** value is either: (1) a *reference* to a CMS element (e.g., a specific user), in the case of **CMS Data Types**; or (2) a *value* that is *copied* when it is used (e.g., in a **Statement**), in the case of **Primitive Data Types**. This follows the same data categorization strategy as Java [Sch 11] (and other similar languages, such as C# [NEG⁺ 10]), which typically divide data types in two main categories – *reference types* and *value types* – that determine whether an instance has referential identity or not: reference types have referential identity, while value types do not. However, CMS-IL also provides the following **Data Types**: (1) the **enumeration**, which is a **Primitive Data Type** that allows the Toolkit Developer to specify a set of allowed *names* (which, in turn, represent *specific values*) for that **Data Type**; (2) the **Tuple**, which enables the definition of *n-tuples* (i.e., ordered sequences of *n* elements, each of which can be of any **Data Type**, including other **Tuples**); and (3) the **Array**, which is equivalent to *arrays* in other programming languages, and allows the storage of *several values* of a certain **Data Type**. Furthermore, the Toolkit Developer is also allowed to create new **Data Types** (other than **Tuples** and **Arrays**) by specifying **Entities** and **Associations** in the Domain view (which is explained further down this chapter, in Subsection 8.5.5).

The cornerstone of the Code view is the concept of **Lambda**, which is derived from lambda calculus and is a fundamental concept in many functional programming languages, such as Lisp or Scheme [AS 96]. From a mathematical perspective, a lambda function consists of simply of an *anonymous function*, such as $(x, y) \mapsto x * x + y * y$ or $x \mapsto x$ (which correspond to the anonymous forms for the square sum and identity functions, respectively). Similarly, a CMS-IL **Lambda** consists of a *function* – not to be confused with the **Function** concept, which is presented further down this text – that (1) is *anonymous* (i.e., it has no name), (2) receives a (possibly empty) set of **Parameters**, (3) performs some instructions, and (4) returns a corresponding value.

A **Parameter** is a concept equivalent to UML’s **Parameter** [OMG 11.e] or to the concept of *parameter* in Object-Oriented Programming (OOP) [CN 91]. A **Parameter** has only a **Name** and an *optional Type*: the latter, if specified, is only used by the CMS system as a precondition to automatically ensure that a runtime value of the expected **Type** is

assigned to the **Parameter** (e.g., if a **Lambda** expects a **Parameter** of **Type integer**, then it would be incorrect to provide a **string** when invoking that **Lambda**).

A specific **Lambda**'s set of expected **Parameters** is called its **signature** (which is why **Lambda** inherits from the abstract class **Signature Element**), and is used by CMS-IL to check whether a **Lambda** can be used in the context of other elements that can contain **Lambdas**. This check is made by ensuring that there is no *signature conflict* between the **Lambda** and its container element, namely that: (1) their signatures have the same number of **Parameters**; and (2) their **Parameter**'s **Types**, if declared, do not conflict (e.g., a **Lambda** that receives a single **Parameter** of type **Role** should be used with an element whose signature declares a single **Parameter** of type **User**).

A **Lambda**, being a function, also *returns a value*. This value can be one of the following: (1) one of the aforementioned **Data Types**, or even a **Tuple** composed of other **Data Types**; or (2) the special value **nil**. Although languages such as C# [NEG⁺ 10] treat lambdas as *anonymous pieces of code* that may (or may not) return a value, CMS-IL **Lambdas** *always* return a value, even if that value is **nil**; this is similar to Python's functions, which always return a value (when no value is returned by the developer, the function returns the special value **None**). It should be noted that, like in Python, **Lambdas** can take advantage of the **Tuple Data Type** to (1) return *multiple values of different types*, and (2) receive *structures of related values* (e.g., like those that can be defined with C's **struct** [KR 88]) without requiring the definition of multiple **Parameters** (one for each value).

Another fundamental concept in this view is the **Function**. Considering the previous definition of the **Lambda** concept, it suffices to say that defining a **Function** consists simply of *assigning a name to a Lambda*. The reason why this concept is particularly important is that CMS-IL does not support the definition of **Lambdas** on their own (i.e., without being within the context of another element that will *contain* the **Lambda**). Thus, the only way to define a **Lambda** that can be used in multiple locations of the CMS-IL model (which, in turn, enables *code reuse*) is to define a **Function**, which can then be used in most situations that expect a **Lambda** (it is also possible to define an *alias* for a **Lambda**, which in practice is the same as creating a **Function**; aliases are further described in Section 8.7).

A **Function** can contain any kind of **Lambda** provided by CMS-IL. This is relevant because, unlike other programming languages, CMS-IL provides two different kinds of **Lambda**: the **Pure Lambda** and the **Platform Lambda**.

A **Pure Lambda** consists of *platform-independent source code* that can operate not only over instances of the various elements defined in other Toolkit views, but also over the CMS system itself. A **Pure Lambda** is composed of a set of **Statements**, which are *instructions* to be run by the CMS system. CMS-IL supports a *simplified subset* of the

statements defined by the Ruby programming language [FM 08], as CMS-IL does not support the definition of classes⁷: creating data structures is handled by the Domain view (explained in Subsection 8.5.5) and **Tuples**. However, CMS-IL also adds a small set of predefined classes, namely: (1) classes such as **Role** or **User**, for manipulating instances of the Toolkit’s elements; and (2) the class **CMS**, for interacting with the CMS system itself. Of course, it will be the CMS system’s responsibility to dynamically convert these CMS-IL **Statements** into code that can be run on its environment; an example would be the conversion of a **Statement** “<body> **unless** <condition>” [FM 08] into a set of PHP statements, when running on a CMS system such as Drupal or Joomla (which are analyzed in Chapter 5 and Appendix B).

On the other hand, a **Platform Lambda** can be considered to be a *platform-specific Lambda*, and contains a set of **Environment Snippets**. An **Environment Snippet** is a *platform-specific source code segment* (e.g., a string with some Java statements) that is meant to be executed directly by the CMS system in which the model is deployed, and consists merely of a pair <target environment, code to run>. An **Environment Snippet** can also specify an **Environment Check**, which consists of a condition (or set of conditions, organized in a logical boolean manner) that must be satisfied in order to run that **Snippet** (e.g., by ensuring that the current environment consists of Drupal 7 and Java 6 or higher). Because a **Platform Lambda** contains multiple **Environment Snippets** (each for a different language and/or environment), it is possible for a Toolkit Developer to *provide multiple native implementations of the same desired behavior* (e.g., an equivalent set of instructions in Java and in PHP), and the CMS system can choose at runtime which of the implementations to run, according to its own environment, and the **Environment Checks** (if any) that each implementation assumes. The definition of multiple implementations, in turn, can mitigate the **Platform Lambda**’s platform-specificity. Furthermore, if the Toolkit Developer intends to deploy the Toolkit onto a specific CMS, then using that CMS’s API in the CMS-IL model becomes a trivial matter, thus effectively addressing the issue of integrating a model with a system that already exists (an issue for which MDA is typically criticized [Tho 04]).

Finally, CMS-IL also provides a Ruby-based error-handling mechanism. This is embodied by the **Error** concept, which can be considered as equivalent to Ruby’s exceptions [FM 08], and consists of: (1) a **Name**, which allows a certain error to be identified and handled separately from other errors; and (2) being a **Signature Element**, it also contains a set of **Parameters** that serve as arguments for the error (e.g., the environment variables that led to the error’s occurrence). Furthermore, an **Error** can also be a specialization of

⁷Further details of the various kinds of **Statement** available are provided in the “CMS-IL User’s Guide” [SS 11.b].

another `Error` (its *parent*), in which case it can be considered as a particular case of that parent `Error`, and inherits all of the parent's `Parameters`. Errors are raised in CMS-IL by means of the `raise` keyword (which is also present in Ruby), which can be invoked by raising: (1) a previously declared `Error`, in which case the `Error` can (and should) receive its `Parameters`; or (2) a new `Error`, previously *undeclared*, in which case it is not possible to provide any `Parameters` to the `Error`. Likewise, `Errors` can be handled in CMS-IL source code by using `begin/rescue` code blocks (in the same manner as Ruby).

Listing 8.10 depicts some examples of the concrete syntax for the Code view:

Listing 8.10: Concrete syntax for the Toolkit's Code view.

```

1 Error "Blowing Up"
2   with (string errorMessage)
3
4 Error "Pipe Failure" ("Blowing Up")
5   with (string pointOfFailure_Description)
6
7 pipeKaboom = lambda (pipe) -> (string) = {{
8   if pipe.isOK
9     return "All is OK in PipeLand!"
10  if pipe.junctionIsFaulty
11    raise "Pipe Failure", "Boom! There goes the pipe!", "The pipe failed in the junction"
12    raise "Unknown Error"
13  }}
14
15 processWithWebService = lambda (operand1, operand2) =
16   when "PHP 5.3" do [[
17     $client = new SoapClient("http://localhost/WebService.wsdl");
18     $result = $client->getResult(>> operand1 <<, >> operand2 <<);
19     return $result; ]]
20   when "DotNetNuke" do [[
21     using System;
22     using System.Web.Services;
23     HttpRequest req = (HttpRequest) WebRequest.Create("http://localhost/WebService.asmx");
24     (...)
25     WebResponse response = req.GetResponse();
26     (...)
27     return result; ]]
28
29 Enumeration "Morning Type" ("Fine", "Good")
30
31 Function "Greet and Say Something"
32   receives (User user, message, "Morning Type" morningKind)
33   returns boolean
34   performs {{
35     CMS.tellUser "Hello, #{userName}! #{morningType} morning we're having, heh?",
36       userName:user.GivenName, morningType:morningKind
37     CMS.tellUser "Did you know that: #{message}", message:message
38   }}

```

- Lines 1–2 illustrate the declaration of an **Error** called **Blowing Up**, which receives a single **Parameter** (a **string** called **errorMessage**);
- Lines 4–5 declare another **Error**, **Pipe Failure**, which inherits from **Blowing Up** and contains another **Parameter**, **pointOfFailure_Description**;
- Lines 7–13 define a **Pure Lambda** (i.e., a **Lambda** containing CMS-IL platform-independent code, delimited by the `{{` and `}}` tokens) that (1) receives a **Parameter** **pipe** (with no specified **Type**), (2) returns a **string**, and (3) in its body, invokes the methods **isOk** and **junctionIsFaulty** in the **pipe** object. This example also illustrates the raising of **Errors**, more specifically: (1) the **Error Pipe Failure**, previously declared in lines 4–5, is raised in line 11, with some values for its **Parameters**; and (2) an undeclared **Error**, **Unknown Error**, is raised in line 12.

This **Lambda** is also given an *alias*, **pipeKaboom**, which allows the future usage of this **Lambda** without having to explicitly define a **Function** for it (aliases are presented further down this chapter, in Section 8.7);

- On the other hand, lines 15–27 illustrate a **Platform Lambda** (i.e., a **Lambda** that contains platform-specific code, delimited by the `[[` and `]]` tokens) that: (1) receives two **Parameters**, both with unspecified **Types**; (2) does not declare any type for its return value, and so any return value is possible; (3) is prepared to run in either a PHP 5.3 environment or in the DotNetNuke CMS, with PHP code or C# code, respectively; and (4) attempts to contact a web service that will process the values of its two **Parameters** (which are included in the source code with `>>` and `<<`).

Of course, it will be the CMS's responsibility to not only interpret the **Environment Checks**, but also parse the platform-specific code, reorganize it so that it is syntactically correct (e.g., in C#'s case, to distinguish **using** statements from other source code, and adjust the resulting code accordingly), and run it;

- Line 29 defines an **enumeration**, **Morning Type**, containing a set of names (**Fine** and **Good**) that represent the only possible values the **enumeration** may hold; and
- Lines 31–38 depict a simple **Function**, called **Greet And Say Something**, that (1) receives a **User**, an object **message** (which can be of any **Type**), and a **Morning Type** object (the **enumeration** defined in line 29), (2) shows some messages to the current user (note that string interpolation is performed in the same manner as in the **WebSite Template's Contents** view), and (3) returns the **boolean** value **true**.

Additionally, although this **Function** contains a **Pure Lambda**, it could instead contain a **Platform Lambda** and invoke CMS-specific functionality to display those messages to the user.

There are some noteworthy topics to be pointed out regarding the definition of **Functions** and **Pure Lambdas**.

The first topic concerns this view’s concepts from a mathematical perspective. Inspired by functional languages [AS 96], the Code view is meant to be viewed as the underpinnings for the definition of *mathematical functions*, which will be evaluated while the web application is running. Thus, this view defines mostly concepts that are oriented toward this purpose: *lambdas*, *functions*, and function *arguments* (known as **Parameters** in CMS-IL, UML, and object-oriented programming languages). However, it should be noted that we do not consider CMS-IL itself to be a functional language, because it is more oriented toward the changing of state during the web application’s runtime, in the likeness of imperative languages such as C [KR 88]; this mutable state is determined primarily by the **Entities** of the Domain view (described in Subsection 8.5.5), which reflect the web application’s domain model.

Another topic is related to the variables used in the CMS-IL **Statements** that are depicted in Listing 8.10. These use *dynamically typed variables* [Pie 02], which means that the type of a variable does not need to be declared, and is *only determined at runtime*. More specifically, CMS-IL supports the usage of *duck typing* in the definition of **Functions** and **Lambdas**. **Duck typing** consists of a dynamic typing approach in which the possible actions that can be performed over an object (provided as a parameter) are not determined by the object’s *type*, but rather by the functionalities (i.e., properties and methods) that are necessary when using the object. An even better definition (in our own opinion) is the following:

Duck typing allows an object to be passed in to a method that expects a certain type even if it doesn’t inherit from that type. All it has to do is support the methods and properties of the expected type *in use by the method*.⁸

The usage of duck typing in CMS-IL allows Toolkit Developers to specify the web application’s behavior (as **Functions** and **Lambdas**) while concentrating on the functionalities offered by a certain object, instead of focusing on the object’s type. Nevertheless, CMS-IL Toolkit Developers can easily insert some type-verifications (used as *sanity checks*) into **Functions** and **Lambdas**, by optionally specifying the **Types** of their **Parameters** and/or return value (and thus disable duck typing for that **Parameter**). This last feature can help mitigate potential problems that could arise from CMS-IL’s use of duck typing (e.g., two types may each provide a methods called `log`, but with completely different – and unexpected – semantics), as well as facilitate the creation of CMS-IL Toolkits by developers that are more familiar with static typing languages (such as Java [Sch 11]).

The final topic regards the usage of the Code view’s elements by other Toolkit views. More specifically, other views can only (1) define **Pure Lambdas** (instead of the more

⁸Phil Haack, “How Duck Typing Benefits C# Developers”, <<http://haacked.com/archive/2007/08/19/why-duck-typing-matters-to-c-developers.aspx>> (accessed on March 14th, 2012)

generic `Lambda`) and (2) reference `Functions`. The reason why `Platform Lambdas` cannot be directly defined in other views is to ensure that those views are kept as platform-independent as possible (and to keep platform-specific details as localized as possible). In turn, this leads to a CMS-IL model that is potentially easier to adapt to a new CMS system, as the platform-specific elements are located only in a limited set of `Functions`, instead of being scattered throughout the model.

It should also be noted that, whenever a `Pure Lambda` instance is defined in the context of other views, then both its signature (i.e., its ordered set of expected `Parameter Types`) and its `Return Type` – if any – must not conflict with the those of the element that contains it. The Toolkit concrete syntaxes illustrated in this dissertation (and further detailed in the “User’s Guide” [SS 11_b]) often deal with this potential issue in a simple manner, by defining the `Pure Lambda` at the same time as its container element, which effectively applies the `Pure Lambda`’s signature to the element. However, this caveat also applies when an element references a `Function`, and possible alternative concrete syntaxes should be aware of this issue regarding signature conflicts.

8.5.3 Events View

The Events view is one of the views that effectively addresses the *extension of the CMS’s functionalities*. In particular, this view allows developers to provide source code (namely `Pure Lambdas` or `Functions`) to be run whenever a specific event (such as a new user’s registration, or the beginning of a new request) takes place in the CMS system. Figure 8.16 illustrates the abstract syntax for the Events view.

This view is centered around (1) the occurrence of events and (2) the running of event handlers when they do. It should be noted that this merely consists of applying the Observer design pattern [GHJV 95], which is very often used in CMS systems to enable their extension by third-parties [SS 08_b].

The notion of *events that can occur* is captured in CMS-IL by the **Event** concept, which represents events that may occur during the CMS system’s usage. In particular, this view defines two concepts that inherit from **Event**, **CMS Event** and **Custom Event**: the former is used to represent *generic* CMS events that typically occur while the CMS system is used (such as a user’s successful login), while the latter enables the declaration of *new kinds of Toolkit-specific events*. There are several kinds of **CMS Event** defined by CMS-IL, resulting from our analysis of CMS systems and the lifecycle events that they typically allow developers to hook into (these events, some of which have already been mentioned, are further explored in the “User’s Guide” [SS 11_b]). On the other hand, a **Custom Event** consists simply of a **Name**, which identifies the event. Furthermore, because

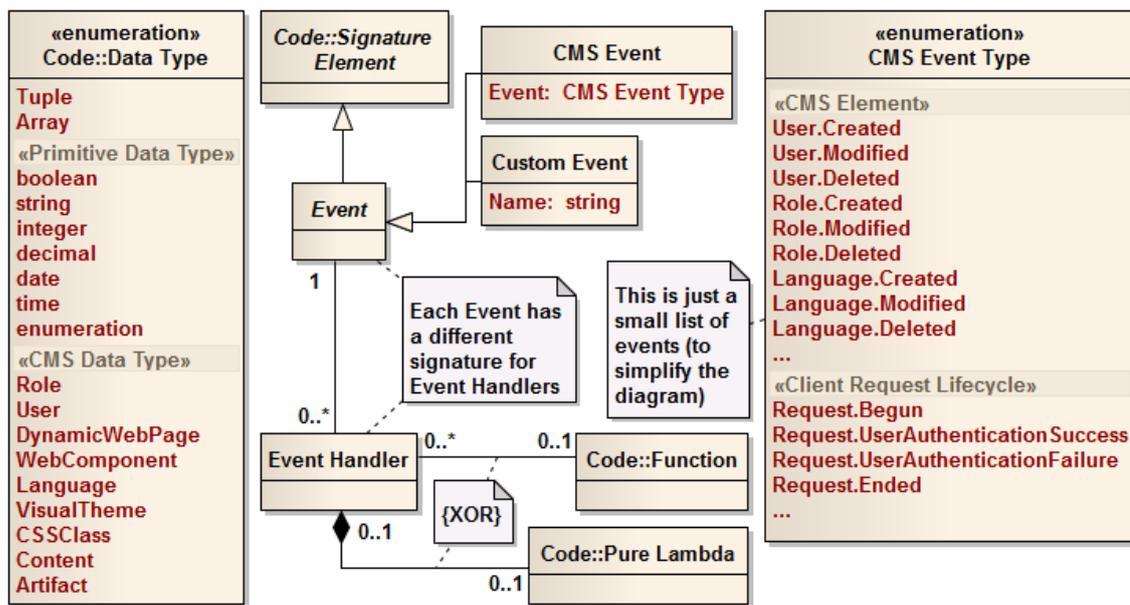


Figure 8.16: Abstract syntax for the Toolkit's Events view.

it is a **Signature Element**, an **Event** declares a set of expected **Parameters** – which are specific to the event itself – that will be provided to any event handler that addresses it (e.g., the `Role.Created` **CMS Event** provides the newly created `Role` as its **Parameter**, so that the corresponding event handlers can perform their own work).

On the other hand, the notion of *event handler* is captured by the **Event Handler** concept (sometimes just called **Handler**, for simplicity), which determines the code that will be run when the corresponding **Event** occurs. Each **Event Handler** either (1) references an already existing **Function** (see Subsection 8.5.2) or (2) provides a **Pure Lambda** that contains the code to run. Furthermore, the signature of the **Event Handler**'s **Function** or **Pure Lambda** must not conflict with its **Event**'s signature, and the value that is returned, if any, is discarded.

Listing 8.11 provides some examples of the concrete syntax for the Events view:

- Line 1 depicts a declaration of a **Custom Event**, `Timer Elapsed`, that provides no **Parameters**;
- Lines 3–4 define another **Custom Event**, called `Magic Has Happened`, that provides two **Parameters** (`magicTrick` with no specified **Type**, and `magician` of **Type** `User`);
- Lines 6–9 illustrate an **Event Handler**, with a **Pure Lambda** (defined in lines 7–9), for the `Magic Has Happened` **Custom Event**. This **Pure Lambda** receives two **Parameters**, `trick` and `performer`, which have no declared **Types** (and thus this **Pure Lambda**'s signature not conflict with the signature for the `Magic Has Happened` **Custom Event**), and simply calls a **Function** `log` that is provided by the `CMS` class; and

Listing 8.11: Concrete syntax for the Toolkit's Events view.

```
1 Event "Timer Elapsed" can occur
2
3 Event "Magic Has Happened" can occur
4   with (magicTrick, User magician)
5
6 When event "Magic Has Happened" occurs
7   do lambda (trick, performer) = {{
8     CMS.log "A magic trick occurred! It was #{trick} and was performed by #{performer}!", trick:trick,
9     performer:performer
10  }}
11 Function "Tell Magic Trick Occurred"
12   receives (theTrick, User whoPerformedTheTrick)
13   performs {{
14     CMS.tellUser "Wow! A magic trick!"
15   }}
16 When event "Magic Has Happened" occurs call "Tell Magic Trick Occurred"
```

- Lines 11–16 illustrate another **Event Handler**, this one referencing a **Function**: lines 11–15 declare a simple **Function Tell Magic Trick Occurred**, while line 16 declares that the **Tell Magic Trick Occurred Function** is a **Handler** for the **Magic Has Happened** event.

It should be mentioned that these two alternatives (the usage of either a **Function** or a **Pure Lambda**) are meant only to facilitate the definition of **Event Handlers**. Once again, it is important to remember that *a **Function** is only used to name a **Lambda***. This leads to a Toolkit Developer being able to either use an existing **Function** (which promotes code reuse), or define a new **Pure Lambda**, which is meant to be used *only* in the context of that **Handler**.

8.5.4 Variability View

The **Variability view** allows the Toolkit Developer to define specific *variability points*, which will assume the form of *configuration options* for the Toolkit, when it is deployed on a CMS system. In turn, it will be up to the CMS system to endow the user with an interface that supports this configuration, according to the options specified in this view. Figure 8.17 illustrates this view's abstract syntax.

This view is fairly simple, as it provides one important concept, **Variability Point** (sometimes called just **Point**, for text brevity), which represents a configuration option that will condition the Toolkit's operation. A **Variability Point** consists simply of: (1) a **Name**; (2) a **Default Value** (a string that can be left unspecified, but if it is specified then it must be parsed by the target CMS system); and (3) a **Type**, which assumes a value from **Variability Point Type**. The latter is very similar to the **Data Type** element of

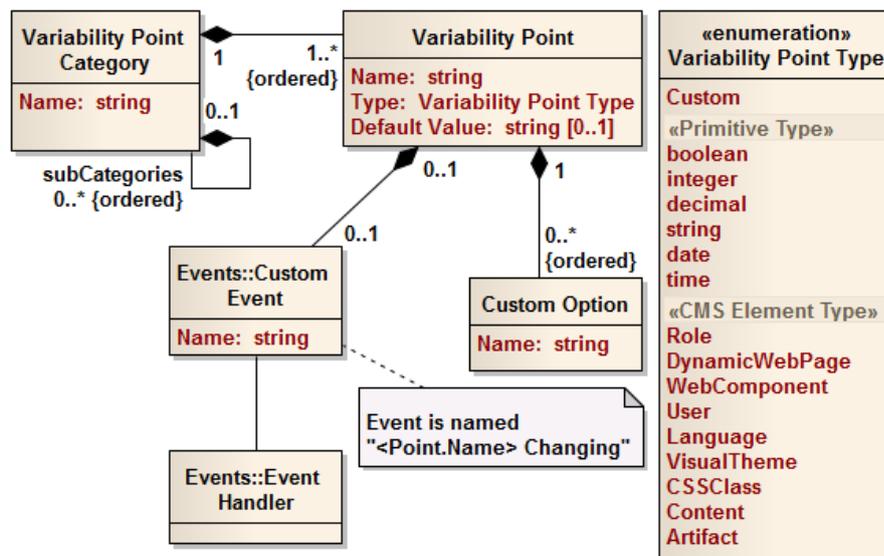


Figure 8.17: Abstract syntax for the Toolkit’s Variability view.

the Code view (see Subsection 8.5.2); however, a **Variability Point** has an additional option to provide an ordered set of **Custom Option** values (which are very much like the Code view’s **enumerations**, and are identified solely by their **Name**), and cannot have a **Tuple** as its **Type**. The reason why the Code view’s **enumeration** is not used here is that a **Custom Option** can *only* be a name, while it is possible for an **enumeration** to assume other kinds of values (further details are available in the “User’s Guide” [SS 11_b]).

Furthermore, each **Variability Point** belongs to a **Variability Point Category** (also just called **Category**), which is just a classifier for **Points**. Toolkit Developers can organize such **Categories** in a tree-like structure, which in turn will allow the Toolkit’s users to configure it in a step-by-step manner, instead of being presented with all available options in a single screen. Each root **Category** (i.e., a **Category** that is not contained by any other **Categories**), in turn, can be *specific* to a kind of **WebComponent** (defined in the **WebComponents** view) or *global* to the Toolkit itself, depending on whether the **Category** is declared within the context of a **WebComponent**.

A **Variability Point** also declares a **Custom Event** (see Subsection 8.5.3) called `<Variability Point’s Name> Changing`, in which the “Variability Point’s Name” string results from the concatenation of the **Point’s Category Names** with the **Point’s Name** itself, using `:` as a separator (e.g., for a **Variability Point Maximum Size** contained in a **Category Recycle Bin**, which in turn is contained within another **Category Document Management**, the name of the corresponding **Custom Event** would be `Document Management:Recycle Bin:Maximum Size Changing`). Although it is not required that Toolkit Developers provide an **Event Handler** for this **Custom Event**, any such **Handlers** will be (1) provided with the **Variability Point’s** new value in a **Parameter**, and (2) run *before* the **Point’s** value

is changed (thus preventing the `Point` from being changed if an error is raised by the `Handler`), which can be useful for purposes such as logging or user-input validation. It is also possible for a Toolkit Developer to provide a `Handler` at the same time that the `Variability Point` is declared, which expedites the process of declaring the `Point` and handling its changes with some validation code.

Listing 8.12 provides an example of this view's concrete syntax:

Listing 8.12: Concrete syntax for the Toolkit's Variability view.

```
1 Variability Category "Document Management Options"
2   defines
3     Point "Role responsible for document management" is Role
4   contains (
5     Category "Operational Parameters" defines (
6       Point "Use version control system" is boolean with default value "true"
7       Point "Maximum number of documents to keep in Recycle Bin" is integer
8         when set do lambda (newValue) = {{ CMS.log newValue }}
9     )
10    Category "Access Control" defines
11      Point "Permissions policy" with
12        Option "Optimistic (access is allowed if any Role allows access)"
13        Option "Pessimistic (access if blocked if any Role denies access)"
14  )
15
16 Function "Do Something With Number"
17   receives (number)
18   performs {{ (...) }}
19 When event "Document Management Options:Operational Parameters:Maximum number of documents to
20   keep in Recycle Bin Changing" occurs
   call "Do Something With Number"
```

- Line 1 defines a `Variability Point Category`, `Document Management Options`, that contains all the other `Categories` and `Variability Points` defined in the example;
- Line 3 presents a `Variability Point`, designated as `Role responsible for document management`, that references a `Role` in the CMS system in which the Toolkit is installed;
- Lines 5 and 10 define two other `Categories`, `Operational Parameters` and `Access Control`, that are sub-categories of `Document Management Options`;
- Line 6 depicts another `boolean Variability Point`, which is called `Use version control system` and has a `Default Value` of `true`;
- Line 7 also illustrates a `Variability Point`, an `integer` called `Maximum number of documents to keep in Recycle Bin`, with a corresponding `Event Handler` consisting of a `Pure Lambda` that simply logs the new value;
- Line 11 portrays another `Variability Point`, `Permissions policy`, that provides a set of `Custom Options`, which are illustrated in lines 12–13; and

- Lines 19–20 specify that the `Function Do Something With Number` (which is defined in lines 16–18) is an additional `Event Handler` for the `Custom Event` that is associated with the `Variability Point` defined in line 7 (as this example shows, the immediate declaration of a `Handler`, as done in line 8, is much cleaner).

It is noteworthy to point out an issue that apparently could arise from defining these `Variability Point` instances in the IL2 metalevel (see Figure 8.3), but having the “instances of the instances” in the IL0 metalevel. More specifically, the issue would be that the `Variability Point` instances in metalevel IL0 (i.e., the *values* for the configuration options) would participate in an instance-of relationship with elements in metalevel IL2 (the `Variability Point` elements defined in the Toolkit itself), which would in turn violate the Strict metamodeling doctrine that CMS-IL’s metamodel architecture is meant to address. However, this apparent issue is not real in practice, because `Variability Points` are present in the IL2, IL1, and IL0 metalevels, although they are *implicit* (as illustrated in Figure 8.18).

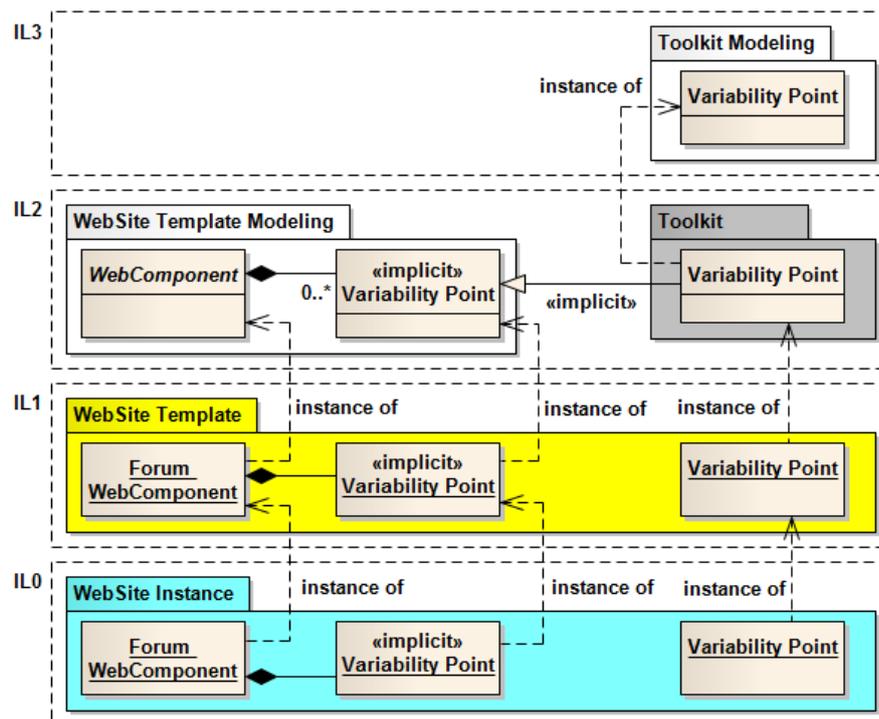


Figure 8.18: A WebSite Template contains implicit Variability Points.

Figure 8.18 shows that *each kind of WebSite Template WebComponent implicitly contains a set of variability points* (i.e., configuration options, which are CMS-specific). An example is the `Forum WebComponent`, for which some CMS systems could provide a variability point such as `Number of posts to show per page`. Nevertheless, specifying such details is not relevant when modeling a WebSite Template (namely because most CMS systems do

not support changing the variability points for `WebComponents` that are already defined), which is why this feature is not included in WebSite Template Modeling. Furthermore, the `Variability Points` defined in a Toolkit will (when possible) take advantage of the same CMS-specific mechanism that is used to store configuration options for predefined `WebComponents`, which means that any instance of Toolkit Modeling's `Variability Point` could be considered a specialization of a `Variability Point` concept in WebSite Template Modeling (if that concept was not just implicit). Thus, it is safe to say that, in practice:

- In IL2, WebSite Template Modeling can be considered to also define an *implicit Variability Point* concept (because most CMS systems also provide such a mechanism), of which any instance of Toolkit Modeling's `Variability Point` can be considered to be a *specialization*, in a manner similar to Toolkit Roles. In other words, this metalevel models variability points for *types of WebComponent* (e.g., `Forum`);
- In IL1, a WebSite Template *implicitly* defines instances of the (also implicit) `Variability Point` in WebSite Template Modeling *or* of the `Variability Point` instances that were specified in the Toolkit. In other words, the Template implicitly defines variability point *slots* for *specific WebComponents* (e.g., `MyBlog`); and
- In IL0, the WebSite Instance's variability point elements actually hold the configuration option *values* for the slots that were (implicitly) defined in the Template.

Because of this, there are no instance-of relationships crossing more than a single metalevel frontier, and so the rule of Strict metamodeling is not broken.

8.5.5 Domain View

The Domain view is defined in a manner similar to its CMS-ML-homonym view, and is used to define the domain `Entities` that will be manipulated by the Toolkit's CMS Event Handlers and `WebComponents` (described in Subsections 8.5.3 and 8.5.6, respectively). Figure 8.19 illustrates the abstract syntax for this view.

As Figure 8.19 shows, there are some significant differences between the Domain views of CMS-ML and CMS-IL, which warrant further explanation.

The most relevant difference is that CMS-IL adds the concept of **Method**, which is equivalent to UML's `Operation` [OMG 11-e], or to the concept of *method* in OOP [CN 91]. An `Entity` can hold a set of `Methods`, which are always invoked in the context of a specific instance of that `Entity`. A `Method` *may* contain a `Pure Lambda` (which is platform-independent, as described in Subsection 8.5.2): if it does not contain a `Pure Lambda`, then the `Method` is considered to be *abstract* (i.e., it is not implemented), and it must be implemented in any non-abstract `Entities` that specialize the `Entity` containing this

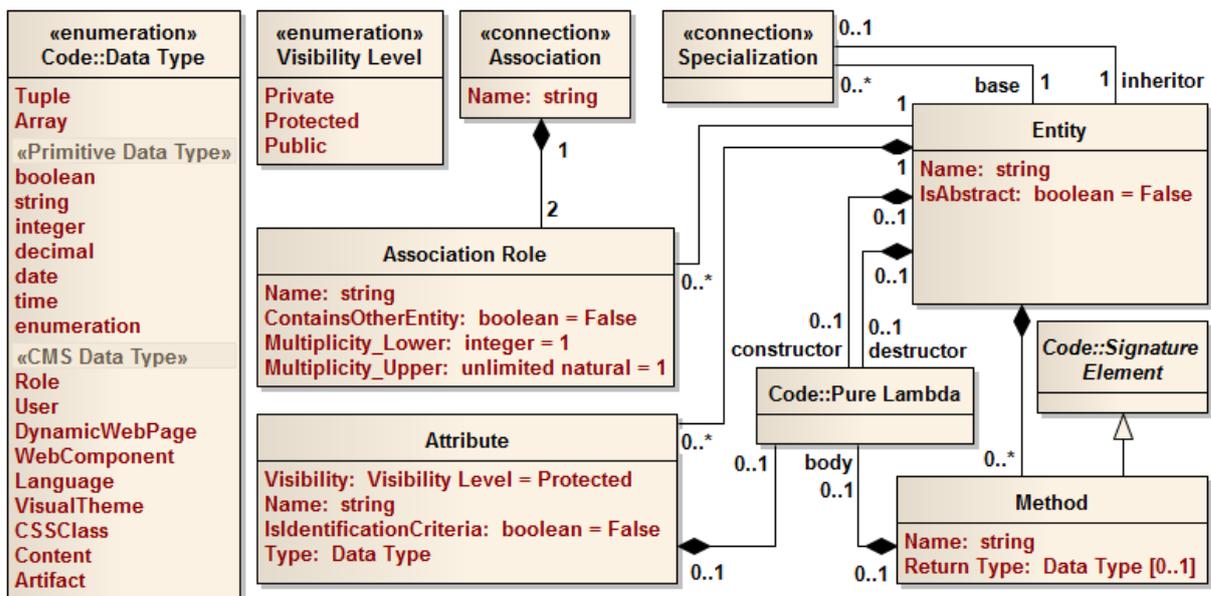


Figure 8.19: Abstract syntax for the Toolkit's Domain view.

abstract `Method` (just like in typical OOP languages). Because it is a `Signature Element`, a `Method` can also receive an ordered set of `Parameters`, and its `Pure Lambda` (if any) can access not only the `Method`'s `Parameters`, but also the `Entity` instance for which it was invoked, with the `self` keyword.

Furthermore, the `Data Types` available in the Domain view are those defined in the Code view (see Subsection 8.5.2). As was previously mentioned, in addition to the `Data Types` defined in CMS-ML's Domain view, CMS-IL expands the set of `CMS Data Types` with values such as `User` or `Content`, which enables the referencing of other CMS elements.

Another difference is that an `Attribute` can also have its value determined by a `Pure Lambda`, instead of being used as a slot for "holding" that value. However, if the `Attribute` uses a `Pure Lambda` to determine its value, then: (1) it is not possible to *assign* values to that `Attribute` (because no such slot exists); and (2) the `Attribute` will not be considered as a value to be stored in the CMS database (or other persistence mechanism that is used). This can be considered as being akin to UML's notion of *derived property* [OMG 11_e], in which the `Property`'s value is *derived* from the values of other `Properties` in the `Classifier`. Nevertheless, from a practical perspective, defining this kind of `Attribute` is equivalent to defining a `Method` that (1) receives no `Parameters` (and can only access the current instance, via the `self` keyword) and (2) returns a value that complies to the `Attribute`'s `Type`.

Furthermore, an `Attribute` can have a **Visibility** that determines whether it can be accessed from other `Entities`, or from any `Methods` or `Functions` that manipulate its parent `Entity`. This `Visibility` can assume the following values: (1) `Private`, meaning

only its **Entity**'s **Methods** can access and manipulate the **Attribute**; (2) **Protected** (the default value), meaning that only the **Methods** of either its **Entity** or of an **Entity** that inherits from that **Entity**; or (3) **Public**, in which case the **Attribute** can be accessed and manipulated by *anything* that can access its **Entity**. These semantics are very much like in other object-oriented programming languages, such as Java [Sch 11], and so we will not elaborate on them in this dissertation.

Finally, an **Entity** can itself contain two **Pure Lambdas** that will act as the **Entity**'s *constructor* and *destructor* (both are optional, and will be invoked after an instance is created and before an instance is destroyed, respectively). Unlike programming languages like Java or C#, CMS-IL only supports the definition of a parameterless constructor in any **Entity**. This is because such constructors are responsible only for *initializing the new instance* – accessed via the **self** keyword – with default values (which is why **Attribute** does not provide a default value property), or other initialization operations (such as logging or inserting the instance into a registry).

Listing 8.13 presents some examples of the concrete syntax for the Domain view. These examples are almost semantically equivalent to the CMS-ML examples that were represented in Figure 7.17, namely:

Listing 8.13: Concrete syntax for the Toolkit's Domain view.

```
1 Entity "Person" has
2   Attribute "Name" as identification string
3   birthDate = Attribute "Date of birth" as date
4   Attribute "ID Number" as integer
5   Attribute "CMS User" as User
6   contacts = Attribute "Contacts" as string[1..*]
7   Attribute "Age" as integer given by {{
8     result = date.today - self.birthDate
9     result = result.inYears
10    return result
11  }}
12 Method "getFirstContact"
13   returns string
14   performs {{
15     result = self.contacts.atIndex 0
16     return result
17   }}
18
19 Entity "Person" (as 1 "Owner") is associated with Entity "Credit Card" (as * "Card") as "Owns"
20
21 Entity "Folder" (as "Container") contains Entity "Document" (as * "Content") as "Holds"
22
23 Entity "Person" inherits from Entity "Animal"
```

- Lines 1–17 declares an **Entity** that is similar to the example CMS-ML **Entity** depicted in Figure 7.17a. However, this example also defines (1) an additional

derived **Attribute** *Age*, in lines 7–11, that uses a simple calculation to determine the person’s age, and (2) a **Method**, *getFirstContact*, that receives no **Parameters**, returns a **string**, and performs a simple indexing operation, in lines 12–17;

- Line 19, which is equivalent to Figure 7.17b, defines a typical **Association** between **Person** and **Credit Card**;
- Line 21 also defines an **Association** (but now with containment semantics) between **Folder** and **Document**, and is equivalent to Figure 7.17c; and
- Line 23 establishes a **Specialization** relationship between **Person** and **Animal**, and is equivalent to Figure 7.17d.

It should be noted that, although the **Pure Lambdas** depicted in lines 7–11 and 14–17 of Listing 8.13 both use *dynamically typed variables* [Pie 02] (i.e., they use variables, called **result** in both cases, whose type is only checked at runtime), the **Attributes** defined in lines 2–11 can be considered as *statically typed* (because their types are declared and checked at design time). The reason why **Attributes** are statically typed is because, when the CMS-IL model is deployed, they will be also used by the CMS system to define the database schema with which to store their values. If a dynamic typing mechanism was to be used when declaring **Attributes**, this would require that the CMS system either (1) do a “simulated run” of the model, in order to discover the type of the first value that is assigned to the **Attribute** (and risk further problems if another value, of a different type, was assigned later on to that same **Attribute**), or (2) use a storage mechanism that stores *not only the value but also its type* (such as SQLite⁹); both of these options pose considerable problems and constraints, which is why the static typing mechanism is used.

Another noteworthy issue is that the abstract syntax and the concrete syntax for a **Method** are almost the same as for a **Function** (see Subsection 8.5.2). The reason for this similarity is that these two concepts are almost equivalent, the only differences between them being that: (1) a **Function** is not “attached” to any element (i.e., it just exists, and can be invoked from any point in the Toolkit), while a **Method** is part of an **Entity**, and can only be invoked in the context of an instance of that **Entity**; and (2) a **Function**’s **Body** *must* be specified, and can be *any* kind of **Lambda** (including a **Platform Lambda**), while a **Method**’s implementation – if any – must be a **Pure Lambda** (which, by nature, is platform-independent and easier to specify and validate). Thus, when considering the differences and similarities between these two concepts, the rationale for these syntaxes is to minimize the differences between them, in order to facilitate the Toolkit Developer’s definition of **Methods** and **Functions** without requiring the learning of two distinct concrete syntaxes.

⁹<http://www.sqlite.org> (accessed on February 18th, 2012)

8.5.6 WebComponents View

The WebComponents view, as its CMS-ML-homonym, allows the Toolkit Developer to specify new kinds of `WebComponent` that can be used afterward in WebSite Templates. This view is where the *functionalities of the CMS are extended*, along with the Events view (described in Subsection 8.5.3). Like the CMS-ML view, it is centered around the definition of `WebComponents` and `Support WebPages`; however, this view also includes the definition of `Web Event Handlers`, which are a specialization of the `Event Handler` concept defined in the Events view. Figure 8.20 illustrates a simplified representation of the abstract syntax for this view (a more detailed specification is available in the “User’s Guide” [SS 11.b]).

Toolkit Developers act on this view by defining `WebComponents`, as well as `Support WebPages` for those `WebComponents`. As Figure 8.20 shows, the Toolkit’s `WebComponent` concept is particularly important, because its *instances* are automatically considered *specializations* of the WebSite Template Modeling’s `WebComponent`; in turn, this specialization allows Template Developers to create WebSite Templates in which those new kinds of `WebComponent` are used, instead of being limited to the generic kinds provided out-of-the-box by CMS-IL.

In the WebSite Template’s Structure view (see Subsection 8.4.1), a `WebComponent` was considered a *basic unit of functionality* that the Template Developer would place in `Dynamic WebPages`. However, in this view, a **WebComponent** consists of a user interface that will (1) interact with the user, (2) manipulate the web application’s domain model (defined in the Domain view, see Subsection 8.5.5), and (3) issue instructions to be performed by the CMS. These interactions will be performed in a manner that helps the user to fulfill a set of tasks (which are identified in CMS-ML, but not in CMS-IL, as developers are typically not responsible for defining business-oriented tasks, but rather for implementing the functionality that *supports* those tasks). Furthermore, instances of the `WebComponents` defined by the Toolkit Developer will be the aforementioned units of functionality that will be placed by Template Developers.

A **Support WebPage** is also a user interface that will perform some work. However, it is different from a `WebComponent`, in that: (1) it consists of a *web page* (i.e., a node, with a URL, to which a web browser can navigate); and (2) it *supports* a `WebComponent`, namely by addressing some operations of the `WebComponent`’s tasks. A typical example of a `Support WebPage` in a Document Management System (DMS) web application would be a web page to edit a document’s metadata.

Each `WebComponent` and `Support WebPage` is a **WebInteractionSpace**, which can be regarded as a “canvas” that will display information to the user, and with which the user will interact. This “canvas”, in turn, endows `WebComponent` and `Support WebPage`

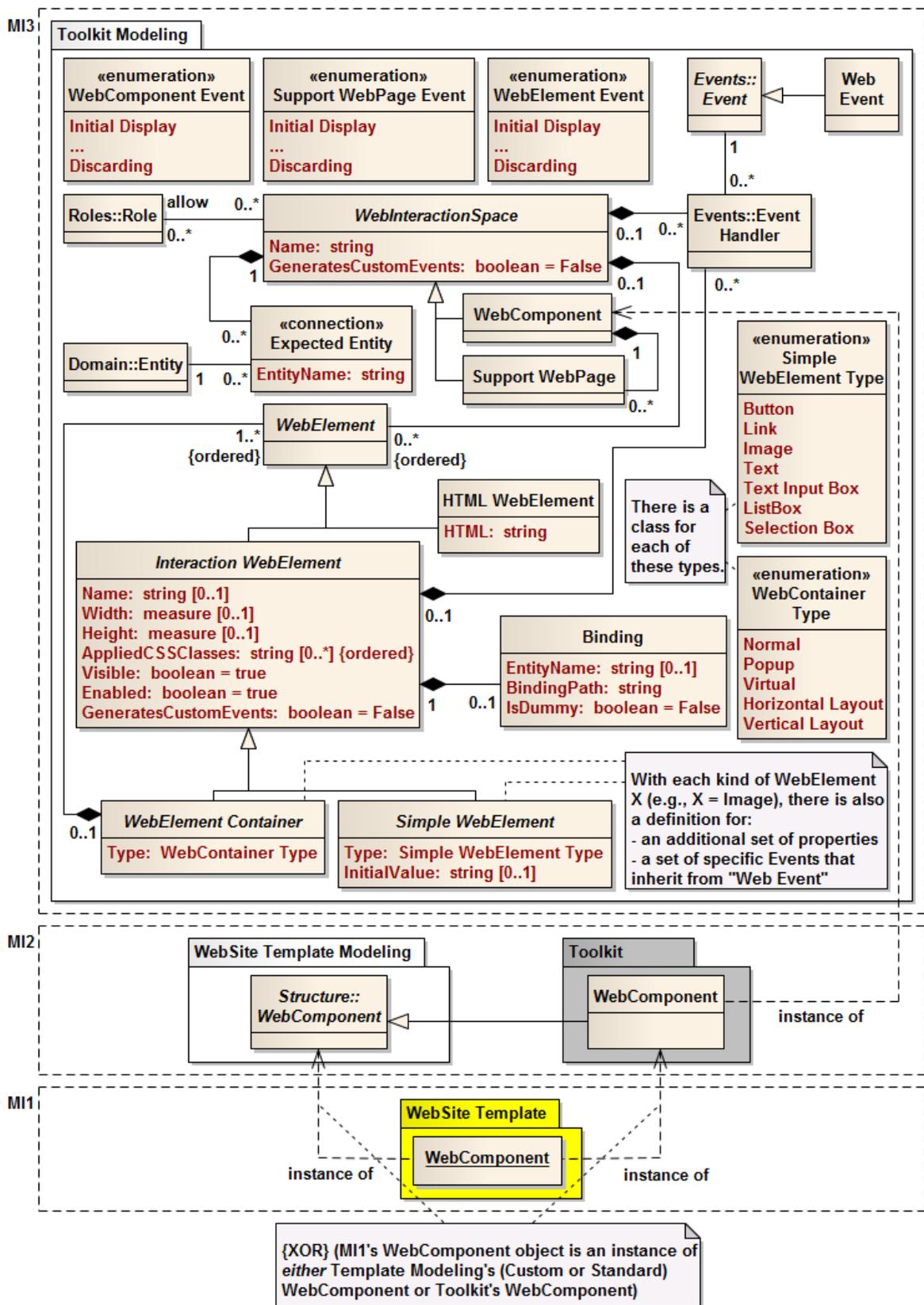


Figure 8.20: Abstract syntax for the Toolkit's WebComponents view.

with some noteworthy features, namely: (1) it can be accessed by a limited set of **Roles** (if none are specified, then it is freely accessible by anyone); (2) it can receive a set of **Entities**, to which it assigns *names* that identify them within the **WebInteractionSpace**; and (3) it contains a set of **WebElements**, which are placed on specific positions of the **WebInteractionSpace**.

The first feature is used to ensure that only specific **Roles** can access potentially sensitive pages. It is nothing more than syntactic sugar for specifying an **Event Handler** that runs when the **WebInteractionSpace** is initialized, and ensures that the current user's set of roles include any of the **Roles** expected by the **WebInteractionSpace**. This is why we will not explore this feature further in this dissertation.

The second feature is addressed by the **Expected Entity** concept, which is used to provide *arguments* to the **WebInteractionSpace**, in order to specify over which parts of the domain model it will operate. If a **WebInteractionSpace** is to manipulate (i.e., access and/or change) a certain instance I_X of a domain **Entity** X , then I_X must be accessible by one of the following ways: (1) I_X is itself provided by an **Expected Entity**; or (2) I_X is accessible by traversing one or more **Associations** from an instance I_Y of another **Entity** Y , I_Y being provided by an **Expected Entity**. From a practical perspective, **Expected Entities** are typically implemented as HTTP request parameters (either GET query string parameters or POST form values).

The last feature is what effectively addresses the *user interaction* aspect of the **WebInteractionSpace**. More specifically, this aspect is performed by the **WebElements** that are contained within the **WebInteractionSpace**. Like in CMS-ML, a **WebElement** consists of an element that will be rendered in HTML (so that it can be displayed in a web browser) and with which the user may be able to interact (e.g., entering some values in a set of form fields, and clicking on a button to submit those values to the web application). A **WebElement** consists of: (1) an optional **Name** that identifies the element; (2) an optional **Width** and **Height**, which respectively specify the desired width and height for the **WebElement**; (3) an ordered set of CSS class names (not to be confused with the **CSS Class** concept defined in the **WebSite Template's Visual Themes** view) that should be considered as being applied to instances of the **WebElement** (so that its visual aspect can be configured afterward in **WebSite Templates**); (4) a property **Visible**, which can be changed at runtime and determines whether any user that can view the **WebElement's** parent element can also view the **WebElement** itself; and (5) a property **Enabled**, which is similar to **Visible**, but instead determines whether users can *interact* with the **WebElement**. From a practical perspective, a **WebElement** can be considered as the CMS-IL equivalent to a control in the Microsoft ASP.NET [MF 10] or in Java Server Pages [RLJ⁺ 03] frameworks.

CMS-IL defines a variety of `WebElement` types (which can also be found in CMS-ML), namely (1) `HTML WebElement`, (2) `Simple WebElement`, and (3) `WebElement Container`.

An **HTML WebElement** consists of a simple piece of HTML that will be included when its parent (a `WebInteractionSpace` or `WebElement Container`) is rendered in a web browser; due to its nature, it is not possible for users to interact with this `WebElement` (unless the HTML string itself defines browser elements with which the user can interact).

On the other hand, a **Simple WebElement** is the *basic element of user interaction* that can be placed on a `WebInteractionSpace`. There are various kinds of `Simple WebElement` defined by CMS-IL – namely (1) `Button`, (2) `Link`, (3) `Image`, (4) `Text`, (5) `Text Input Box`, (6) `ListBox`, and (7) `Selection Box` – all of which are also present in CMS-ML. However, unlike CMS-ML (which defines a `Simple WebElement` concept supported by a `Simple WebElement Type` enumeration), *CMS-IL defines a concept for each of these types* (these are not represented in Figure 8.20 for simplicity); this is because each of these concepts will have its own behavior and will provide a specific set of events, in addition to the events that are common to all `Simple WebElements` (e.g., an `Image WebElement` should not provide the same events as a `Text Input Box WebElement`, because they have completely different natures and purposes).

Furthermore, a **WebElement Container** (also called just *container* in this text, for brevity) is a `WebElement` that is meant to *contain other WebElements*. CMS-IL defines a variety of `WebElement Container` kinds that is different from CMS-ML’s, as it provides the (1) `Normal`, (2) `Popup`, (3) `Virtual`, (4) `Horizontal Layout`, and (5) `Vertical Layout` concepts (but CMS-ML’s `List` and `Table` are not present, and the `Binding` container is ultimately replaced by the `Virtual` container). The `Horizontal` and `Vertical Layout` containers, which are not present in CMS-ML, can also be used to replace the `Binding` container, but their main purpose is to assist the Toolkit Developer in establishing a basic visual layout (either horizontal or vertical, respectively) for the container’s `WebElements`.

The reason for CMS-IL offering less `WebElement Container` concepts than CMS-ML is that the `List` and `Table` containers are used *only* to (1) determine the HTML that *wraps* its contained `WebElements` (in selection lists and tables, respectively), and (2) optionally establish a new `Binding` context. However, web application developers are often used to specifying such HTML by hand, and so they are likely more inclined to define such tables and lists by (1) using a `Normal` or `Virtual` container, (2) including `WebElements` to represent the information that should be represented, and (3) including some `HTML WebElements` that contain the intended HTML to wrap the previously mentioned `WebElements`¹⁰. Nevertheless, CMS-ML `Lists` and `Tables` are transformed into a CMS-IL `Virtual WebElement`

¹⁰This would involve an in-depth technical discussion regarding the definition of such HTML structures, which is out of the scope of this dissertation. Further details are available in the “CMS-IL User’s Guide” [SS 11.b].

Container, when the model synchronization mechanism between the two languages is further detailed in Chapter 10.

Each concept that is either a **WebElement** or a **WebInteractionSpace** also provides a number of **Web Events**, which are **Events** (see Subsection 8.5.3) that (1) signal particular occurrences that may take place during the lifetime of the concept's instances, and (2) provide, in the first **Parameter**, the instance that triggered the event. Thus, each concept's nature determines: (1) the specific set of **Web Events** that are provided by that concept (e.g., a **Button** provides a **Web Event** which signals that the user has clicked on the corresponding web browser button); and (2) the *signature* of each of those **Web Events**. Likewise, **WebElements** and **WebInteractionSpaces** can also have a number of **Event Handlers** for those **Web Events** that they provide.

Finally, the abstract syntax presented in Figure 8.20 could present a limitation regarding the definition of **Event Handlers** for **WebElements** or **WebInteractionSpaces**, as it would become necessary to define *all* of those **Event Handlers** in the same model in which these elements are specified. In turn, this would mean that *other* Toolkit Developers would not be able to add their own **Event Handlers** to these elements (e.g., to send an e-mail when the user clicks on a certain button), unless they could change the original Toolkit.

This is why **WebElement** and **WebInteractionSpace** define the **Generates Custom Events** property: this property specifies whether its element (1) generates a set of **Custom Events** that accurately mimics the set of **Web Events** made available by that element, and (2) automatically invokes those new **Events** when the corresponding **Web Events** take place. This effectively establishes a level of indirection that other Toolkit Developers can take advantage of, by providing their own **Custom Event Handlers** to address those **Custom Events**. Although this property is **False** by default (i.e., an element does not automatically generate new **Custom Events**), it nevertheless allows Toolkit Developers to easily provide *specific extensibility points*. It should also be mentioned that this feature is syntactic sugar for manually (1) specifying new **Custom Events** and (2) invoking those **Custom Events** from the **Event Handlers** associated with the element's **Web Events**.

Listing 8.14 presents a simple example of the concrete syntax for this view:

- Lines 1–17 illustrate a **WebComponent**, **Manage Documents**, that:
 - generates **Custom Events** (due to the **Customizable** keyword);
 - invokes the **Function Manage Documents Is Displayed** when it is displayed to the user; and
 - contains a set of **WebElements** – some of which with no **Name** – that either:
 - (1) have only layout purposes (the **Horizontal Layout** and **Vertical Layout** containers);
 - (2) are used to define HTML strings (the **HTML WebElements** that hold the values ``, ``, ``, and ``);
 - (3) are used only as

Listing 8.14: Concrete syntax for the Toolkit's WebComponents view.

```

1 Customizable WebComponent "Manage Documents"
2   as <
3     HorizontalLayout <
4       VirtualContainer <
5         #" <ul>"
6         VirtualContainer "Document List" bound to "Document.all" < #" <li>" Text "Name" bound to
          ".Name" #" </li>" >
7         #" </ul>" >
8       VerticalLayout <
9         Link "Create Document" to Page "Edit Document"
10        Button "Edit Document" on "Click" do lambda (sender) = {{ document = $Document
          List$.current ; CMS.goToPage($Edit Document$, document) }}
11        Button "Delete Document"
12          on "Click" do lambda (sender) = {{ document = $Document List$.current ; document.delete }}
13      >
14    >
15  >
16  on "Initial Display" call "Manage Documents Is Displayed"
17  is supported by
18  WebPage "Edit Document"
19    expects Entity "Document" as "document"
20    as <
21      HorizontalLayout <
22        Text "Name:"
23        TextBox "Name Value" bound to "document.Name" >
24      HorizontalLayout <
25        Text "Description:"
26        TextBox "Description Value" bound to "document.Description" >
27      HorizontalLayout <
28        Button "Cancel" on "Click" do lambda (sender) = {{ CMS.goBackToFrontEnd }}
29        Button "Confirm"
30          on "Click" do lambda (sender) = {{ $document$.save ; CMS.goBackToFrontEnd }} >
31    >

```

containers for other **WebElements** (the anonymous **Virtual** container in line 4); (4) establish a **Binding** that provides a set of **Document Entities** (the **Document List Virtual** container depicted in line 6); or (5) are **Simple WebElements** that display information (the **Text** element) or receive user input (the **Link** and **Button** elements);

- Lines 18–31 depict a **Support WebPage**, **Edit Document**, that (1) supports the **Manage Documents WebComponent**, (2) expects a **Document Entity**, which it calls **document**, and (3) displays – and also allows editing of – the values of **document's Attributes**, by using a set of **WebElements** which are bound to those **Attributes**.

It should be noted that the **Expected Entity** concept does not automatically ensure that a corresponding **Entity** instance is received by the **WebInteractionSpace**. Toolkit Developers can ensure that such an instance is received by checking whether the **Expected**

Entity’s variable (e.g., `document` in line 19 of Listing 8.14) has the value `nil` (e.g., by inserting the verification `if document == nil`).

Furthermore, although the example in Listing 8.14 is similar to the CMS-ML one presented in Figures 7.21a and 7.21b, they are *not* equivalent: the former contains further details concerning the `WebElement`’s `Bindings` and their events, and the latter provides a richer description of the visual layout of those `WebElements` (in CMS-IL, such visual details are typically relegated to CSS styles, as dictated by web design’s best practices [Mey 06]).

8.6 WebSite Annotations Modeling

Like CMS-ML, CMS-IL also provides a WebSite Annotations mechanism that allows WebSite Template Developers to provide platform-specific configuration instructions (which, in turn, should be processed by the target CMS system itself, if possible). This mechanism follows the same strategy as its CMS-ML-homonym, because they are founded on the same principles and share the same purpose (i.e., defining concepts that reference and decorate WebSite Template concepts). Figure 8.21 illustrates the abstract syntax for WebSite Annotations Modeling.

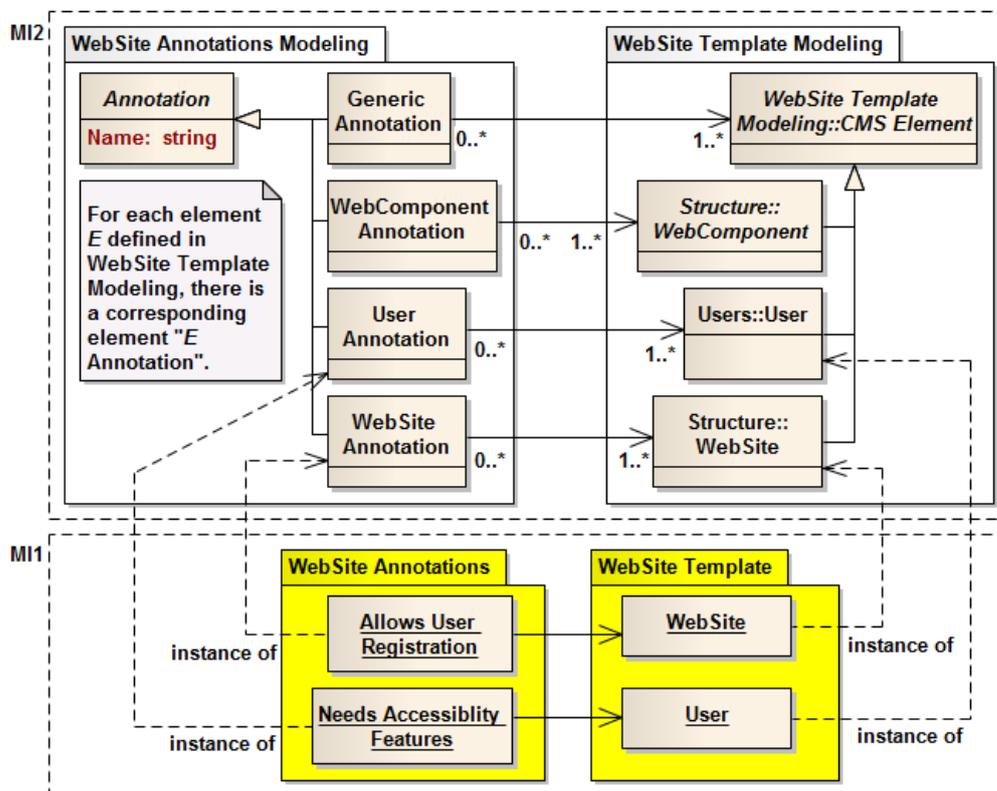


Figure 8.21: Abstract syntax for the WebSite Annotations model.

Listing 8.15 depicts some examples of applying `Annotations` to Template elements:

Listing 8.15: Concrete syntax for Annotations.

```

1 @" Allows Content Subscription"
2 WebComponent "Student Forum" of Standard type "Forum" (...)
3
4 @" Allows User Registration"
5 @" Use Caching"
6 WebSite "WebSite" (...)
7
8 @" Needs Accessibility Features"
9 User " John Doe" ("john.doe@company.com")

```

- The example on line 1, which is semantically equivalent to the one in Figure 7.29a, uses the `WebComponent` Annotation `Allows Content Subscription` to annotate the `Student Forum WebComponent` (the Annotation itself is represented in the form `@<Annotation name>`, just before the corresponding Template element’s declaration);
- On the other hand, line 4 (semantically equivalent to the example in Figure 7.29b) applies the `WebSite` Annotation `Allows User Registration` to a `WebSite`;
- Line 5 applies another `WebSite` Annotation, `Use Caching`, to that same `WebSite`; and
- Finally, line 8 applies the `User` Annotation `Needs Accessibility Features` (suggested in Figure 8.21) to the `User John Doe`.

This concrete syntax is based on the one for Java’s Annotation mechanism [Sch 11], as they serve similar purposes, namely to define metadata that can be applied to their elements (CMS-IL `WebSite` Template elements and Java source code elements, respectively). They present some differences, nevertheless, as Java allows the definition of parameterized annotations and CMS-IL does not. The reason for this shortcoming is that the definition of Annotations with parameters would require that the Annotation’s source code (meant to be run on the target CMS, and thus extend its functionalities) be defined in the IL1 metalevel. However, extending a CMS’s functionalities is the *Toolkit’s* objective (in metalevel IL2), for which it defines the views described in Section 8.5. Thus, the usage of parametrized annotations in CMS-IL would lead to a *loss of abstraction*, when considering the objectives of *extending the CMS system vs. configuring it*.

8.7 Defining Aliases

As most figures in this chapter have shown, CMS-IL elements are uniquely identified by their *names*, which are surrounded by double quotes (e.g., “`Morning Type`”, “`Timer Elapsed`”) to remove any possible ambiguity derived from the use of whitespace characters. However, although this may make a CMS-IL model into something that is near plain

English (and thus easier to read by developers not experienced with CMS-IL), developers with more programming experience are likely to find the constant usage of double quote-surrounded identifiers cumbersome and counterproductive, as they increase the number of characters that are required to create a complete and correct CMS-IL model.

To address this potential issue, we have added the possibility of defining **aliases** for any instance of CMS-IL's WebSite Template Modeling and Toolkit Modeling's concepts. An alias consists simply of an identifier (a set of alphanumeric characters, with no whitespace between them), and is defined in the same manner as a *variable* in mainstream programming languages such as C [KR 88]. However, the reason we call these *aliases* (rather than *variables*) is that an alias is supposed to do just that: serve as an *alternate name* for a certain instance, in such a manner that it can then be used *anywhere* where the corresponding element's name would be used.

Listing 8.16 illustrates some situations in which the aliases can be defined. More specifically, an alias can be defined:

Listing 8.16: Defining CMS-IL aliases.

```
1 sayHowdy = Function "Greet and Say Something" (...)  
2  
3 sayHowdy = Function "Greet and Say Something"  
4  
5 sayHello = sayHowdy
```

- *When the corresponding instance is being declared.* Line 1 indicates that the **Function Greet and Say Something**, which is being declared (the full body for this **Function** was previously depicted in Listing 8.10), will also have an alias **sayHowdy**;
- *As an alternate name for an already existing instance.* Line 3 indicates that the **Function Greet and Say Something**, already declared elsewhere in the model, will have an alias **sayHowdy** (in practice, line 1 would be equivalent to declaring the **Function**, and afterward defining the alias for the **Function**); and
- *As another alternate name for an already existing alias.* Line 5 defines an alias **sayHello** for the previously defined alias **sayHowdy**. In practice, this is equivalent to writing **sayHello = Function "Greet and Say Something"** (like in line 3, but with a different alias name). Furthermore, **sayHello** can then be used in any situation where **sayHowdy** could be used.

8.8 Reminders

As was discussed in Chapter 7, CMS-ML provides the concepts of **Additional CMS Feature** and **Additional Toolkit Feature** (see Section 7.7). These allow the **WebSite**

Template Designer and the Toolkit Designer, respectively, to indicate additional *desired features* that could not be modeled using the concepts provided by CMS-ML.

CMS-IL does not define such **Additional Feature** concepts. This is because it is not up to CMS-IL’s target audience – web application developers – to determine the desired features (much less its requirements); instead, those should be specified by the intended users, namely business stakeholders, which are CMS-ML’s target audience.

Nevertheless, CMS-IL does define a similar concept: **Reminders**. There are actually two different kinds of **Reminder**: the **CMS Reminder** and the **Toolkit Reminder**. Figure 8.22 illustrates how these **Reminder** concepts are related to other elements in the language (of course, just as in CMS-ML, all **WebSite Template Modeling** and **Toolkit Modeling** concepts inherit from the **CMS Element** and **Toolkit Element** concepts, respectively).

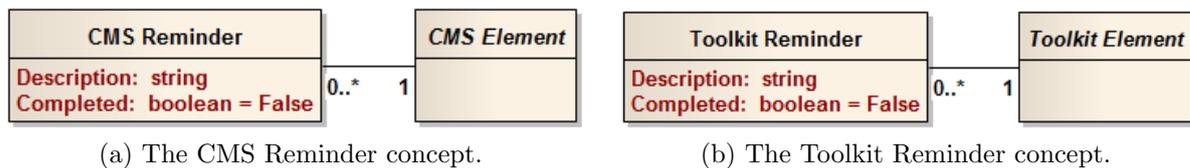


Figure 8.22: Abstract syntax for CMS-IL Reminders.

A **CMS Reminder** is used only within **WebSite Templates** (in the IL1 metalevel), and indicates a specific element of the model which the **WebSite Template Developer** must be particularly mindful of (e.g., it can indicate that a certain **WebComponent** instance must show advertisements according to a specific algorithm). A **Toolkit Reminder** serves a very similar purpose, but it is only used within **Toolkits** (in the IL2 metalevel).

These **Reminders** should not be confused with *comments*, which are used only to *document* the model. In fact, both of these **Reminder** concepts contain not only a **Description**, but also a property **Completed**, which flags *whether the Reminder has already been addressed* (e.g., if a **Toolkit WebComponent** with the aforementioned algorithm has been defined, and if the **Template’s WebComponent** instance is afterward classified as an instance of that **Toolkit WebComponent**). With this feature, it is possible for a **Template Developer** or **Toolkit Developer**, respectively, to determine whether a model contains any **Reminders** to be addressed (i.e., changes that must be made to the CMS-IL model), in order to make it comply with some intended meaning (according to our approach, described in Chapter 6, this meaning would be the one expressed by business stakeholders in a corresponding CMS-ML model).

Listing 8.17 depicts some examples of the concrete syntax for these **Reminder** concepts:(1) line 1 shows a **CMS Reminder** that is applied to a **Role** and has already been addressed (i.e., its **Completed** property has the value **True**); and (2) line 4 also shows a **CMS**

Listing 8.17: Concrete syntax for CMS-IL Reminders.

```

1 !rem>This Role must be assigned to the CMS's most important administrator user.<!
2 Role "Big Boss"
3
4 !todo>This must rotate through a predefined set of TV channels, on a 5 minute-per-channel basis.<!
5 WebComponent "My TV Viewer" of Custom type "WebTV Receiver"

```

Reminder, although this one is applied to a **WebComponent** and has not been addressed yet. **Toolkit Reminders** are represented *exactly* in the same manner.

It should be noted that, although these examples are very similar to the ones in Figures 7.31b and 7.31a (respectively), they *are not equivalent*. More specifically, and in addition to the previously mentioned reasons, line 1 is applied to a **Template Role**, while Figure 7.31b is applied to a **Toolkit Role**.

Furthermore, the reason for **CMS Reminders** and **Toolkit Reminders** being represented with the same syntax is that: (1) there is no danger of erroneously using one instead of the other, because they are located in different metalevels; and (2) this syntax sharing facilitates the learning of the language, as it is not necessary to learn two different syntaxes for nearly equivalent concepts.

8.9 Importing Toolkits

Just as in CMS-ML, it is possible for Toolkits to be imported by **WebSite Templates** and by **Toolkits**. This is done by following the same strategy as CMS-ML, namely by defining two **Toolkit Import** concepts, respectively located in metalevels IL2 and IL3. Figure 8.23 illustrates the abstract syntax for these two concepts.

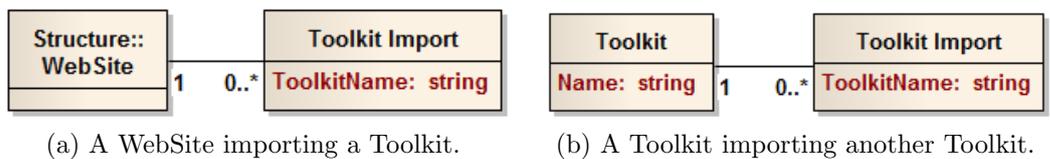


Figure 8.23: Abstract syntax for the CMS-IL Toolkit Import mechanism.

Listing 8.18 illustrates the concrete syntax for these two **Toolkit Import** concepts. More specifically, lines 1–3 depict how a **WebSite** (in the context of a **WebSite Template**) imports **Toolkits**, while line 5 shows a **Toolkit** importing another **Toolkit**.

It should be noted that the **Toolkits** are imported by specifying their *names*. The rationale for this is the same as for CMS-ML, namely to (1) solve the metalevel-boundary problem that would arise from a possible relationship between **Templates** and **Toolkit**,

Listing 8.18: Concrete syntax for Toolkit Import elements.

```

1 WebSite "My Personal WebSite"
2   imports Toolkit "WebTV Toolkit"
3   imports Toolkit "Document Management Toolkit"
4
5 Toolkit "WebTV Toolkit" imports Toolkit "Resource Manager Toolkit"

```

and (2) potentially reduce the effort of modeling a Toolkit, because imported Toolkits do not need to be defined using CMS-IL. This last reason also enables the usage of elements defined in another Toolkit (e.g., a CMS extension that has been previously developed with a traditional web application development approach), which addresses the issue of integrating the CMS-IL model with existing systems that have been previously installed on the CMS system.

The usage of Toolkit elements as new concepts for WebSite Template modeling is also similar to the manner in which it is performed in CMS-ML. As was already mentioned (and shown in Figures 8.14 and 8.20), only Toolkit `Role` and `WebComponent` instances can be considered as new concepts for `Template Roles` and `WebComponents`, respectively.

Listing 8.19 illustrates the concrete syntax for the usage of these new concepts in a WebSite Template: (1) line 1 (equivalent to Figure 7.34b) declares a `Role` that is an instance of a `Role Document Manager`, defined in a Toolkit `DMS` that is imported by the current Template; and (2) line 3 (equivalent to Figure 7.34a) declares a `WebComponent` that instances the `TV Receiver WebComponent` in another imported Toolkit, `WebTV`. Note that the `::` token is used to separate the names of the Toolkit and the element.

Listing 8.19: Concrete syntax for defining WebSite Template elements as instances of Toolkit elements.

```

1 Role "MySite Document Manager" ("DMS"::"Document Manager")
2
3 WebComponent "My Favorite TV" ("WebTV"::"TV Receiver")

```

Furthermore, it is also possible for a Toolkit *B*'s elements to use and/or refine another Toolkit *A*'s elements. CMS-IL only supports the refinement of elements in the *Roles and Domain views*; it is also possible to use (by name) elements from *all Toolkit views*, thus enabling scenarios such as a `Function` in Toolkit *B* creates an instance of a `Role` defined in Toolkit *A*. However, aliases defined in a Toolkit cannot be used outside of that Toolkit, even if the Toolkit is imported by a Template or another Toolkit; this limitation is meant to avoid potential errors that could occur if aliases with the same names were defined in different Toolkits.

Listing 8.20 depicts some examples of how to use (or refine) elements defined in another Toolkit. More specifically: (1) line 1 illustrates a `Role Specialization` between two

Roles, one of them defined in another Toolkit (**WebTV**); (2) lines 3 and 4 are semantically equivalent to the example in Figure 7.34c, and respectively represent a **Specialization** and an **Association** between different **Entities**, some of which are defined in different Toolkits; and (3) line 6 indicates that an **Event Magic Has Happened**, defined in a Toolkit **MagicTricks**, will also be handled by the **Function Send Magic Email** (which is defined in the current Toolkit).

Listing 8.20: Concrete syntax for using or refining elements from another Toolkit.

```
1 Role "Cable Client" specializes Role "WebTV" :: "TV Viewer"
2
3 Entity "Document" inherits from Entity "Resources" :: "Resource"
4 Entity "Document" (as 1..* "Documents") is associated with Entity "Entities" :: "Person" (as 1 "Author")
   as "Author"
5
6 When event "MagicTricks" :: "Magic Has Happened" occurs call "Send Magic Email"
```

Summary

In this chapter, we have presented the CMS-IL modeling language, which is low-level in relation to CMS-ML, but still platform-independent. Nevertheless, there are fundamental differences between CMS-ML and CMS-IL, namely: (1) CMS-IL is completely textual; (2) CMS-IL is intended for use by technical stakeholders (e.g., developers) who are responsible for specifying the web application's *how*, rather than the *what*; (3) CMS-IL is more focused on expressive power, rather than presenting a small set of modeling elements that non-technical stakeholders can easily learn and use (which is the case with CMS-ML); and (4) CMS-IL does not provide **Additional Feature** concepts (although it does provide **Reminders** to indicate that some part of the model is still unaddressed). Furthermore, like CMS-ML, CMS-IL provides an extensibility mechanism (also called Toolkit), which enables the addition of new modeling elements to the language in a controlled manner.

However, now that these CMS-oriented languages have been presented, some issues remain, namely: (1) ensuring that two models, specified using CMS-ML and CMS-IL respectively, are semantically equivalent; and (2) ensuring that this semantic equivalence relationship is maintained throughout model changes, even if those changes are applied by designers to one or *both* models.

In the next chapter, we present the MYNK model synchronization language, the remaining component of the approach described in Chapter 6. This language allows us to not only derive a CMS-IL model from a CMS-ML model (and vice versa), but also to ensure that changes to one (or even both) of the models will be propagated to the other model, thus maintaining consistency between the two models.

Chapter 9

MYNK: Model Synchronization Framework

A complex system that works is invariably found to have evolved from a simple system that worked. The inverse proposition also appears to be true: A complex system designed from scratch never works and cannot be made to work.

Systemantics: How Systems Really Work and How They Fail

JOHN GALL

The CMS-ML and CMS-IL languages (presented in Chapters 7 and 8, and respectively meant for business stakeholders and developers) are integral components of the solution proposed in Chapter 6. However, although these languages can address the modeling needs of their target audiences when creating new CMS-based web applications, there is still the need for a mechanism that (1) supports the obtainment of CMS-IL models from CMS-ML models, and (2) ensures *both kinds of model are kept consistent with one another*. A lack of such a mechanism would mean that models in different languages (e.g., CMS-ML and CMS-IL) would have to be *manually synchronized*, which would have the drawbacks of: (1) requiring additional effort to perform that manual synchronization task; and (2) becoming an error-prone task, because of its repetitive nature.

The **MYNK** (Model sYNchronization framewoRK) model synchronization language, the remaining component of this development approach, was created to address this need. MYNK allows not only the obtainment of a CMS-IL model from a CMS-ML model (and

vice versa), but also ensures that changes to one of the models – or even to both models – will be propagated to the other model, in order to maintain consistency between the two.

In this chapter, we provide a brief description of MYNK. More specifically, this chapter describes: (1) the rationale for defining the MYNK language; (2) the approach used by MYNK to maintain models consistent with each other; (3) MYNK’s relationship with the ReMMM metamodel; (4) the language’s abstract syntax and concrete syntax; and (5) some aspects regarding conflict resolution, and the compatibility between MYNK and modeling languages. For additional information regarding this language, we also advise readers to consult the “MYNK User’s Guide” [SS 11_c].

9.1 Current Model Transformation Issues

Although there are model transformation languages and frameworks¹ available (some of which have been analyzed in Chapter 3), we have found none that effectively supports scenarios in which *both the source and target models have suffered changes* that made them inconsistent with each other.

It should be noted that, in the context of this dissertation, there is an explicit difference between *model transformation* and *model synchronization*. A **model transformation** receives a source model S and *outputs* a target model O ; it is also possible for a model transformation to receive a source model S and a target model T , and *change* T to become equivalent to S . On the other hand, a **model synchronization** receives *two* models A and B , and *changes* both models so that they become equivalent to each other (if possible).

Furthermore, a model transformation can be considered as a special case of a model synchronization. Typical model transformations (such as those defined using ATL), which receive a source model S_1 and output another model O_1 – specified in languages S and O , respectively –, are in practice equivalent to (1) starting with a *blank* target model O_0 (i.e., the model is new and has no content), and (2) changing O_0 to become equivalent to S_1 , and thus obtain a different model O_1 . Even QVT transformations (which receive a source and a target model, and may change the target model accordingly) operate as a model synchronization mechanism, because they (1) analyze the source model and the target model, (2) determine the inconsistencies between them, and (3) if they can change the target model (i.e., if they are **enforceable**), change (or remove) those inconsistencies.

Although these analyzed languages can support the modification of a model B_1 to become equivalent to a model A , this typically implies that some changes made to B_1

¹In this chapter, we use the terms “language” and “framework” in an interchangeable manner, because most model transformation languages are often defined with a homonym framework that supports it (e.g., ATL). Any exceptions to this will be explicitly mentioned in the text.

(especially changes to elements that have also been changed in A) will be *lost*. Figure 9.1 provides a tentative illustration of this problem.

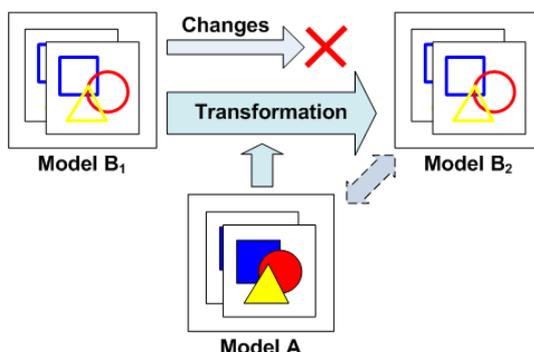


Figure 9.1: Typical model transformations may lose changes.

This is a particularly relevant limitation for the proposed CMS-based web application development approach (previously described in Chapter 6), which contemplates that both the *Business Designer* and the *System Designer* (the roles contemplated by this approach) may be making changes to their respective models at the same time. Obviously, it would be unacceptable for explicit changes, made by either role, to just be lost in translation.

This problem can also be found in related work explicitly focused on approaches for model synchronization. One of these approaches [XLH⁺ 07] uses ATL to perform model synchronization in a manner similar to QVT, by defining relatively small transformations which will then change a target model to be equivalent to a source model. On the other hand, [GW 09] addresses bidirectional model synchronization and explicitly defines the notion of *correspondence*, which establishes trace relationships between source and target models. However, all these approaches still carry the potential for *loss of information* in the target model, if that information *conflicts* in some manner with the one in the source model; furthermore, because the approach described in [GW 09] is *bidirectional*, this information loss can also be extended to the *source model*, further complicating matters.

This potential loss is addressed in [CS 07] by defining the concepts of **Synchronization Decision** and **Synchronization Action**, which consist of decision points (and possible actions) that allow the user to choose a course of action when such conflicts are found. However, these decision points are only associated to model elements (**Artefacts**), which can cause problems when decisions may span several such elements.

9.2 MYNK Overview

In the context of the proposed development approach, MYNK addresses the following objectives: (1) ensuring that two CMS-ML and CMS-IL models are semantically equivalent

to one another; and (2) ensuring that this semantic equivalence relationship is maintained *throughout model changes*, regardless of those changes being applied to one or both models.

Furthermore, MYNK considers a model to be not just a simplified representation of a certain reality in its *latest state*, but rather a sum of its *history* (i.e., the various changes that it suffered until it got to the current state). Of course, this does not hinder the definition of model *snapshots* (at specific points in the model’s lifecycle), that can be used by someone to understand the reality being modeled.

It is important to note beforehand that MYNK is not intended to enable *round-trip engineering* scenarios, an overview of which is illustrated in Figure 9.2a: (1) considering a model A_1 (specified in a certain language A), a model B_1 is obtained via some mechanism (e.g., a model transformation T_{A-B} , from A_1 to B_1); (2) likewise, by using a model transformation T_{B-A} , a model A_2 is obtained from B_1 . If the transformations T_{A-B} and T_{B-A} are *exactly symmetrical* (i.e., they are the opposite of one another), then A_1 and A_2 should be semantically equivalent (in the sense that they should have the same meaning). However, this kind of scenario presents the caveat of requiring that both languages have the same level of expressiveness, as otherwise information would be lost between the various transformations (and so T_{A-B} could never be symmetrical to T_{B-A} , and vice versa).

Instead, MYNK aims at enabling scenarios like the one depicted in Figure 9.2b. This scenario starts with two semantically equivalent models, ML_1 and IL_1 , modeled with the CMS-ML and CMS-IL languages respectively. After the Business Designer makes changes (marked C_{ML_1} in the figure) to ML_1 , in order to obtain a different model ML_2 , the model synchronization mechanism will receive those changes and generate the corresponding set of changes (marked C_{IL_1}) in the CMS-IL language. C_{IL_1} is then applied to IL_1 , originating a different model IL_2 that is intended to be semantically equivalent to ML_2 .

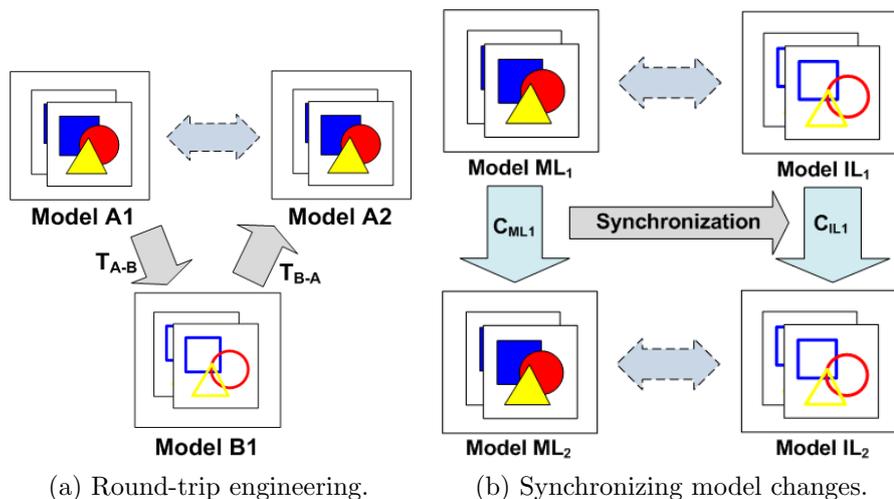


Figure 9.2: Possible synchronization scenarios.

However, in practice it is not certain that two CMS-ML and CMS-IL models will be in sync after *just a single change propagation* (in fact, it is most unlikely). This is because there may still be changes to be applied to the CMS-ML model, derived from (1) changes made to the CMS-IL model (by the System Designer) or (2) changes that are implied by the initial $CMS-ML \rightarrow CMS-IL$ change propagation operation (which is illustrated in Figure 9.2b). More specifically, the model synchronization operation should consist of a $CMS-IL \rightarrow CMS-ML \implies CMS-ML \rightarrow CMS-IL$ cycle (not illustrated in Figure 9.2b for simplicity), which should be performed until a *fixed-point* is reached (i.e., the application of a set of changes to a certain model results in that same model).

The way in which MYNK operates can be considered analogous to a variety of tools and practices, of which we highlight some illustrative examples in the following paragraphs.

The notion of *capture-replay testing* consists of recording events and applying them again, at a later point in time. These tests are often used in automated software testing scenarios (particularly in regression and UI testing), and are done by (1) recording a sequence of events and their expected results, (2) simulating the occurrence of those recorded events, and (3) evaluating whether the results are equivalent to the ones expected. An analogy can be found in MYNK, when considering such events to be model changes.

MYNK's change propagation process is also very similar to what happens in a *Revision Control System (RCS)* such as Subversion or Mercurial (see Subsection 3.3.3): the only information that goes to – or comes from – the developer's *working copy* is a set of *deltas* (i.e., differences) between the most recent version of the artifact and the version that is on the developer's working copy. A developer can (1) make changes to her own working copy, and (2) *merge* changes from other developers' working copies with her own, in order to keep synchronized with the rest of the development team.

Another adequate metaphor for this process is the *pantograph*, a tool that links two pens so that any movement to one of the pens is automatically reproduced (possibly with a different scale) by the other pen. This metaphor is particularly interesting, as MYNK operates much like a pantograph, by taking changes to one model and automatically reproducing them on another model (e.g., CMS-ML and CMS-IL, respectively).

9.3 MYNK and the ReMMM Metamodel

MYNK assumes that models (along with their metamodels) can be specified using the ReMMM (meta)metamodel. This is because MYNK “sees” any model as a set of `Model Elements` which are *linked* to each other, in a graph-like manner. Figure 9.3 provides a tentative illustration of the manner in which MYNK interprets models: Figure 9.3a (adapted from Figure 8.14, see Chapter 8) reflects a CMS-IL Toolkit `Role` instance and

the relationship to its metaclass from a UML perspective, while Figure 9.3b reflects that same model but from MYNK’s perspective.

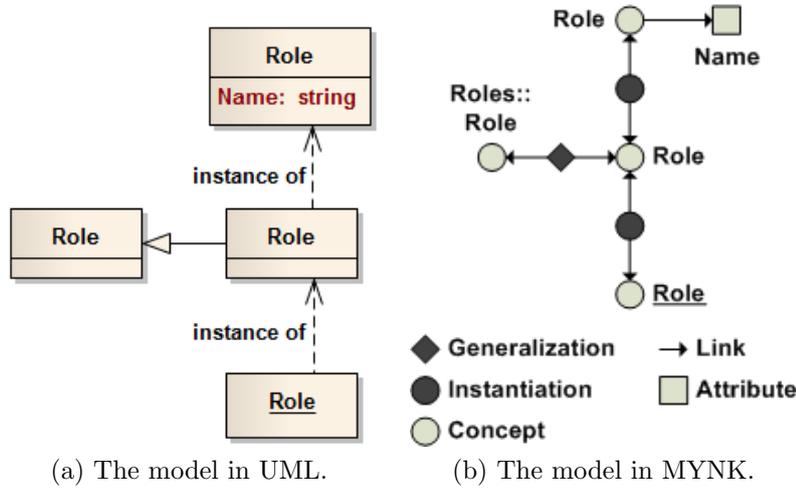


Figure 9.3: How the same model is viewed by MYNK and UML.

It is important to note that, in MYNK, there is no distinction between elements of different metalevels. As Figure 9.3 suggests, the elements from *all metalevels* (i.e., from the model, metamodel, metametamodel, etc.) are all considered to be in the *same model*, and connected by the **Instantiation** relationship (provided by ReMMM, and represented in Figure 9.3b as black-filled circles). Thus, MYNK considers both the ontological and linguistic types of the instance-of relationship, as (1) the ontological instance-of is supported by the **Instantiation** relationship, and (2) the linguistic instance-of is subjacent to each of the **Model Elements** that are interpreted by MYNK (e.g., each of the black-filled circles in Figure 9.3b represents an *instance* of ReMMM’s **Instantiation** metaclass, as shown by the figure’s legend). This is ultimately an application of the Library metaphor (explained in Chapter 2 and illustrated in Figure 2.14b), in which a small set of elements – the ReMMM metamodel – is used to define elements in various metalevels – the model.

9.4 Abstract Syntax

The MYNK language, although not very complex (regarding the *number* of concepts that it defines), organizes its concepts into a set of modules, according to the principle of separation of concerns². These modules are:

²We do not call these *views*, unlike CMS-ML and CMS-IL, because MYNK synchronization developers will use all of these concepts at the same time. However, this division makes it easier to explain and understand the MYNK language.

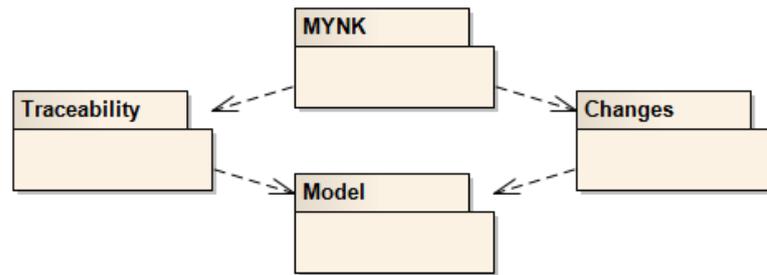


Figure 9.4: Overview of the MYNK language.

- *Model*, which defines the basic concepts to which MYNK-compatible languages must comply;
- *Traceability*, which establishes the concepts for traceability between model elements;
- *Changes*, which specifies the possible changes (or deltas) that can occur on MYNK-compatible models, and which are the foundation for MYNK’s change propagation mechanism; and
- *MYNK*, which uses the other modules to establish a coherent model synchronization mechanism.

Figure 9.4 illustrates the relationship between these modules (like in Chapters 7 and 8, we use UML to describe MYNK’s abstract syntax, for the same reasons).

These modules will be further explained in the remainder of this section. Nevertheless, interested readers may also consult the “MYNK User’s Guide” [SS 11_c] for a detailed description of these concepts.

9.4.1 Model

The *Model* module is the fundamental building block for the MYNK language, as it defines the **MYNK Artifact** concept that (1) is used in every other MYNK module and (2) determines the set of characteristics to which languages must comply if they are meant to be MYNK-compatible. Furthermore, this module also establishes the connection between the MYNK language and the ReMMM metamodel (which was explained in Chapter 6). Figure 9.5 depicts the abstract syntax for this module.

The aforementioned **MYNK Artifact** (or just **Artifact**, for brevity) is used to represent *something* (e.g., a model element, or even the model itself) that MYNK synchronizations can handle, and over which they will operate.

When considering that MYNK assumes models to be specified using the ReMMM metamodel, it follows that **MYNK Artifact** and ReMMM’s **Model Element** should be related. In fact, the relationship between them is that any **Model Element** instance should be *classifiable* as a **MYNK Artifact** (i.e., the former should exhibit a set of characteristics

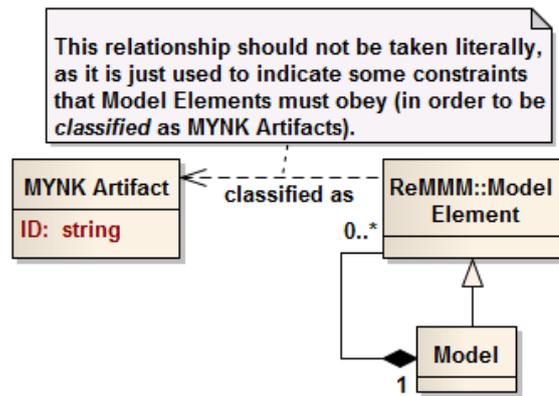


Figure 9.5: Abstract syntax for the Model module.

that would also allow it to be considered as an `Artifact`). However, from a practical perspective, MYNK synchronization developers only need to be aware that each `Artifact` instance corresponds to a specific `Model Element` instance.

Each `Artifact` instance is also *uniquely identifiable* by an ID of type `string`; this ID can be anything, such as a name – which must be unique within the model – or a UUID (Universally Unique Identifier), as long as it unambiguously identifies a *specific Model Element*.

9.4.2 Traceability

The *Traceability* module, although relatively simple, is responsible for establishing *trace relationships* (also called *trace links*, or just *traces*) between MYNK `Artifacts`. Figure 9.6 presents the abstract syntax for this module.



Figure 9.6: Abstract syntax for the Traceability module.

The most important concept in this module is the **Trace**, which is what effectively establishes a correspondence between two (or more) MYNK `Artifacts` in different models. To do so, a `Trace` contains a set of **Trace References**, which are *named references to MYNK Artifacts*.

A simple example of `Traces` with `Trace References` can be the following: if we have two CMS-ML and CMS-IL `Website Template Roles` – respectively R_{ML} and R_{IL} – that are meant to be semantically equivalent, then there will be a corresponding `Trace` element, which in turn will contain two `Trace References` – $TR_{R_{ML}}$ and $TR_{R_{IL}}$, each referencing

the corresponding `Role` in the CMS-ML or CMS-IL models – called `CMS-ML Role` and `CMS-IL Role` respectively. It should be noted that the aforementioned names are only used by the CMS-ML–CMS-IL MYNK synchronization, when needing to analyze the “opposite” element(s), and are not meant to be a pivotal piece of the example (these `Trace References` could be named anything, as long as the corresponding MYNK synchronization was also adjusted to recognize those new names).

The Traceability module’s concepts are based on related work (described in the next paragraphs) that deals with traceability between models.

The approach described in [ALC 08] defines a metamodel for a model transformation traceability engine, meant to establish trace links between model elements while a transformation is taking place. However, the ultimate objective of this effort is the optimization and refactoring of model transformations, by supporting the creation and visualization of traces *after* the transformation takes place, and obtaining information regarding the transformation’s operation. Thus, the approach is limited in that it does not use trace links as transformation *drivers*. Furthermore, it also presents the limitation of only establishing 1:1 (one-to-one) trace links, which might not be adequate for heterogeneous models (i.e., models whose abstract syntaxes present significant differences from each other), as it may be necessary for a single model element to be traceable to a *set* of other elements.

On the other hand, [DPFK 08] establishes a very simple metamodel for establishing trace links between elements of two different metamodels. Although this metamodel is very similar to our own, it does present the limitation of needing to be custom-tailored for the two metamodels, because each trace link is an instance of a class `XTraceLink` (`X` being the class of one of the elements being traced, and `XTraceLink` inheriting from a `TraceLink` class).

The approach in [SKR⁺ 08] defines another traceability metamodel, oriented toward supporting traceability between different products in the same product line. This metamodel is more complex than the aforementioned ones, but defines a set of concepts (namely regarding context and scope) that we do not consider to be applicable to MYNK’s purpose. Nevertheless, MYNK’s Traceability module shares many similarities with this metamodel.

Furthermore, the model synchronization approach described in [CS 07] also defines a `Trace` element, which connects two different model elements (`Artefacts`). However, this also carries the limitation of only supporting 1:1 trace links, which (as previously mentioned) might make this approach unsuitable for heterogeneous models.

Finally, the Kermeta metamodeling language also provides a traceability metamodel³ that is also similar to MYNK’s. However, Kermeta’s traceability metamodel refines the

³“Chapter 2. Presentation of the Traceability Metamodel”, *Traceability MDK for Kermeta*, <<http://www.kermeta.org/docs/fr.irisa.triskell.traceability.documentation/build/html.chunked/Traceability-MDK/ch02.html>> (accessed on March 30th, 2012)

concept of **Reference** with **FileReferences** and **ModelReferences** (to trace textual and graphical elements, respectively). In this regard, MYNK is only concerned about the model’s abstract syntax, and not with the manner in which it is represented (which can depend not only on the concrete syntax, but also on the designer’s own modeling style).

9.4.3 Changes

The Changes module is responsible for defining the concepts that are necessary to represent *changes to MYNK Artifacts* (regardless of the metalevel in which those **Artifacts** are located). These concepts, in turn, are a cornerstone of MYNK’s change propagation (and processing) mechanism. Figure 9.7 illustrates this module’s abstract syntax.

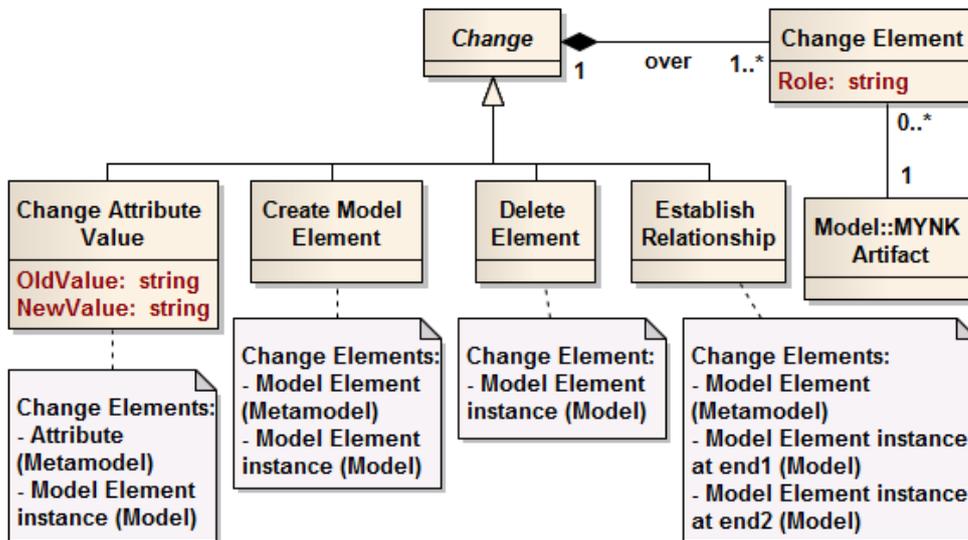


Figure 9.7: Abstract syntax for the Changes module.

Like the Traceability module, the Changes module is relatively simple, as it defines a small set of concepts. The most relevant concept is **Change**, which represents a *change in one or more model elements*. A **Change** contains a set of **Change Elements**, which are references to the model elements that are involved in the change being modeled.

It should be noted that the **Change** concept is abstract (otherwise, this concept would not be expressive enough for a computer to be able to answer the question “*What was the change?*”). More specifically, it is refined by the following concepts:

- **Change Attribute Value**, indicating a change in the *value* of a property (called **Attribute** in the ReMMM metamodel), for which it references (1) the metamodel element that represents the property itself, and (2) the instance of the model element that contains the property. Furthermore, it also contains not only the *new value*, but also the *old value*;

- **Create Model Element**, representing the creation of a new *instance* of a certain Model Element (e.g., CMS-IL’s **Entity**, or even ReMMM’s **Instantiation**);
- **Establish Relationship**, which references a certain metamodel element R (e.g., CMS-ML’s **Association**) that represents a relationship between Model Elements, and establishes a new *instance* of R between two other instances in the model; and
- **Delete Element**, which represents the *removal* of a certain instance (of either Model Element or Relationship) from the model.

MYNK defines this set of change concepts (and no additional ones) because these are the minimum necessary to specify any changes to a model, as long as it – and its metamodel – conforms to the ReMMM metamodel.

It should be noted that these **Change** concepts also provide the information that is necessary to determine whether a *conflict* is present. Specifically, a **conflict** occurs whenever a certain change is to be applied to a model, but that model contains information that is inconsistent with the intended change. An example of this could be a change to an attribute’s value (by using **Attribute Value Change**): (1) consider a change C that is to be applied to a certain attribute A ; (2) C is meant to change A ’s value from V_1 to V_2 (i.e., its **Old Value** and **New Value** properties assume the values V_1 and V_2 , respectively), and so it expects that A ’s value is initially V_1 ; (3) however, if A ’s value is *not* V_1 , then there is a conflict, and C cannot be applied to A . The topic of conflict resolution will be further explained in Section 9.6.

Furthermore, and unlike other MYNK modules, **Change** instances are *not* meant to be specified by the MYNK synchronization developer, but rather by a *modeling tool* for a MYNK-compatible language (e.g., a CMS-ML modeling tool). The MYNK synchronization process will then try to match these **Changes** with **Operation** specifications (explained in Subsection 9.4.4), in order to obtain a set of corresponding **Changes** to be applied to the corresponding model.

We have also found some related work regarding the modeling of changes. In particular, [GFD 05] and [DGF 05] define a model’s history as a *sequence of versions*, namely by defining a set of concepts ***History** (one for each metamodel element, such as **PackageHistory** for UML’s **Package**), which allows the modeling of the various versions of a model. However, the preservation of the model’s history as a set of snapshots notwithstanding, we consider that this approach suffers from two problems when considering model transformations: (1) the aforementioned issues – regarding the possible loss of data between transformations, although the lost data would still be accessible in older versions of the model – still apply; and (2) although it would be possible to obtain the changes that originated a new version, this would add an extra degree of complexity to the model transformation process.

9.4.4 MYNK

The MYNK module is responsible for using the concepts that were specified in the Changes and Traceability modules, and defining the concepts that support the MYNK language’s purpose (the definition of model synchronization operations). Figure 9.8 provides a simplified illustration of the abstract syntax for the MYNK module.

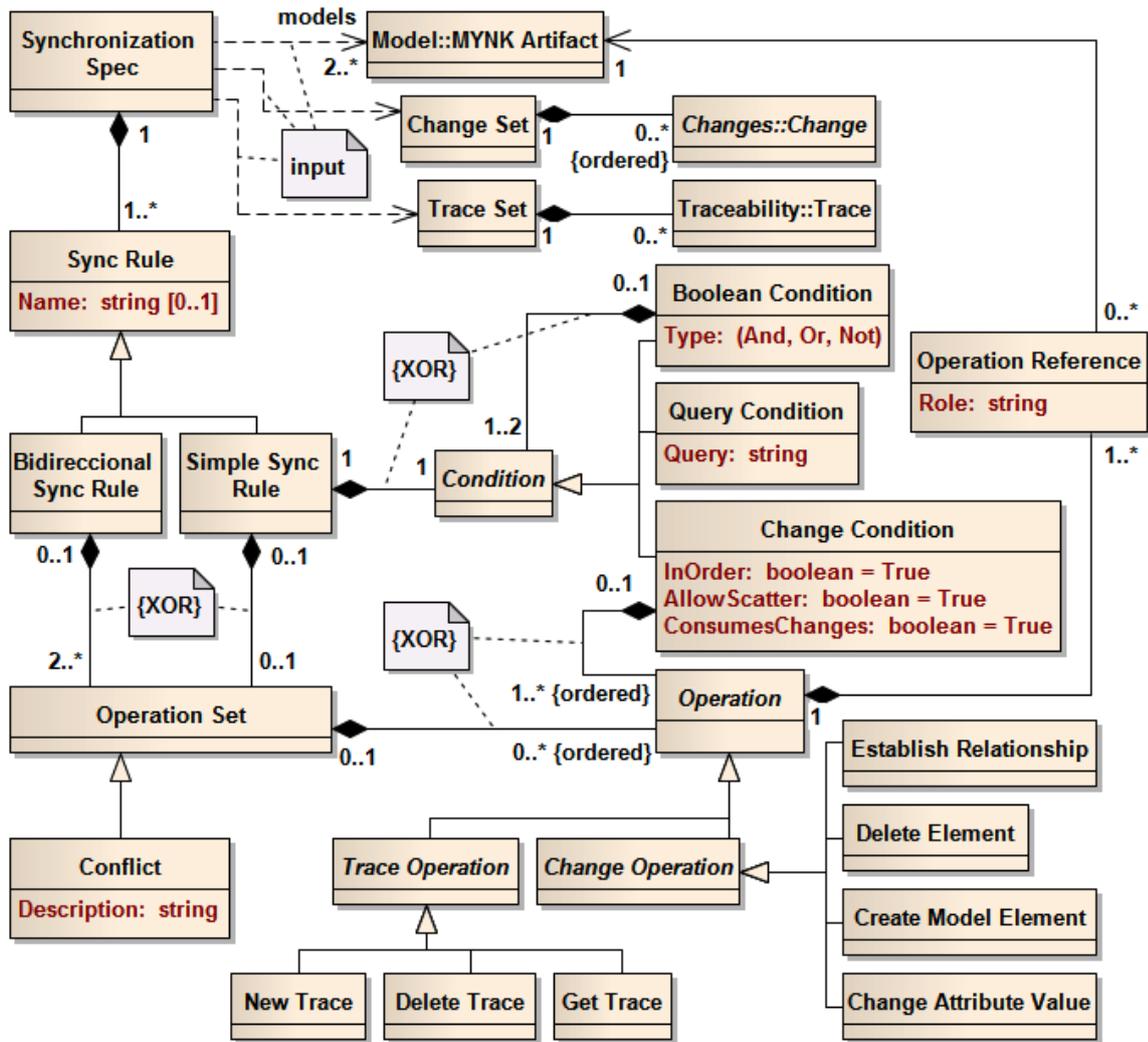


Figure 9.8: Abstract syntax for the MYNK module.

The underlying idea to these concepts is that MYNK will work by: (1) analyzing a set of **Changes**; (2) determining which synchronization rules contain condition operations that match with some of the provided **Changes**; (3) obtain *templates* for the set of **Changes** that should result from the provided **Changes**; and (4) based on a provided set of **Traces**, convert those templates into **Changes** to be applied to the models being synchronized. This workflow will be elaborated on further down this chapter.

We start this module’s description by the **Synchronization Spec** (or just **Sync Spec**, for brevity), which is a container for all the other concepts defined in this module. When defining a new model synchronization, the designer starts by specifying a **Synchronization Spec** element, within the context of which other MYNK elements will be defined.

A **Synchronization Spec** contains a set of **Sync Rules**, which (as the concept’s name suggests) are *rules* that guide the model synchronization process. More specifically, there are two types of **Sync Rules** – **Simple Sync Rules** and **Bidirectional Sync Rules** – which perform synchronization in different manners (these will be explained further down this section). A **Sync Rule** can also be characterized by a **Name**, which is an optional **string** that can be used to indicate the purpose of the rule.

In addition to the **Sync Spec** and **Sync Rule** concepts, this module also defines another fundamental concept, **Operation**. An **Operation** can be considered as “something to be done”. MYNK provides different kinds of **Operation**, which are derived from the concepts defined in the **Changes** and **Traceability** modules: **Trace Operations** and **Change Operations**. A **Trace Operation** consists simply of an instruction to fetch a **Trace** element, create a new **Trace** element, or remove an existing one. On the other hand, a **Change Operation** can be considered as a *template* or a *possible match for a Change*, which is why the available concrete **Change Operations** are defined in a way very similar to the concrete **Changes** in the **Changes** module.

Operations are grouped into **Operation Sets**, which can then be considered as expected sets of **Changes** (eventually combined with tracing instructions) to be matched when performing model synchronizations. **Operation Sets** are important because **Sync Rules** also contain such **Sets**. However, the kind of **Sync Rule** also determines how these **Operation Sets** are interpreted:

- A **Bidirectional Sync Rule** contains only a list of two (or more) **Operation Sets**. Each of those **Sets** can act as the *condition* for applying the **Sync Rule**, and the result of rule is provided by combining all of the remaining **Operation Sets**. It should be noted that there is a caveat regarding the **Set** that acts as the condition, as the **New Trace** and **Delete Trace Operations** are ignored when determining whether the **Operation Set** is a match for the provided **Changes**;
- On the other hand, a **Simple Sync Rule** contains only one **Operation Set**, which is the *result* of the rule. The rule’s condition is instead specified by the **Condition** concept, which can be of one of the following types: (1) a **Boolean Condition**, which contains other **Conditions** and combines them using boolean operators (**and**, **or**, or **not**); (2) a **Query Condition**, which contains a **Query** (a simple SQL-like string) that can be performed over input models and returns a value of **true** or **false**; or (3) a **Change Condition**, which contains an ordered set of **Operations**

that must be successfully matched to a set of **Changes**. The **New Trace** and **Delete Trace Operations** cannot be used in the context of **Change Conditions**.

In turn, the matching of **Operations** to **Changes** leads to the **Change Condition** concept having a set of properties that determine the manner in which such matching should occur: (1) the **In Exact Order** property specifies whether the corresponding **Operations** must be *matched in the exact same order in which they are specified*; and (2) the **Consumes Changes** property (used only in case of a successful match) determines whether the matched **Changes** will still be available for matching by other **Sync Rules**, or if they will be removed from the set of **Changes** so that no other rules can match **Operations** with them.

It is also possible to manually specify the existence of *conflicts* that can occur. This is captured by the **Conflict** concept, which is a particular kind of **Operation Set** that signals the need for a conflict resolution mechanism (explained further down this chapter). However, there are some restrictions to defining **Conflicts**, as they (1) cannot contain **Operations**, and (2) cannot be matched with any **Changes**. The rationale for these constraints is that it would make no sense for a **Conflict** to be used in such cases.

Finally, at runtime, a **Synchronization Spec** receives: (1) a **Change Set**, which is a container for an ordered set of **Change** elements; (2) a **Trace Set**, which is also a container, but now for a set of **Trace** elements; and (3) a set of (ReMMM-based) models. These elements are depicted in Figure 9.8 as being related to the **Synchronization Spec** concept by means of UML dependencies.

9.5 Concrete Syntax

As was previously mentioned, MYNK is a textual language, with a concrete syntax that is inspired in model querying and transformation languages, namely SQL and QVT (see Subsections 3.1.1 and 3.3.1, respectively). Listing 9.1 presents an example of the concrete syntax for the MYNK language, illustrating some relevant aspects that will be further explained in the following paragraphs. Nevertheless, interested readers may find additional details regarding MYNK's concrete syntax in the "MYNK User's Guide" [SS 11_c].

This example begins with the declaration of a new **Synchronization Spec**, indicated by the **Synchronization** keyword in line 1. This **Spec** is used to perform synchronization between two models, as indicated in line 2 by the variables `cmsml_model` and `cmsil_model` (which correspond to CMS-ML and CMS-IL models, respectively). It should be noted that indicating the name of the language for each declared model is optional, as it serves only as a reminder of what models should be provided as the **Spec**'s parameters.

The MYNK synchronization developer then specifies a set of **Sync Rules** within this **Sync Spec**, as illustrated in lines 4, 20, 29, and 35. A **Rule** is always declared with the

Listing 9.1: Concrete syntax for MYNK.

```

1 Synchronization
2   between (cmsml_model in "CMS-ML", cmsil_model in "CMS-IL")
3   consists of {
4     Rule "Sync Rule example: Create Binding to Create Reminder" [
5       When
6         CreateModelElement(model: cmsml_model, class: SELECT "Toolkit.Binding", element: theB)
7         ChangeAttributeValue(element: theB, attributeName: "EntityName", newValue: newEntityName)
8         ChangeAttributeValue(element: theB, attributeName: "BindingPath", newValue: newBindingPath)
9         ChangeAttributeValue(element: theB, attributeName: "IsDummy", newValue: newIsDummy)
10        EstablishRelationship(model: cmsml_model, type: Association, end1: theCMSMLWebElement,
11                               end2: theB)
12      do
13        CreateModelElement(model: cmsil_model, class: SELECT "Toolkit.Reminder", element:
14                              theReminder)
15        ChangeAttributeValue(element: theReminder, attributeName: "Description", newValue: (" Bind to
16          " + newEntityName + "." + newBindingPath)) WHEN newIsDummy IS False
17        ChangeAttributeValue(element: theReminder, attributeName: "Description", newValue: " Dummy
18          binding. Do nothing.") WHEN newIsDummy IS True
19        ChangeAttributeValue(element: theReminder, attributeName: "Completed", newValue: SELECT
20          True) WHEN newIsDummy IS True
21        NewTrace(theB as "CMS-ML Binding", theReminder as "CMS-IL Reminder")
22        GetTrace(from: theCMSMLWebElement, role: "CMS-IL WebElement", element:
23                  theCMSILWebElement)
24        assocBetweenReminderAndElement => SELECT WHERE type: Association, model: cmsil_model,
25          end1: SELECT "Toolkit.ToolkitElement", end2: SELECT "Toolkit.Reminder"
26        EstablishRelationship(model: cmsil_model, type: Association, class:
27          assocBetweenReminderAndElement, end1: theCMSILWebElement, end2: theReminder) ]
28    Rule "Bidirectional Sync Rule example: Creation of WebSite" [
29      CreateModelElement(model: cmsml_model, class: SELECT "WebSiteTemplate.WebSite", element:
30        theCMSMLWebSite)
31      ChangeAttributeValue(element: theCMSMLWebSite, attributeName: "Name", newValue:
32        newName)
33      NewTrace(theCMSMLWebSite as "CMS-ML WebSite", theCMSILWebSite as "CMS-IL WebSite")
34    <->
35      CreateModelElement(model: cmsil_model, class: SELECT "WebSiteTemplate.WebSite", element:
36        theCMSILWebSite)
37      ChangeAttributeValue(element: theCMSILWebSite, attributeName: "Name", newValue: newName)
38      ChangeAttributeValue(element: theCMSILWebSite, attributeName: "Description", newValue:
39        newDescription OR "")
40      NewTrace(theCMSMLWebSite as "CMS-ML WebSite", theCMSILWebSite as "CMS-IL WebSite") ]
41    Rule "Bidirectional Sync Rule example: WebSite name change" [
42      GetTrace(from: theCMSILWebSite, role: "CMS-ML WebSite", result: theCMSMLWebSite)
43      ChangeAttributeValue(model: cmsml_model, class: SELECT "WebSiteTemplate.WebSite",
44        element: theCMSMLWebSite, attributeName: "Name", oldValue: oldName, newValue:
45        newName)
46    <->
47      GetTrace(from: theCMSMLWebSite, role: "CMS-IL WebSite", result: theCMSILWebSite)
48      ChangeAttributeValue(model: cmsil_model, class: SELECT "WebSiteTemplate.WebSite", element:
49        theCMSILWebSite, attributeName: "Name", oldValue: oldName, newValue: newName) ]
50    Rule "Bidirectional Sync Rule example: WebSite description change" [
51    <->
52      ChangeAttributeValue(model: cmsil_model, class: SELECT "WebSiteTemplate.WebSite", element:
53        theCMSILWebSite, attributeName: "Description", oldValue: oldDescription, newValue:
54        newDescription) ] }

```

Rule keyword, regardless of its kind; the distinction between **Simple** and **Bidirectional Sync Rules** is made within the Rule's body, which is delimited by the [and] tokens. Once again, although the Rules presented in Listing 9.1 all have names, this is not mandatory, as such a name is only used to convey the corresponding Rule's purpose.

Lines 5–19 illustrate the body of a **Simple Sync Rule**. This kind of Rule is declared by using the **When** and **do** keywords, which indicate the Rule's condition and its resulting **Operation Set**, as depicted in lines 6–10 and lines 12–19, respectively (this particular **Sync Rule** will be explained further down this section).

On the other hand, lines 20–28 declare a **Bidirectional Sync Rule**. This Rule is characterized by containing a set of **Operation Sets** (separated by the <-> token, as shown in line 24). In particular, the Rule for this example specifies two **Operation Sets** – containing (1) a **Create Model Element**, a **Change Attribute Value**, and a **New Trace**, and (2) a **Create Model Element**, two **Change Attribute Values**, and a **New Trace**, respectively –, either of which may act as the Rule's condition; of course, the one that does *not* act as the Rule's condition will act as its result. Furthermore, it should be noted that **Trace Operations** (e.g., **New Trace**) are not considered when an **Operation Set** is used as a condition.

The **Operations** specified in Listing 9.1 are all defined in the same manner: the **Operation's** name, followed by a set of arguments. An example can be found in the following **Operation** representation (corresponding to line 6 of Listing 9.1):

```
| CreateModelElement(model: cmsml_model, class: "Toolkit.Binding", element: theB)
```

This representation reflects a **Create Model Element Operation**, which indicates the creation of a new instance of the **Model Element** concept. This **Operation** has three parameters that have the following roles (respectively): (1) the **model** in which the **Operation** should find a match; (2) the **Model Element** that is the **class** for the new **Model Element**; and (3) the new instance itself. The explicit definition of these roles (represented in Figure 9.8 by the **Operation Reference** concept) also means that the order in which an **Operation's** parameters are not specified is not important, which is particularly helpful to avoid errors when defining new **Sync Rules** (or when reading existing ones). In the case of this particular example, and because it is specified within a **Change Condition**, this **Operation** should be matched to a **Create Model Element Change** with the following properties: (1) the new **Model Element** instance should have been created within the **cmsml_model** model (provided as a parameter to the **Synchronization Spec**, see line 2); and (2) the new instance should have a **CMS-ML Toolkit Binding** as its ontological metaclass (i.e., it should be related to **Binding** by ReMMM's **Instantiation** relationship).

This `Operation` example also brings up some important issues, namely: (1) the possible values that can be provided as parameters, and (2) the binding of variables. Although these issues are closely related to each other, they do present some caveats that MYNK synchronization developers should be aware of.

The first issue regards the fact that an `Operation` parameter can receive one of the following: (1) a *variable*; (2) a SQL-like *query* that returns a value; (3) a *string*; or (4) a concatenation of the above. The way in which each of these is interpreted depends on the kind of value that is expected for the parameter. For example, a parameter that should receive an `Attribute` value can receive either: (1) a variable, which can be either bound or unbound (explained next), depending on whether its corresponding value – if bound – should be considered when determining matches; or (2) a string. On the other hand, a parameter that is meant to reference a `Model Element` instance can receive a variable, or a query that returns a specific `Model Element`. It is also possible to specify that a certain `Operation` should only be performed when a certain condition is met, by using the `WHEN` keyword (as shown in lines 13–15).

Queries are specified using the (SQL-inspired) `SELECT` keyword. In the same manner as SQL queries, a MYNK query consists of a *filter* over the set of `Model Elements` that the MYNK synchronization is currently handling. Furthermore, it can also be used to indicate that what follows the `SELECT` keyword must not be used as-is, but should instead be *evaluated*. Listing 9.1 does provide some examples of this keyword’s usage, of which we highlight the following:

- The parameter `class` in line 6 receives a string, “Toolkit.Binding”. This string is an identifier for a MYNK `Artifact` – which in turn references a specific `Model Element` instance⁴ – and so it must be evaluated, in order to return the corresponding `Model Element` (instead of the string itself);
- One of the arguments in line 15, `True`, is neither a string nor a variable, and thus must be evaluated, in order to yield the corresponding boolean value of `True`; and
- Line 18 contains a query that will return one of the elements which defines the CMS-IL modeling language itself (more details regarding this query will be provided further down this section).

Another useful MYNK construct is the *macro*, an example of which can be found in line 18. This construct (characterized by the `=>` token in its concrete syntax) allows a synchronization developer to specify a query and assign it a *name*. These queries, however, are not immediately evaluated; instead, the macro is *expanded* (and its query is evaluated) only when its name is used (e.g., the macro `assocBetweenReminderAndElement`, defined in

⁴In this particular example, we have chosen a readable identifier for each MYNK `Artifact` corresponding to CMS-ML and CMS-IL `Model Elements`, solely for clarity purposes. However, as was previously mentioned, language designers are free to adopt any naming scheme for their own elements.

line 18, is only expanded and evaluated when its name is used in an **Operation**, which happens in line 19). This is inspired in the notion of macro that is present in programming languages such as C [KR 88], which typically consider a macro to be a snippet of source code that is defined once and, *at compile-time*, is included in various points of a program.

The second issue concerns the usage of variables (and their possible bindings) as values for **Operation** parameters. MYNK considers that variables (i.e., identifiers that are not strings nor queries) can be in one of two states: *bound* or *unbound*. Variables are bound when they have been already assigned a value (i.e., they have been *bound* to that value, which can be a string or a **Model Element** instance), and are unbound otherwise. When a **bound** variable is used in an **Operation**, it establishes a constraint that *only Changes that match that variable's current value are to be considered*. On the other hand, an **unbound** variable is used to indicate that *the variable should assume the values found in Changes that also have values matching those in the Operation's bound variables* (and thus become a bound variable). Of course, it is up to the *synchronization engine* (the program that performs MYNK synchronization) to determine whether there are any **Changes** that are adequate matches for an **Operation Set**. Furthermore, when determining matches between **Operations** and **Changes**, each **Sync Rule** establishes a *scope*, within which all bindings are performed; when a different **Rule** is analyzed, a different scope is established, thus losing all bindings – if any – that were previously made during another analysis of a **Rule**. This ensures that the order in which variables are bound (in the context of different **Rules**) does not affect the synchronization process.

MYNK's variable-binding mechanism was inspired on the notion of *backward chaining* [RN 09], an inference method that can be characterized (from a simplistic point of view) as *working from goal to rules*. The operation of inference methods is based on a *knowledge base* and a set of *goals*: (1) the **knowledge base** (also called *rule base*) consists of a set of *statements*, typically specified as logical rules such as **if condition then result**, that constitute the *knowledge* which the system will work with; and (2) the **goals** are hypothetical statements that the user asks or intends to see proven. Inference engines that employ backward chaining start from the provided goals, and use them – along with the knowledge base's rules – to infer new information, until the goals are proven (if possible)⁵. The similarity to MYNK's variable-binding comes from the fact that **Changes** and **Sync Rules** can be considered as analogous to goals and knowledge base rules, respectively, as MYNK's synchronization engine should operate by receiving a **Change Set** and, while processing those **Changes** (in the order in which they are provided), trying to find **Sync Rules** that are matches for those **Changes**.

⁵A detailed explanation regarding the operation of inference engines is out of the scope of this dissertation, and so it will not be provided here.

Referring to the **Simple Sync Rule** example presented in Listing 9.2 (which is taken from Listing 9.1), the synchronization engine should interpret it in the following manner:

Listing 9.2: Example of Simple Sync Rule (excerpt from Listing 9.1).

```

1 Rule "Sync Rule example: Create Binding to Create Reminder" [
2   When
3     CreateModelElement(model: cmsml_model, class: SELECT "Toolkit.Binding", element: theB)
4     ChangeAttributeValue(element: theB, attributeName: "EntityName", newValue: newEntityName)
5     ChangeAttributeValue(element: theB, attributeName: "BindingPath", newValue: newBindingPath)
6     ChangeAttributeValue(element: theB, attributeName: "IsDummy", newValue: newIsDummy)
7     EstablishRelationship(model: cmsml_model, type: Association, end1: theCMSMLWebElement, end2:
      theB)
8   do
9     CreateModelElement(model: cmsil_model, class: SELECT "Toolkit.Reminder", element: theReminder)
10    ChangeAttributeValue(element: theReminder, attributeName: "Description", newValue: ("Bind to " +
      newEntityName + "." + newBindingPath)) WHEN newIsDummy IS False
11    ChangeAttributeValue(element: theReminder, attributeName: "Description", newValue: "Dummy
      binding. Do nothing.") WHEN newIsDummy IS True
12    ChangeAttributeValue(element: theReminder, attributeName: "Completed", newValue: SELECT True)
      WHEN newIsDummy IS True
13    NewTrace(theB as "CMS-ML Binding", theReminder as "CMS-IL Reminder")
14    GetTrace(from: theCMSMLWebElement, role: "CMS-IL WebElement", result: theCMSILWebElement)
15    assocBetweenReminderAndElement => SELECT WHERE type: Association, model: cmsil_model,
      end1: SELECT "Toolkit.ToolkitElement", end2: SELECT "Toolkit.Reminder"
16    EstablishRelationship(model: cmsil_model, type: Association, class: assocBetweenReminderAndElement,
      end1: theCMSILWebElement, end2: theReminder) ]

```

- The **Create Model Element Operation** of line 3, being in the Rule's condition, is interpreted as needing to match a **Create Model Element Change** which (1) takes place in the CMS-ML model (provided in the `cmsml_model`), (2) uses the **Toolkit Binding** concept as the metaclass, and (3) binds the newly created **Model Element** instance to the `theB` variable;
- The **Change Attribute Value Operations** in lines 4–6 are all meant to be matched to **Change Attribute Value Changes** that (1) take place on the newly created **Model Element** instance (already bound to the `theB` variable), and (2) are performed over the `EntityName`, `BindingPath`, and `IsDummy` properties. Furthermore, the new values for these properties are bound to the `newEntityName`, `newBindingPath`, and `newIsDummy` variables, respectively (the old values are ignored, because the **Model Element** instance `theB` was just created);
- The **Establish Relationship Operation** represented in line 7 should be matched to an **Establish Relationship Change** that (1) takes place in the `cmsml_model` model, (2) creates an **Association**, and (3) makes that **Association** take place between the `theB` instance and another instance, to be bound to the variable `theCMSMLWebElement` (a CMS-ML **Binding** is always a part of a **WebElement**, as was previously depicted in Figure 7.20). It is not necessary to specify that the new **Association** instance

- must have a specific `Model Element` as its ontological metaclass, because there is only one `Association` between CMS-ML's `Binding` and `WebElement`;
- The `Rule`'s result starts with a `Create Model Element Operation`, depicted in line 9, that (1) is performed in the `cmsil_model` model, (2) creates an instance with the `Toolkit Reminder` concept as its metaclass, and (3) binds this new instance to the `theReminder` variable;
 - Afterward, three `Change Attribute Value Operations` are specified, meant to change the values of the `Description` and `Completed` properties of the `theReminder Model Element`. These changes are performed in the following manner:
 - If the `newIsDummy` variable has the value `False`, only the first `Change Attribute Value` will actually be performed (as indicated by the query `WHEN newIsDummy IS False` that is appended to the `Operation`). If it is performed, it changes the value of `Description` to a string that results from concatenating the values of `newEntityName` and `newBindingPath` to other strings. The `Completed` property is not assigned because it has the default value of `False` (see Figure 8.22b);
 - Otherwise, if `newIsDummy` is `True`, only the second and third `Change Attribute Values` will be performed (according to their respective queries), in which (1) `Description`'s value is set to a string, and (2) the value of `Completed` is set to `True` (because the original CMS-ML `Binding` is meant to be ignored);
 - The `Rule`'s result continues with a `New Trace Operation`, in line 13, which establishes a new `Trace` between the values of the `theB` (bound in line 3) and `theReminder` variables. The role played by these elements (as per the `Trace Reference` concept, see Figure 9.6) is also indicated, after the `as` keyword; in this particular case, `theB` and `theReminder` have the roles `CMS-ML Binding` and `CMS-IL Reminder`, respectively. These trace links can then be used in `Get Trace Operations`;
 - After creating the `Reminder`, the only remaining step is its association to the CMS-IL `WebElement` that corresponds to `theCMSMLWebElement` (which was bound in line 7). Line 14 initiates this process, by defining a `Get Trace Operation` which (1) takes the `theCMSMLWebElement` variable, (2) finds the set of all `Traces` in which the variable's `Model Element` participates (in a manner that is transparent to the synchronization developer), and (3) obtains the corresponding `Model Element` that plays the role `CMS-IL WebElement`. The obtained `Model Element` instance is then bound to the variable `theCMSILWebElement`;
 - Line 15 defines a macro, `assocBetweenReminderAndElement`. The expansion of this macro will result in a query that returns all `Model Elements` which (1) are included in the `cmsil_model` model, (2) are instances of `Association`, and (3) take place between the CMS-IL `Toolkit's Reminder` and `Toolkit Element` (`Toolkit Reminders`

- are associated to `Toolkit Element`, from which all `Toolkit` concepts inherit, as was previously shown in Figure 8.22b). In practice, this query will return the `Model Element` corresponding to the association between `Reminder` and `Toolkit Element`;
- The result is finalized by line 16, with an `Establish Relationship Operation` that (1) is performed in the `cmsil_model` model, (2) creates an `Association`, (3) makes that new `Association` assume the result of `assocBetweenReminderAndElement` – a macro that gets expanded into a query – as its metaclass, and (4) uses that same `Association` to link the `Model Elements` represented by `theReminder` and `theCMSILWebElement`.

Thus, this `Sync Rule` establishes that the creation of a CMS-ML `Binding` corresponds to the creation of a CMS-IL `Reminder`. This `Reminder` has a `Description` that varies according to whether the `Binding` is a dummy (more details on `Bindings` are available on the “CMS-ML User’s Guide” [SS 10.c]). Furthermore, if the `Binding` is a dummy, then the `Reminder`’s `Completed` property is also set to `True`. Figure 9.9 provides a (simplified) MYNK-oriented illustration of how the CMS-ML and CMS-IL models should look like after this synchronization takes place.

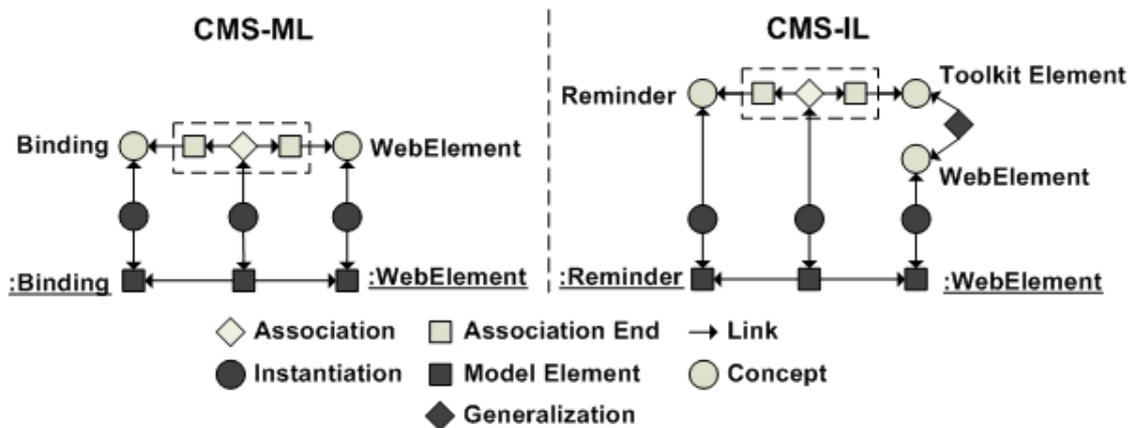


Figure 9.9: CMS-ML and CMS-IL models after applying a `Sync Rule`.

Another noteworthy topic regarding the example in Listing 9.1 is that `Bidirectional Sync Rules` are particularly useful to avoid defining *multiple* `Simple Sync Rules`, in which each rule has a condition with a set of `Operations` in a model and results in `Operations` to another model, but those sets of `Operations` are *semantically equivalent among themselves*. In other words, `Bidirectional Sync Rules` simplify cases in which a rule *A*’s condition and result are equivalent to another rule *B*’s result and condition (respectively), by collapsing such semantically equivalent rules (*A* and *B*) into a single one.

Finally, it should be noted that different MYNK-supporting tools (i.e., tools that *record* the various `Changes` that are made to a model) may provide `Changes` in a different order than the one in which a certain `Sync Rule` (to which those `Changes` are meant to

be matched) defines its `Operations` (e.g., instead of having `Change Attribute Value Operations` for `Name` and then `Description` – as in lines 26–27 of Listing 9.1 –, a tool could provide a `Change Set` in which `Description` is changed *before* `Name`). Nevertheless, as was previously mentioned, it is possible for a `Change Condition` to indicate that, rather than matching its `Operations` in the exact order in which they were specified, their order should not be considered when determining matches. Although, from a theoretical perspective, it would be possible for this to lead to matches that would make no sense (such as changing the value of an `Attribute` on a `Model Element` that hasn't been created yet), in practice the aforementioned variable-binding mechanism would ensure that only reasonable matches were produced (assuming, of course, that the `Changes` in the provided `Change Set` are presented in a correct and consistent order).

9.6 Conflict Resolution

As was previously mentioned, MYNK does not exclude the possibility of *conflicts* occurring during model synchronization operations. In fact, such conflicts are to be expected in practice, because stakeholders will try to modify their model to reflect their own view and interpretation of the desired web application. Thus, if two stakeholders have different (and conflicting) views of that system, then their respective changes to the same model are likely to conflict with each other.

Considering the concepts defined by the ReMMM metamodel and the various types of `Changes` that MYNK contemplates, it is possible to anticipate the kinds of conflict that may occur:

- A `Create Model Element Operation` can lead to a conflict when the ID of the MYNK `Artifact` is specified as a string, query result, or already-bound variable (i.e., it is not an unbound variable), *and* another MYNK `Artifact` with that same identifier already exists;
- A `Change Attribute Value Operation` can originate conflicts when the attribute's old value is specified (and it is not an unbound variable), *and* that same attribute has a different value in the model;
- An `Establish Relationship Operation` can originate conflicts when one of the following conditions occurs:
 - The new element is a `Generalization`, and it would establish a circular loop of `Generalization` elements (i.e., the model would state that there were `Model Elements` inheriting from themselves, because `Generalization` is transitive);
 - The new element is an `Instantiation`, and its creation would establish a circular loop (similarly to the aforementioned `Generalization`);

- The ID for the new `Model Element` is specified (and it is not an unbound variable), *and* another MYNK `Artifact` with that identifier is already present in the model (as in the `Create Model Element Operation`); or
- The new element is an `Association` (between two `Model Elements`), and this new element violates at least one of the constraints specified in its metaclass (e.g., considering the `Association` between CMS-ML’s `WebElement` and `Binding` – see Figure 7.20 –, it should not be possible for an instance of `WebElement` to be associated to two or more instances of `Binding`);
- A `Delete Element Operation` can lead to a conflict when the `Model Element` to delete does not exist.

Language-specific semantic constraints – such as forcing `Model Elements` to have unique names – are not yet supported with the current revision of MYNK (although this would be a desirable feature, and so it is considered as future work).

Another possible source of conflicts, for *all* of these `Operations`, is when a `Model Element` is referenced (e.g., as a metaclass for a new element) but it does not exist. An example of a case in which this can happen is with the result of a `Get Trace Operation`: if there is no `Trace` element with the intended characteristics (e.g., because the `Trace Set` provided to the `Sync Spec` was incomplete), then the resulting element will be `nil`, which in turn can negatively affect future `Operations`.

There is a large set of situations that can lead to the conflicts presented above. Thus, and because it would not be practical to require that a synchronization engine consider all those situations and try to automatically address them, conflicts should be resolved interactively by the user, in a manner similar to the *merging of changes* in revision control systems (RCS, previously analyzed in Chapter 3).

We consider that, from a practical perspective, the possible conflict resolution approaches can be categorized according to two orthogonal perspectives, (1) the *moment at which the resolution is performed* and (2) the *modeling language used*.

The *moment at which the conflict resolution is performed* concerns the stage in the synchronization process at which the resolution actually takes place. We consider that there are two possible moments to address conflict resolution, which in turn lead to the corresponding approaches: (1) the *corrective* approach, and (2) the *preemptive* approach.

The **corrective approach** consists of manipulating the models being synchronized, so that they accurately reflect their expected states *after* the conflicting `Operation` (`Op`) would take place; in other words, the user is expected to bypass `Op`’s execution and instead manually update the model(s) to what would be their expected state *after* `Op` took place. This is similar to the manner in which conflicts are resolved in an RCS: when a user updates her working copy, and a conflict is detected in a text file *F*, it is up to the user to

edit F so that it includes not only its previous contents, but also the changes that have been committed to the repository (i.e., so that its changes are *merged*). For this operation, the user typically has access to (1) F 's local copy as it was *before* the update, (2) F 's current contents in the remote repository, and (3) F in its current conflicted status, with the conflicting text segments delimited by a set of markers. However, while in an RCS the user does this merging *after* the file synchronization operation takes place (and F is in a conflicted state), in MYNK this correction takes place *instead* of the **Operation**; this is in order to avoid potential problems that could arise from the **Operation** performing unexpected changes to the model.

On the other hand, the **preemptive approach** is characterized by the fact that the conflict resolution takes place *before* the conflict actually occurs (hence the name). In other words, this conflict resolution is performed by: (1) pausing the synchronization process just before **Op** takes place; (2) correcting the models so that they are in a state that is compatible with what **Op** would expect; and (3) resuming the synchronization process (and running **Op**).

Both the corrective and preemptive approaches require that users be able to manipulate the synchronization's **Trace Set**, so that they can update the **Trace** elements that connect those models (a task which is typically performed in a **Sync Rule** by using the **Get Trace**, **New Trace**, and **Delete Trace Operations**). Furthermore, they also require that users have some knowledge of what the synchronization **Operation** will actually do, and what conditions it expects in order to be correctly performed; however, this caveat can be mitigated by annotating **Operations** with adequate comments and/or providing proper documentation.

Finally, regarding the *modeling language used*, the person who is performing the synchronization can choose to resolve the conflict by changing the models employing either (1) the model's language itself (e.g., CMS-ML or CMS-IL), or (2) the ReMMM modeling language. This choice should be made according to the person's comfort and proficiency with ReMMM vs. the model's language, which is ultimately a subjective issue that depends on factors such as a person's experience with modeling languages and metamodeling. Because of the subjectivity of this topic, we will not discuss it further in this dissertation.

9.7 Characteristics of MYNK-Compatible Languages

In order for a **Synchronization Spec** to consider – and be used with – a certain language (e.g., CMS-ML, CMS-IL), that language must be *compatible* with MYNK. This compatibility consists of exhibiting a specific set of characteristics that match the assumptions made during the definition of the MYNK language, so that the synchronization

engine can “hook into” the language’s elements, analyze them, and (if necessary) change them. These characteristics can be brought together into a small set of requirements⁶ for MYNK-compatible languages, which is explained in the following paragraphs.

The language can be defined with the ReMMM metamodel. For a language to be usable in a MYNK synchronization, the elements of its metamodel and model(s) *must* be classifiable according to a small set of basic elements (such as **Concept**, **Association**, or **Instantiation**).

Although it would be possible for a language L , based on a metamodel other than ReMMM, to provide a *superset* of these elements (in order for MYNK to be able to use that language), it should be noted that ReMMM is – in our perspective – a minimalist language defined with the sole purpose of supporting this classification scheme in a manner that is as simple as possible. In turn, this would make it likely that L (and L -based models) also conforms to the ReMMM metamodel, which would fulfill this requirement.

Any change to a model should be specified as a Change (or set of Changes). As was previously mentioned, in order for a **Synchronization Spec** to perform its synchronization tasks, it must receive (1) a set of ReMMM-based models, and (2) a set of **Changes**, which effectively give the synchronization engine the *history* of those models. Furthermore, the **Changes** and MYNK modules – presented in Subsections 9.4.3 and 9.4.4, respectively – are mainly based on the concept of **Change** and **Operation** (and the latter is meant to be matched with **Changes**).

Thus, it is necessary that any and all possible changes to a model be specifiable with the **Change** concept (more specifically, with the different specializations of **Change** that are defined in the **Changes** module), in order for MYNK to correctly analyze the models’ histories and further evolve those models (so that, when the synchronization process ends, they are semantically equivalent to each other).

It should be noted that this requirement is closely related to the previous one, regarding ReMMM. Considering that MYNK’s **Changes** module was inspired on ReMMM and the various kinds of changes that can occur over a ReMMM-based model, if a language is defined using ReMMM as its metamodel, then it automatically supports MYNK’s **Changes**.

The language supports a mechanism for translating MYNK Artifacts to specific model elements. As was previously mentioned in this chapter, each of the language’s **Model Elements** must be *unambiguously* accessible via a **MYNK Artifact** (and its corresponding **ID**). This requirement is suggested in Figure 9.5, which indicates that any **Model**

⁶The term “requirement” is used here in an informal manner, and does not assume the formal meaning from other software engineering disciplines such as Requirements Engineering.

Element should exhibit the characteristics that would make it be “classifiable” as a **MYNK Artifact**. However, the MYNK language does not specify the manner in which those characteristics should be supported by the language.

An easy way to address this requirement is for all of the language’s concepts to define an **Attribute Identifier**; this attribute, in turn, would have the same value as the corresponding **MYNK Artifact**’s ID. If such an attribute was available, then the translation between **MYNK Artifacts** and model elements could be performed in a straightforward manner. In fact, this is the strategy followed during the validation of this research work, which is described in Chapter 10 (support for other kinds of translation approaches is not currently addressed by MYNK).

Thus, any languages that fulfill these requirements can be used with the MYNK synchronization engine. Although, from a theoretical perspective, it would be possible to use other languages (which do not address these requirements) with MYNK – by using mechanisms such as synchronization engine extensions –, we do not consider such possibilities in either this version of MYNK or this dissertation.

Summary

In this chapter, we have presented the MYNK model synchronization language. This textual language is the cornerstone of the CMS-based web application development approach that was presented in Chapter 6, because it enables the synchronization (either in a real-time or delayed manner) of multiple models that can be changed simultaneously by different stakeholders.

In the next chapter, we present and discuss our validation of this research work, namely of the components presented in these chapters (CMS-ML, CMS-IL, and MYNK). This validation was conducted by defining a small set of case studies, which all together should present a reasonable degree of complexity, and allow the assessment of whether this approach is adequate for the modeling and development of CMS-based web applications.

Chapter 10

Validation

In theory, there is no difference between theory and practice. But, in practice, there is.

JAN L. A. VAN DE SNEPSCHEUT

The previous chapters have presented the proposed approach for CMS-based web application development – as well as its supporting languages – with a considerable level of detail. The CMS-ML and CMS-IL modeling languages are meant to support different kinds of stakeholder (the Business Designer and the System Designer, respectively), while the MYNK model synchronization language is used to support the automatic synchronization of changes between CMS-ML and CMS-IL models.

This chapter presents the validation efforts that were performed to determine the viability of these research results. This description consists of: (1) a feature-based comparison (similar to the one in Chapter 4) between CMS-ML, CMS-IL, and the other State of the Art web modeling languages; (2) the validation of CMS-ML and CMS-IL through a set of case studies; (3) the validation of MYNK in performing model synchronization between CMS-ML and CMS-IL models; and (4) an assessment – based on the research questions identified in Chapter 1 – of whether our results do accomplish their intended objective.

It should be noted that this chapter does not contemplate the validation of the *ReMMM* metamodel or of the *guidelines for language specification* (described in Chapter 6). This is because we consider that the definition of the CMS-ML and CMS-IL languages is itself a validation of these two contributions. Considering that this definition process is already explained in Chapters 7–8, its presentation in the current chapter would not bring any added-value to this dissertation.

Furthermore, we do not present the validation of the proposed approach for CMS-based web application development, because we were unable to perform an *adequate* validation of it

(i.e., with the participation of both technical and non-technical stakeholders). Nevertheless, we did perform a preliminary validation of this approach (during the validation of its components: CMS-ML, CMS-IL, and MYNK), in order to determine whether the approach could be used in practice.

10.1 Web Modeling Language Comparison

An important factor in determining the relevance of CMS-ML and CMS-IL as web modeling languages is the set of features that these languages offer, more specifically the variety and relevance of those features. Thus, one of the validation efforts performed was a feature-based comparison between CMS-ML, CMS-IL, and the web modeling languages that have been analyzed in Chapter 4. The results of this comparison are depicted in Table 10.1, and explained in the following paragraphs.

A – Domain modeling. Both CMS-ML and CMS-IL address domain modeling, although the concepts provided by CMS-IL have a higher degree of expressiveness than CMS-ML’s (e.g., CMS-ML does not support CMS-IL’s `Method` concept). Moreover, it should be noted that the domain modeling concepts provided by these two languages are independent from: (1) persistence, as they do not convey any database- or storage-specific details; and (2) UI, as their UI modeling does not require that every interface element be bound to a domain element.

B – Business Logic modeling. Although CMS-ML and CMS-IL both support business logic modeling, they do so differently.

CMS-ML addresses this aspect in Toolkit models, as the Tasks view allows Toolkit Designers to specify sequences of `Actions` to be performed when using the web application; these `Actions`, in turn, are the driver for the UI- and domain-oriented operations that can be specified in other views (e.g., the Interaction Triggers view). Furthermore, the Side Effects view allows the Toolkit Designer to specify domain manipulation using the basic CRUD (Create, Read, Update, and Delete) functions, although this specification is done in a very rudimentary manner.

On the other hand, CMS-IL Toolkit models are not oriented toward supporting such sequences of actions, but rather toward how the CMS system should behave when certain events occur (e.g., a click on a certain `Button`). CMS-IL also supports domain manipulation patterns: (1) typical patterns are provided by the `new`, `delete`, and `save` keywords [SS 11_b], which provide CRUD functionality; and (2) custom patterns can be specified via `Lambdas` and `Methods` in domain `Entities`. This strategy allows CMS-IL

Table 10.1: Comparison between CMS-ML, CMS-IL, and other web modeling languages.

	Web-ML	UWE	XIS2	Out-Syst.	Sket.	CMS-ML	CMS-IL
A. Domain modeling	✓	✓	✓	✓	✗	✓	✓
<i>Independent from persistence</i>	✓	✓	✓	✗	—	✓	✓
<i>Independent from UI</i>	✓	✗	✓	✓	—	✓	✓
B. Business Logic modeling	✓	✓	✓	✓	✓	✓	✓
<i>Domain manipulation using patterns</i>	✓	✗	✓	✓	✗	✓	✓
<i>Custom patterns</i>	✓	—	✓	✗	—	✗	✓
<i>Low-level specifications</i>	✓	✓	✗	✓	✓	✗	✓
<i>Domain query</i>	✓	✓	—	✓	✓	—	✓
<i>Domain manipulation</i>	✓	✗	—	✓	✗	—	✓
<i>Process specification</i>	✗	✓	—	✓	✗	—	✗
C. Navigation Flow modeling	✓	✓	✓	✓	✓	✗	✗
D. User Interface modeling	✓	✓	✓	✓	✓	✓	✓
<i>Access control specification</i>	✓	✗	✓	✓	✗	✓	✓
<i>Custom interface elements</i>	✓	✗	✗	✓	✓	✓	✓
<i>Interaction patterns</i>	✓	✗	✓	✓	✓	✓	✓
<i>Custom interaction patterns</i>	✗	—	✗	✗	✓	✗	✗
<i>UI elements bound to domain elements</i>	✓	✓	✓	✓	✓	✓	✓
<i>Bindings are customizable</i>	✗	✗	✓	✓	✗	✓	✓
E. Model-to-model transformations	✗	✓	✓	✗	✗		✓
F. Generated application is complete	✗	✗	✗	✓	✗	✗	✓
G. Independent from deployment environment	✓	✓	✓	✗	✓	✓	✓

to be as expressive as a typical programming language, in a manner similar to how the OutSystems Agile Platform addresses business logic modeling.

C – Navigation Flow modeling. Contrarily to all other web modeling languages, CMS-ML and CMS-IL do not provide support for the modeling of navigation flows between web pages (which are represented as `Dynamic WebPages` in both languages). This is because most CMS systems do not often constrain the manner in which users can

navigate through the website. Instead, in each web page, users are presented with a menu (or similar construct) that depicts the website's structure – or parts of it – as a set of hyperlinks. That menu, in turn, is automatically generated by the CMS system and, depending on the CMS system, the CMS Administrator may be responsible only for specifying *when* a certain hyperlink should appear (e.g., some links should only appear in specific web pages, while other links should only be visible to authenticated users).

Regarding the navigation flow between **Toolkit Support WebPages**, this is specified in an indirect manner: (1) in CMS-ML, this is represented by the Tasks view's **Action Transitions**, taking place between **Actions** that may be supported by different **Support WebPages**; on the other hand, (2) in CMS-IL, this is supported by the Code view, namely by the **CMS** class and its **goToPage** method (see Listing 8.14 in Chapter 8 for an example).

D – User Interface modeling. UI modeling is supported by both CMS-ML and CMS-IL, albeit CMS-IL does not support this in a graphical manner. Regarding access control, it can be specified in both the **WebSite Template** and **Toolkit** metalevels. We also consider that both languages support the definition of custom interface elements, because **WebComponents** (defined in a **Toolkit**) can be considered as such custom elements to be used in **WebSite Templates**; however, it is not possible to define new kinds of **WebElements** – e.g., a new **WebElement** that consists of a **Text** next to a **Text Input Box** –, as this would require the existence of an extra metalevel (to provide basic HTML elements).

Interaction patterns are also supported, as both languages consider that certain events can occur on each **WebElement** (e.g., the **Click** event can take place in a **Button**).

Finally, CMS-ML and CMS-IL both support the binding of domain elements to UI elements, so that the modeled web application can display and/or manipulate certain domain element instances. However, and unlike some of the State of the Art languages analyzed in Chapter 4, these languages address the customization of such bindings by allowing **Toolkit Designers and Developers** to specify **Binding Paths**, which enables scenarios such as having a **Text WebElement** displaying the name of a certain domain instance while another **Text** displays the name of a *related* domain instance.

E – Model-to-model transformations. Although CMS-ML and CMS-IL are not development approaches *per se*, they are meant to support the proposed CMS-based web application development approach that was presented in Chapter 6. Furthermore, the cornerstone of this approach is the use of a model synchronization mechanism, which can be regarded as a generalization of model-to-model transformations. Thus, we consider that support for model-to-model transformations is addressed by the proposed approach (albeit not in the traditional manner provided by the other analyzed approaches).

F – Generated application is complete. This is another aspect in which the differences between CMS-ML and CMS-IL are most noticeable.

CMS-ML is a very high-level modeling language, and not expressive enough to model the details of a web application that addresses all of the requirements desired by its stakeholders. An example of that lack of expressiveness is that CMS-ML does not support the definition of algorithms, as was mentioned in Chapter 7. In turn, this means that a CMS-ML model does not have enough information – on its own – to enable the generation of a complete web application (i.e., the intervention of developers would still be required).

On the other hand, CMS-IL is a low-level language that provides all the concepts that are necessary to develop CMS-based web applications with a low or medium degree of complexity. The reason why we say this is that CMS-IL’s concepts – such as `Role`, `Event Handler`, or `Variability Point` – are in fact provided by most CMS systems, and are used by CMS-based web applications (e.g., plugins for Drupal or Joomla) to provide additional functionality. Of course, concepts that are not supported by a specific CMS system should be *emulated* by the CMS Model Interpreter component (see Chapter 6). Thus, it can be considered that a CMS-IL model can be used to generate a complete CMS-based web application, as further developer intervention is not required.

G – Independent from deployment environment. CMS-ML and CMS-IL models can be considered as being independent from the environment in which they are deployed, because (1) they do not provide any concepts that are used only in a specific CMS system, and (2) they are ideally meant to be deployed in a CMS system that has a CMS Model Interpreter component installed. Although CMS-IL does support the definition of platform-specific code (by means of the `Platform Lambda` and `Environment Snippet` concepts), it is possible for a CMS-IL Toolkit Developer to specify equivalent code for other platforms, thus allowing the removal of any platform-specificity in a CMS-IL model.

10.2 CMS-ML

The CMS-ML modeling language, described in Chapter 7, was designed to (1) be used by non-technical stakeholders – called Business Designers in the development approach presented in Chapter 6 –, and (2) be able to model CMS-based web applications with a moderate degree of complexity. Thus, the validation of CMS-ML was performed by using a set of small illustrative case studies, which were used to validate its usability by non-technical stakeholders (not trained in the implementation of CMS-based web applications), and its usefulness at modeling the intended web applications. In this section, we present the most significant CMS-ML validation case studies that were conducted.

10.2.1 Validation with Non-Technical Stakeholders

As was previously mentioned in Chapter 1, this research work was performed by using the Action Research method, which is based on a continuous cycle of (1) analyzing the problem, (2) finding a course of action that is a potential solution, (3) applying that course of actions in a case study involving “real” participants (i.e., non-researcher participants that actually face that problem in their lives), and (4) analyzing the results to determine whether that course of action indeed solved the problem.

When considering the aforementioned CMS-ML goals (and the language’s target audience), it became necessary to ascertain whether CMS-ML was indeed usable by non-technical stakeholders. In order to address this validation aspect, we chose to gather a small focus group, consisting of participants that were aware of the *usage* of CMS systems, but not of *how* CMS-based web applications are developed (namely regarding technical concepts, such as session-state handling or HTTP request processing). Those participants would then be asked to model a specific CMS-based web application while using only the CMS-ML modeling language.

Thus, CMS-ML was used in the 2009–2010¹ and 2010–2011² editions of the Computer-Supported Collaborative Work (CSCW) discipline, an optional course that is offered to students of the Bologna Master Degree in Information Systems and Computer Engineering (MEIC/b) at Instituto Superior Técnico (IST).

In each occasion, students were assigned the task of designing a CMS-based web application, aimed at fulfilling a specific set of requirements, while using CMS-ML. To do so, they were provided the “CMS-ML User’s Guide” [SS 10.c], as well as a Microsoft Visio stencil³ containing the various CMS-ML modeling elements. The results of these case studies are described in the following paragraphs.

2009–2010: the Lua newspaper. The students of CSCW’s 2009–2010 edition were tasked with modeling a web application for a fictional newspaper entitled “Lua”. This web application should (1) be based on an existing CMS (which students had learned to use in a previous stage of the discipline), and (2) effectively support the newspaper’s most relevant processes (namely regarding news publishing, comment moderation, reader navigation throughout the newspaper’s website, and advertisement-based monetization of the website).

¹Instituto Superior Técnico, “Trabalho Cooperativo Suportado por Computador (2º Sem. 2009/2010)”, <<https://fenix.ist.utl.pt/disciplinas/tcsc/2009-2010/2-semester>> (accessed on June 11th, 2012)

²Instituto Superior Técnico, “Trabalho Cooperativo Suportado por Computador (2º Sem. 2010/2011)”, <<https://fenix.ist.utl.pt/disciplinas/tcsc3/2010-2011/2-semester>> (accessed on June 11th, 2012)

³“Microsoft Visio”, <<http://office.microsoft.com/en-us/visio>> (accessed on June 11th, 2012)

This experiment was performed using the *first* version of CMS-ML, and the feedback obtained allowed us to detect problems that other CMS-ML Template and Toolkit designers were also likely to encounter when specifying their models. Students did report that, aside from the detected problems, the language was simple to use, and that they could easily look at a model and understand what it meant.

As a result of these problems, a small set of modeling elements was discarded (namely in the Toolkit Modeling model, see Figure 7.3), and some changes were made to the semantics and the concrete syntax of other elements. Nevertheless, the most significant change that resulted from this experiment was the removal of the Navigation view (which provided elements that supported navigation between different *Dynamic WebPages*), as students reported that this view would typically result in a “spaghetti” of pages and links between them, without presenting significant added-value to the model (as such navigation links could be configured in the target CMS system afterward).

2010–2011: the HUHO website. The 2010–2011 edition’s students were also tasked with modeling a web application, this time to support a fictional non-profit charity organization called “Help Us Help Others” (HUHO). Like in the previous edition, this web application was to (1) be based on an existing CMS, and (2) support a small set of HUHO’s activities (namely the edition and publishing of HUHO-related news, reader feedback, the submission and sharing of links, and a advertisement-based sponsoring mechanism).

This experiment was conducted with a revised version of CMS-ML, and yielded more favorable results than the previous 2009–2010 experiment. Once again, students considered the language to be easy to understand and apply. A student did suggest that the Permissions view should support user-defined permissions (instead of those specified *a priori* by the language); however, upon further discussion of this subject (namely regarding the complexity for the target audience to add such kinds of permission), the student came to the conclusion that this addition would probably not be worthwhile. Aside from this suggestion, students did not present other suggestions regarding either the addition or removal of modeling elements to the language, or its usability.

10.2.2 WebC-Docs: a CMS-based Document Management System

In addition to the validation case studies presented above, we have also used the CMS-ML modeling language to model other kinds of CMS-based web application. In this section, we present one of the most relevant web applications that we have modeled using CMS-ML: the WebC-Docs document management system.

WebC-Docs [SS 09_b, SS 11_a] is a Document Management System (DMS), originally implemented by SIQuant as a web application based on the WebComfort CMS [SS 08_b]. We consider this web application to feature a moderate degree of complexity, as it endows users with a considerable set of functionalities, including: (1) indexing and search capabilities, supported by the Lucene.Net search engine library⁴; (2) a set of configuration options that adapt the web application to a variety of scenarios; and (3) a management interface that uses a file-system metaphor to display *documents* and organize them in *folders*. In the following paragraphs, we provide an overview of the CMS-ML model of WebC-Docs that was produced in the context of our validation efforts. It should be noted that we do not include the usage of **Annotations** in this model, to keep text and images brief.

WebSite Template. The WebSite Template consists of a small set of **Dynamic WebPages** that present WebC-Docs' user with a front page, as well as functionalities for searching and managing documents (the user interface for the configuration of WebC-Docs is an aspect that should be addressed by the target CMS system). Figure 10.1 illustrates the Macro Structure for the WebC-Docs web application, as well as the WebSite's import of the WebC-Docs Toolkit.

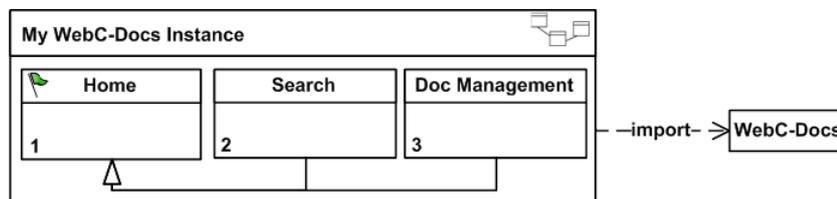
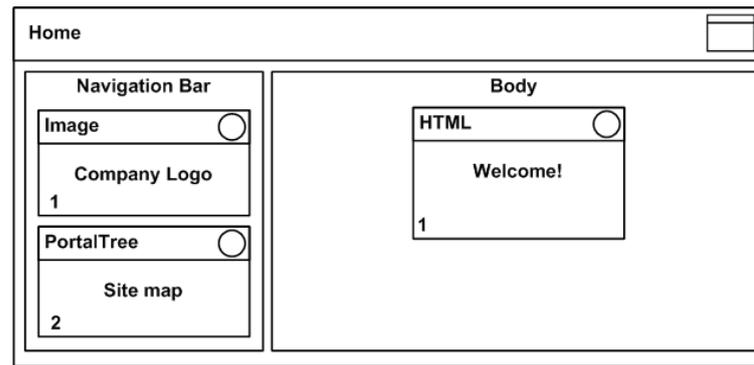


Figure 10.1: CMS-ML WebSite Template for the WebC-Docs web application: Macro Structure view.

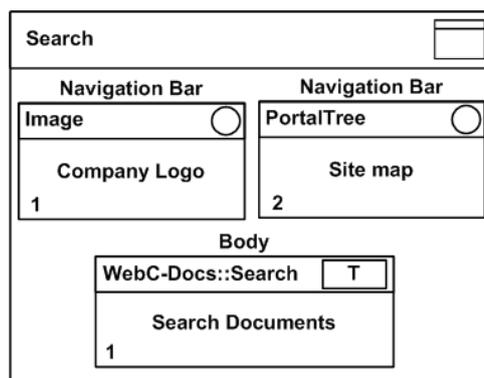
These **Dynamic WebPages** have the same layout (as indicated by the **Dynamic WebPage Template** relationships), and each presents the user with only a few **WebComponents**. More specifically, the **Home WebPage** presents a welcome message, the **Search WebPage** contains a **WebComponent** for searching documents, and the **Doc Management WebPage** provides a **Manage Docs WebComponent** (the latter two being provided by the WebC-Docs Toolkit). Figure 10.2 depicts the Micro Structure view for these **WebPages**.

The **Roles** view is also very simple, as it declares only three **Roles** (illustrated in Figure 10.3): **Operator**, **Manager**, and **Viewer**. The **Operator** should have the ability to add, edit, or remove documents; however, it should not be able to change or configure WebC-Docs or the website itself. Also, the **Manager** should be allowed to do *anything*

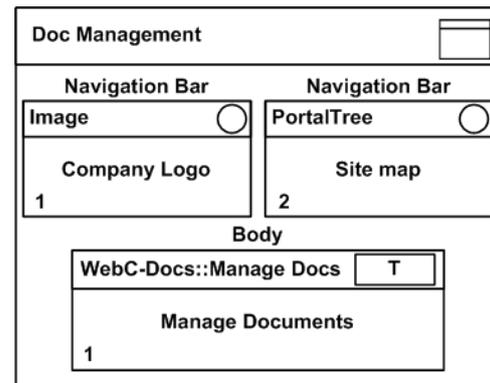
⁴“Apache Lucene.Net”, <<http://incubator.apache.org/lucene.net>> (accessed on June 12th, 2012)



(a) The Home page.



(b) The Search page.



(c) The Doc Management page.

Figure 10.2: CMS-ML WebSite Template for WebC-Docs: Micro Structure view.

regarding WebC-Docs' operation, including the configuration of its **WebComponents**. On the other hand, the **Viewer** should only be able to *view* documents (but not change anything in WebC-Docs or the website). These are all regular **Roles** (i.e., this Template does not define any administrator or anonymous **Roles**), and are instances of the homonym **Roles** that are defined in the WebC-Docs Toolkit. Furthermore, there are no requirements involving these **Roles** that warrant the delegation of responsibilities between them.



Figure 10.3: CMS-ML WebSite Template for WebC-Docs: Roles view.

Finally, the Permissions view – depicted in Figure 10.4 – is only used for setting some non-default permission values:

- Regarding **Dynamic WebPage Permissions**, this Template only defines permissions to (1) restrict the **Viewer**'s access to the **Doc Management WebPage**, and (2) grant some **WebPage** configuration permissions to the **Manager** (as shown in Figure 10.4a);

- The Template also defines finer-grained permissions for the `WebComponents` in the aforementioned `Dynamic WebPages`. In particular, these permissions (1) allow the `Manager` to configure the `Search Documents` and `Manage Documents WebComponents`, (2) allow the `Operator` to edit the content (i.e., documents) of the `Search Documents` and `Manage Documents WebComponents`, (3) prevent the `Operator` from configuring the `Manage Documents WebComponent`, and (4) prevent the `Viewer` from accessing the `Manage Documents WebComponent` in any way whatsoever. Figures 10.4b and 10.4c illustrate the `WebComponent Permissions` for the `WebComponents` in the `Search` and `Doc Management WebPages`, respectively.

Pages	Roles		
	Operator	Manager	Viewer
Home	View		
Search	View		
	Configure	✓	
Doc Management	View		×
	Configure	✓	

(a) Dynamic WebPage Permissions.

Search				
WebComponents	Roles			
	Operator	Manager	Viewer	
Search Documents	View			
	Configure		✓	
	EditContent	✓	✓	

(b) Search Page: WebComponent Permissions.

Doc Management				
WebComponents	Roles			
	Operator	Manager	Viewer	
Manage Documents	View			×
	Configure	×	✓	
	EditContent	✓	✓	

(c) Doc Management Page: WebComponent Permissions.

Figure 10.4: CMS-ML WebSite Template for WebC-Docs: Permissions view.

Toolkit. The CMS-ML Toolkit for the WebC-Docs web application is more extensive than the WebSite Template presented in the previous section, because of the greater number of views. Thus, to make this case study easier to understand, we have opted to address only some of the functionality provided by the original WebC-Docs implementation.

We begin the description of the WebC-Docs Toolkit by its `Roles` view, which is illustrated in Figure 10.5. The `Roles` considered by this Toolkit are the same as those that were discussed in the WebSite Template (and with the same responsibilities): `Operator`, `Manager`, and `Viewer`. Unlike the Template’s `Roles` view, however, these `Roles` are related to each other by `Role Specializations`: (1) `Operator` is a specialization of `Viewer`, because someone with the `Operator` Role should be able to perform the same `Tasks` as a `Viewer` (of course, an `Operator` should be able to perform other `Tasks` as well); and (2) `Manager` is

a specialization of `Operator` (and, by transitivity, of `Viewer`), because a `Manager` should be able to perform a superset of the `Tasks` performed by an `Operator`. It should be noted that the `Template Roles` presented in the previous section are *instances* of these `Roles` (i.e., they are related to each other by ReMMM’s `Instantiation` relationship).

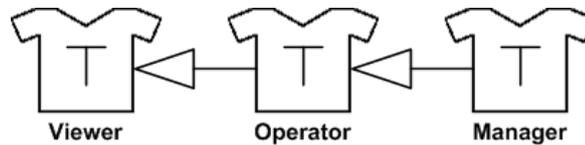
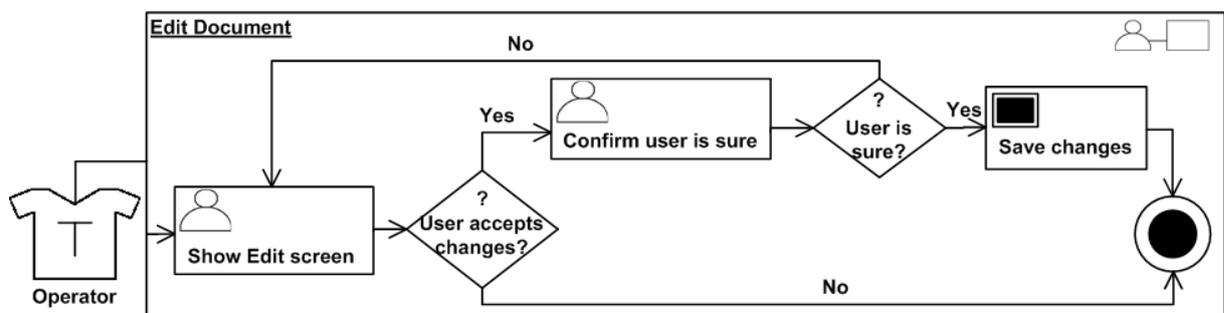
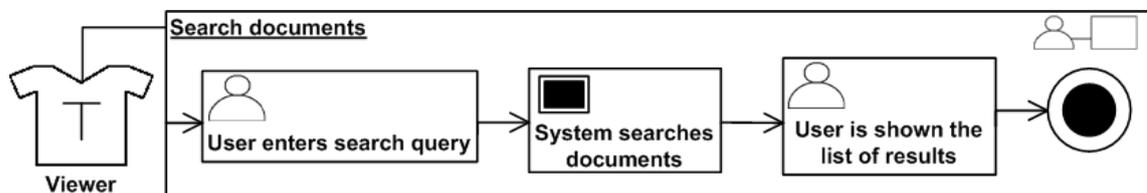


Figure 10.5: CMS-ML Toolkit for WebC-Docs: Roles view.

These `Roles` will be responsible for performing the `Tasks` that concern the viewing and management of documents. Figure 10.6 provides a few examples of `Tasks` that have been defined in this CMS-ML model: (1) Figure 10.6a depicts the `Edit Document Task`, which allows a user to edit a document, and afterward save its changes or discard them; and (2) Figure 10.6b presents the `Search Documents Task`, in which a user enters a search query and the system provides the search’s result. Although there is a much wider variety of `Tasks` that are supported by WebC-Docs, for simplicity reasons we have opted to constrain this validation’s description to a small set of `Tasks` that we consider as adequately illustrative of this language.



(a) The Edit Document Task.



(b) The Search Documents Task.

Figure 10.6: CMS-ML Toolkit for WebC-Docs: Tasks view.

The `Domain` view consists of defining the domain `Entities` for this Toolkit. Although the original WebC-Docs implementation contains the definition of a large number of `Entities` (including `Labels`, `Attributes`, or `Shortcuts`), to simplify this demonstration we

constrain this Toolkit to handle only *folders* and *documents*. This leads to the definition of the `Folder` and `Document` Entities, as depicted in Figure 10.7.

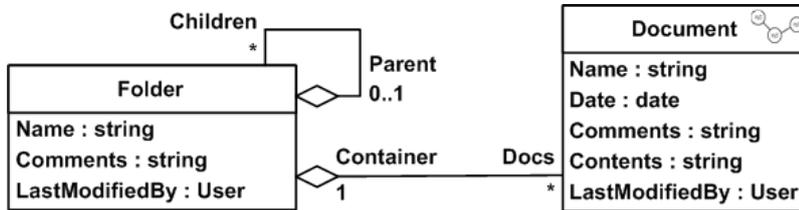


Figure 10.7: CMS-ML Toolkit for WebC-Docs: Domain view.

The States view for this example is very simple, and is only used to define a `Lifecycle` for the `Document` Entity (the existence of which was indicated in Figure 10.7). More specifically, this `Lifecycle` consists of two `States` – `Published` and `Deleted` – which allow a `Document` to be either published or deleted. However, once the `Document` is in the `Deleted` State, it cannot go back to the `Published` State. Furthermore, `Document` instances are never truly lost, because its `Lifecycle` does not contain any end `States`. Figure 10.8 provides an illustration of this view.

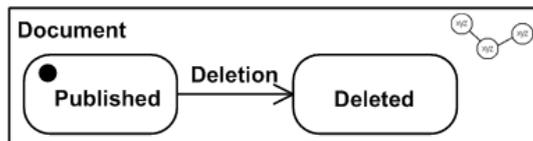


Figure 10.8: CMS-ML Toolkit for WebC-Docs: States view.

In the `WebComponents` view, we define the `Search` and `Manage Docs` `WebComponents` (which are used in the `WebSite Template` presented earlier). Figure 10.9 represents the definition of these `WebComponents`.

The `Manage Docs` `WebComponent`, depicted in Figure 10.9a, contains: (1) a list of `Folders` on the left side of the UI; (2) a (possibly empty) list of `Documents` on the right side; and (3) a set of action `Buttons` located next to each of those lists. The list of `Folders` is represented as an `HTML` `WebComponent` – along with an `Additional Toolkit Feature` – because it should be represented in a tree-like manner, which isn’t supported by any of the `WebElements` provided by CMS-ML. This `WebComponent` is also supported by the `Edit Document Support` `WebPage` (represented in Figure 10.9b), which provides some text boxes that can be used to change the state of the `Document` instance being edited (available as `document`).

On the other hand, the `Search` `WebComponent` (illustrated in Figure 10.9c) provides a text box where users can enter their search queries, and some sorting options (supported by `Selection Box` `WebElements`). The search results are shown in a `Table` `WebElement`

(a) The Manage Docs WebComponent.

(b) The Edit Document Support WebPage.

(c) The Search WebComponent.

Figure 10.9: CMS-ML Toolkit for WebC-Docs: WebComponents view.

Container, which displays each document’s Name and Date (as per Document’s definition in the Domain view).

Although the Side Effects view is considered optional, we consider it relevant to present the Side Effect that will occur when the user chooses to save the changes made to a Document instance (in the context of the Edit Document Support WebPage). This Side Effect, shown in Figure 10.10, indicates that the Save Changes Action (modeled in the Tasks view) consists of changing the values of the various attributes in the Document instance that is being edited. It is important to remember that modeled Side Effects are used *only* to inform the System Designer of changes to make to the system’s state, and are not used to perform any kind of storage access.

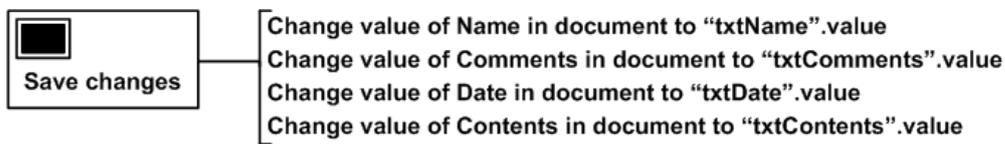


Figure 10.10: CMS-ML Toolkit for WebC-Docs: Side Effects view.

The Interaction Access view – represented in Figure 10.11 – is also very simple, as we only use it to restrict the Viewer Role’s access to the Manage Docs WebComponent and Edit Document Support WebPage. The Search WebComponent should be accessible to anyone, and so the corresponding WebInteractionSpace Access elements can be left with their default values. Furthermore, it is not necessary to specify WebElement Permissions in this model, because we do not consider the need to restrict access to specific WebElements.

		Roles		
WebComponents		Operator	Manager	Viewer
Search	View			
Manage Docs	View	✓	✓	×

(a) WebInteractionSpace Access (Roles and Web-Components).

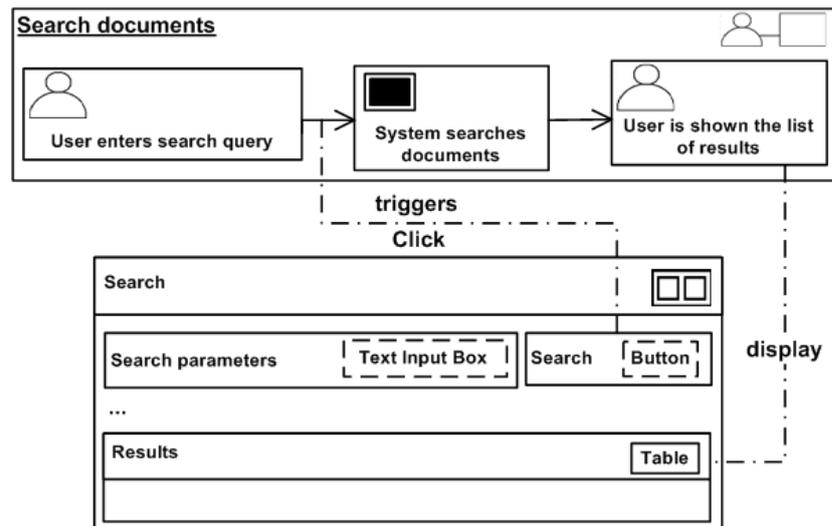
		Roles		
Support Pages		Operator	Manager	Viewer
Edit Document	View	✓	✓	×

(b) WebInteractionSpace Access (Roles and Support WebPages).

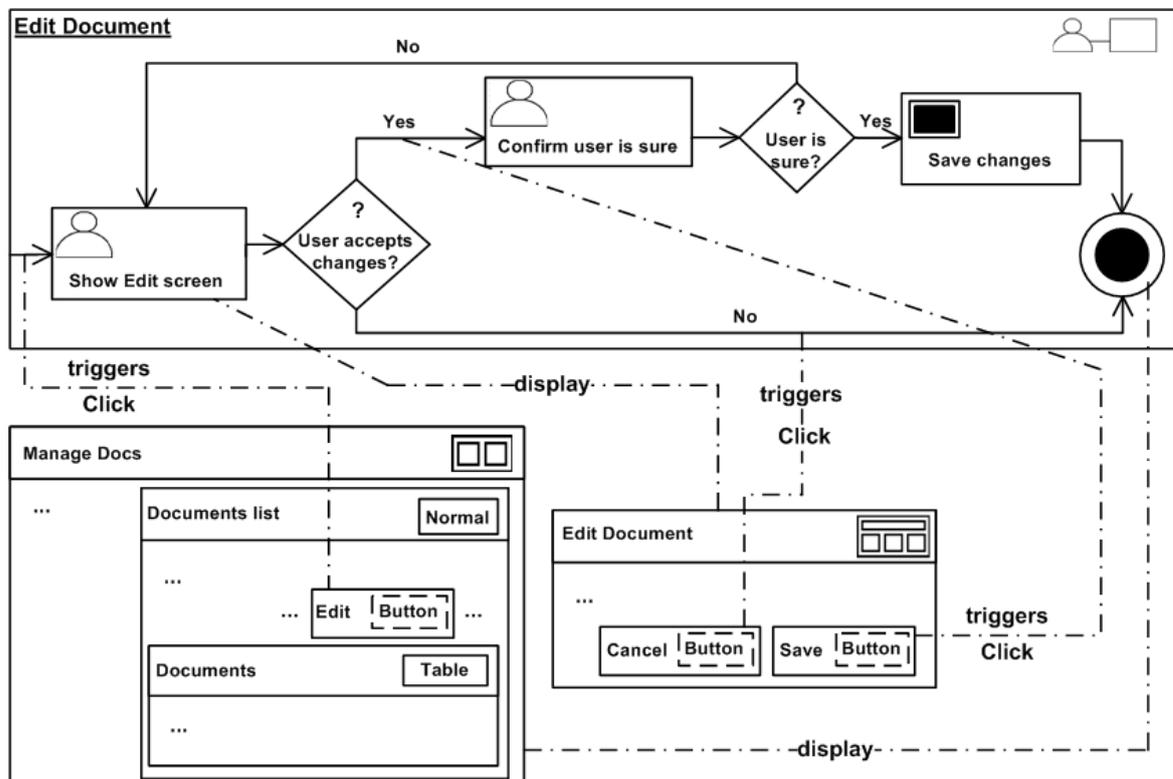
Figure 10.11: CMS-ML Toolkit for WebC-Docs: Interaction Access view.

Finally, the Interaction Triggers view establishes the mappings between the already modeled UI elements and the Toolkit’s Tasks (see Figures 10.9 and 10.6, respectively). More specifically, it uses the Display On Start element to define that the modeled Actions will be supported by specific WebComponents and Support WebPages (i.e., that whenever a certain Action is performed, the corresponding UI element should be shown to the user). It also establishes Triggers between UI elements and the Tasks view’s elements (Actions and Transitions), representing that certain actions over those UI elements will trigger the execution of those Task elements. Figure 10.12 depicts the

mappings that involve the Actions and UI elements already modeled in this example (to facilitate the interpretation of these diagrams, some irrelevant elements – already included in Figures 10.9 and 10.6 – have been omitted, and the Display On Start and Trigger lines are represented with a dashed pattern, in order to avoid confusion with other overlapping lines).



(a) Mappings for the Search Task.



(b) Mappings for the Edit Document Task.

Figure 10.12: CMS-ML Toolkit for WebC-Docs: Interaction Triggers view.

10.3 CMS-IL

Like with CMS-ML, we opted to perform the validation of CMS-IL via the WebC-Docs case study. More specifically, we validated it by defining a CMS-IL model of the WebC-Docs web application. Of course, this model was meant to reflect the original implementation [SS 09_b] as accurately as possible. The following paragraphs illustrate and explain the CMS-IL model that resulted from this validation effort.

WebSite Template. The CMS-IL WebSite Template for WebC-Docs is very similar to the CMS-ML Template that was presented in Section 10.2, and consists of an equivalent set of Dynamic WebPages and Roles (as well as the permission mappings between them). Listing 10.1 provides the representation of this Template.

Listing 10.1: CMS-IL WebSite Template of WebC-Docs.

```

1 WebSite Template consists of
2
3 WebSite "My WebC-Docs Instance"
4   imports Toolkit "WebC-Docs"
5   has
6   HomePage "Home" with
7     Container "Navigation Bar" at (0.5%, 0.5%, 20%, 99%) with
8       WebComponent "Company Logo" of Standard type "Image"
9       WebComponent "Site map" of Standard type "PortalTree"
10    Container "Body" at (21%, 0.5%, 78.5%, 99%) with
11      WebComponent "Welcome!" of Standard type "HTML"
12  Page "Search" follows layout of Page "Home" with
13    WebComponent "Company Logo" of Standard type "Image" in Container "Navigation Bar"
14    WebComponent "Site map" of Standard type "PortalTree" in Container "Navigation Bar"
15    WebComponent "Search Documents" ("WebC-Docs"::"Search") in Container "Body"
16  Page "Doc Management" follows layout of Page "Home" with
17    WebComponent "Company Logo" of Standard type "Image" in Container "Navigation Bar"
18    WebComponent "Site map" of Standard type "PortalTree" in Container "Navigation Bar"
19    WebComponent "Manage Documents" ("WebC-Docs"::"Manage Docs") in Container "Body"
20
21 Role "Operator" ("WebC-Docs"::"Operator")
22 Role "Manager" ("WebC-Docs"::"Manager")
23 Role "Viewer" ("WebC-Docs"::"Viewer")
24
25 Role "Operator" can edit content of WebComponent "Search"."Search Documents"
26 Role "Operator" can edit content of WebComponent "Doc Management"."Manage Documents"
27 Role "Operator" cannot configure WebComponent "Doc Management"."Manage Documents"
28 Role "Manager" can configure Page "Search"
29 Role "Manager" can configure Page "Doc Management"
30 Role "Manager" can (configure, edit content of) WebComponent "Search"."Search Documents"
31 Role "Manager" can (configure, edit content of) WebComponent "Doc Management"."Manage
32   Documents"
33 Role "Viewer" cannot view Page "Doc Management"
34 Role "Viewer" cannot view WebComponent "Doc Management"."Manage Documents"

```

This Template's Structure, Roles, and Permissions views (represented in lines 3–19, 21–23, and 25–33, respectively) are semantically equivalent to those of the aforementioned CMS-ML Template, and thus they will not be further discussed here.

The other CMS-IL Template views (which are useful only for application bootstrapping purposes) are not defined for the following reasons:

- The Users view is unnecessary, as WebC-Docs does not require the existence of any particular CMS user;
- The Languages view is not defined because WebC-Docs does not depend on any particular language;
- The Artifacts and Contents views are not relevant, because WebC-Docs does not require any of its `WebComponents` to have specific strings or contents assigned to them; and
- The Visual Themes view is not important, because WebC-Docs is not meant to *provide* out-of-the-box alternative visual layouts for its `WebComponents` and supporting pages.

Toolkit. The CMS-IL Toolkit model, which is depicted in Listing 10.2, can be considered as a match for the CMS-ML model presented in Section 10.2 (because of the length of this CMS-IL model, only some illustrative parts have been included in this listing, for text brevity). Nevertheless, it should be noted that the Toolkit views in both languages are considerably different, as discussed in Chapter 8.

Listing 10.2: CMS-IL Toolkit for the WebC-Docs web application.

```

1 Toolkit "WebC-Docs" consists of
2
3 Role "Operator"
4 Role "Manager"
5 Role "Viewer"
6 Role "Operator" specializes Role "Viewer"
7 Role "Manager" specializes Role "Operator"
8
9 Enumeration "Document Actions" (
10   "Create",
11   "Delete"
12 )
13 Function "Log Document Action"
14   receives ("Document Actions" action, "Document" document, User theUser)
15   performs {{
16     CMS.log "#{action} action performed by #{user} (document: #{document})", user:theUser,
17     action:action, document:document
18   }}
19 Event "Document Created" can occur with ("Document" document, User creator)
20 When event "Document Created" occurs
21   do lambda ("Document" document, User creator) = {{ CMS.call "Log Document Action", "Create",
138

```

```

22 Event "Document Deleted" can occur with ("Document" document)
23 When event "Document Deleted" occurs
24   do lambda ("Document" document) = {{ CMS.call "Log Document Action", "Delete", document,
    CMS.currentUser }}
25 Event "Manage Docs viewed" can occur
26 When event "Manage Docs viewed" occurs
27   do lambda = {{
28     CMS.log "The Manage Docs WebComponent has been accessed by user #{user} (IP address:
    #{ip_address})", user:CMS.currentUser || "anonymous", ip_address:CMS.currentRequest.ipAddress
29   }}
30
31 Variability Category "WebC-Docs Configuration"
32 contains (
33   Category "Documents"
34     defines
35     Point "Viewers must be authenticated" is boolean with default value "true"
36     when set do lambda (newValue) = {{ CMS.log "VarPoint 'Authenticated Viewers' now has
    value #{val}", val:newValue }}
37   Category "Roles"
38     defines (
39     Point "Manager" is Role
40     Point "Operator" is Role
41     Point "Viewer" is Role
42   )
43 )
44
45 Entity "Folder" has
46   Attribute "Name" as string
47   Attribute "Comments" as string
48   Attribute "LastModifiedBy" as User
49   Method "getDocumentCount"
50     returns integer
51     performs {{ return self."Docs".count }}
52   Method "getDocuments"
53     returns "Document" [*]
54     performs {{ return self."Docs" }}
55
56 Entity "Document" has
57   Attribute "Name" as string
58   Attribute "Date" as date
59   Attribute "Comments" as string
60   Attribute "Contents" as string
61   Attribute "LastModifiedBy" as User
62   Method "getAgeInDays"
63     returns integer
64     performs {{
65     result = date.Today - self."Date"
66     result = result.inDays
67     return result
68   }}
69   Method "getSizelnBytes"
70     returns integer
71     performs {{ return self."Contents".bytesize }}
72

```

```

73 Entity "Folder" (as 0..1 "Parent") contains Entity "Folder" (as * "Children") as
    "Folder_Parent...Folder_Children"
74 Entity "Folder" (as 1 "Container") contains Entity "Document" (as * "Docs") as
    "Folder_Container...Folder_Docs"
75
76 Customizable WebComponent "Search"
77 as <
78   VerticalLayout <
79     HorizontalLayout <
80       TextInputBox "Search parameters" (W:90%)
81       Button "Search"
82       on "Click" do lambda (sender) = {{
83         query = $Search parameters$.value
84         resultSet = CMS.call "System searches Documents", query
85         $Number of results$.value = ("Found #{number} hits".with number:resultSet.count)
86         $Results$.show resultSet
87       }} >
88     HorizontalLayout <
89       Text "Number of results" >
90     HorizontalLayout <
91       VerticalLayout (W:60%) <>
92       VerticalLayout <
93         HorizontalLayout <
94           Text "Sort by:"
95           SelectionBox "Last modification"
96           SelectionBox "In descending order" >
97     > >
98   VirtualContainer "Results" <
99     Link "Name" to Page "Manage Docs"."View Document" bound to "Document.Name"
100     Text "Document date" bound to "Document.Date" > > >
101
102 Customizable WebComponent "Manage Docs"
103 as <
104   HorizontalLayout <
105     NormalContainer "Folder list" (W:25%) <
106       VerticalLayout <
107         Text "Folders:"
108         HorizontalLayout <
109           Button "Add"
110           Button "Edit"
111           Button "View" >
112       !todo>Show the list of folders in a hierarchical tree<!
113       #"Folder tree"
114       Button "Delete" > >
115     NormalContainer "Documents list" <
116       VerticalLayout <
117         Text "Documents:"
118         HorizontalLayout <
119           Button "Add"
120           Button "Edit"
121           on "Click" do lambda (sender) = {{ document = $Documents$.current ; CMS.call "Show
            Edit screen", document }}
122         Button "View"
123         Button "Move" >
124       VirtualContainer "Documents" <

```

```

125     Link "Name" to Page "View Document" bound to "Document.Name"
126     Text "Document date" bound to "Document.Date"
127     Text "Last modified by" bound to "Document.LastModifiedBy" >
128     Button "Delete" > > >
129 >
130 on "Initial Display" do lambda = {{ CMS.trigger "Manage Docs viewed" }}
131 is supported by
132   WebPage "Edit Document"
133     expects Entity "Document" as "document"
134     as <
135       VerticalLayout <
136         HorizontalLayout <
137           Text "Name:"
138           TextBox "txtName" bound to "document.Name" >
139         HorizontalLayout <
140           Text "Date:"
141           TextBox "txtDate" bound to "document.Date" >
142         HorizontalLayout <
143           Text "Comments:"
144           TextBox "txtComments" bound to "document.Comments" >
145         HorizontalLayout <
146           Text "Contents:"
147           TextBox "txtContents" bound to "document.Contents" >
148         HorizontalLayout <
149           Button "Cancel"
150             on "Click" do lambda (sender) = {{ CMS.goBackToFrontEnd }}
151           Button "Save"
152             on "Click" do lambda (sender) = {{ CMS.call "Confirm user is sure" }} > > >

```

The Roles view (in lines 3–7) does not require any introduction, because it is semantically equivalent to its homonym view in the aforementioned CMS-ML Toolkit model.

The Code view, in turn, is provided as a set of `Lambdas` and `Functions` that are mentioned (and used) in the views described in the following paragraphs.

The Events view, depicted in lines 9–29, provides some basic document management-related `Events`, as well as a small set of `Event Handlers` that are called when those `Events` occur. More specifically, this view defines: (1) a `Function`, called `Log Document Action`, that is used to log operations that occur over documents; (2) the `Document Created Event` and an `Event Handler` that calls the `Log Document Action Function`; (3) a likewise `Event` – `Document Deleted` – and `Event Handler`; and (4) a `Manage Docs viewed Event` and an `Event Handler` that logs such occurrences.

The Variability view is specified in lines 31–43, and provides two `Variability Point Categories`, `Documents` and `Roles`. The `Documents Category` contains a single `Variability Point`, as well as a `Lambda` that logs new values when this `Point` is set. On the other hand, the `Roles Category` defines three `Variability Points` that, at runtime, determine which CMS Roles will be considered to be instances of the `Roles` defined in the Toolkit's Roles view. The reason why the values for these `Variability Points` are not explicitly

stored in the domain model (or using a similar mechanism) is that the CMS system itself should store these values; the Toolkit’s code can then access these values via the CMS utility class (e.g., a `Lambda` can test if a certain CMS Role `role` is an instance of WebC-Docs’ `Manager Role` by running⁵:

```
1 | role is_instance_of? (CMS.valueOfVarPoint "WebC-Docs Configuration":"Roles":"Manager")
```

which returns a boolean value that can be used in the context of conditional statements).

The Domain view, represented in lines 45–74, is also defined in the same manner as in the CMS-ML model presented in Section 10.2. More specifically, it defines the `Document` and `Folder` Entities, with the same set of `Attributes` and some `Associations` between these Entities. However, it also defines some `Methods`: (1) `Folder` is augmented with the `Methods` `getDocumentCount` and `getDocuments`, which respectively return the number of documents contained and those documents themselves; while (2) `Document` gets the `Methods` `getAgeInDays` and `getSizeInBytes`, which return the document’s age (in days) and the size of the document’s contents, respectively.

Finally, the `WebComponents` view is provided in lines 76–152, and is nearly equivalent to the one in the CMS-ML model documented earlier in this chapter (for simplicity purposes, purely graphical issues – such as positioning – are not reflected in this model). More specifically: (1) lines 76–100 define the `Search WebComponent`, in a manner that mimics its CMS-ML counterpart (see Figures 10.9c and 10.12a); (2) lines 102–130 establishes the `Manage Docs WebComponent` so that it is equivalent to Figures 10.9a and 10.12b; and (3) lines 132–152 define the `Edit Document Support WebPage` in the same manner as Figures 10.9b and 10.12b. Nevertheless, some additional `Event Handlers` (besides those implied by the aforementioned CMS-ML model) are also included in this example, for illustrative purposes.

10.4 MYNK

The MYNK model synchronization language was validated through the definition of a `Synchronization Spec`, meant to synchronize changes between models of the languages proposed in this dissertation (CMS-ML and CMS-IL). An additional validation effort was performed, consisting of a small `Sync Spec` to synchronize changes between CMS-ML and UML models. These `Sync Specs` are presented in the following subsections (the complete definition of these `Sync Specs` is not included in this dissertation, for text brevity).

⁵It should be noted that Ruby’s `instance_of?` method indicates whether an object is an instance of *exactly* the specified class (more details can be found at http://ruby-doc.org/core-1.9.3/Object.html#method-i-instance_of-3F). However, in order to make CMS-IL easier to read and use by its target audience, its `is_instance_of?` keyword is not treated so strictly.

10.4.1 Synchronizing CMS-ML and CMS-IL Changes

The `Sync Spec` that synchronizes changes between CMS-ML and CMS-IL models contains an extensive set of `Sync Rules`, each of which is responsible for matching a set of `Changes` in the CMS-ML (or CMS-IL) model, and applying a corresponding set of `Changes` to the other model. Listing 10.3 presents the definition of this `Sync Spec`.

Listing 10.3: Synchronization Spec: CMS-ML and CMS-IL.

```

1 Synchronization
2 between (cmsml_model in "CMS-ML", cmsil_model in "CMS-IL")
3 consists of {
4   Rule "WST: Create WebSite" [
5     CreateModelElement(model: cmsml_model, class: SELECT "WebSiteTemplate.WebSite", element:
6       theCMSMLWebSite)
7     ChangeAttributeValue(element: theCMSMLWebSite, attributeName: "Name", newValue:
8       newName)
9     NewTrace(theCMSMLWebSite as "CMS-ML WebSite", theCMSILWebSite as "CMS-IL WebSite")
10    <->
11    CreateModelElement(model: cmsil_model, class: SELECT "WebSiteTemplate.WebSite", element:
12      theCMSILWebSite)
13    ChangeAttributeValue(element: theCMSILWebSite, attributeName: "Name", newValue: newName)
14    ChangeAttributeValue(element: theCMSILWebSite, attributeName: "Description", newValue:
15      newDescription OR "")
16    NewTrace(theCMSMLWebSite as "CMS-ML WebSite", theCMSILWebSite as "CMS-IL WebSite" ) ]
17
18   Rule "WST: Create Container" [
19     CreateModelElement(model: cmsml_model, class: SELECT "WebSiteTemplate.Container",
20       element: theCMSMLContainer)
21     ChangeAttributeValue(element: theCMSMLContainer, attributeName: "Name", newValue:
22       newName)
23     ChangeAttributeValue(element: theCMSMLContainer, attributeName: "Left", newValue: newLeft)
24     ChangeAttributeValue(element: theCMSMLContainer, attributeName: "Top", newValue: newTop)
25     ChangeAttributeValue(element: theCMSMLContainer, attributeName: "Width", newValue:
26       newWidth)
27     ChangeAttributeValue(element: theCMSMLContainer, attributeName: "Height", newValue:
28       newHeight)
29     NewTrace(theCMSMLContainer as "CMS-ML Container", theCMSILContainer as "CMS-IL
30       Container")
31    <->
32    CreateModelElement(model: cmsil_model, class: SELECT "WebSiteTemplate.Container", element:
33      theCMSILContainer)
34    ChangeAttributeValue(element: theCMSILContainer, attributeName: "Name", newValue:
35      newName)
36    ChangeAttributeValue(element: theCMSILContainer, attributeName: "Left", newValue: newLeft)
37    ChangeAttributeValue(element: theCMSILContainer, attributeName: "Top", newValue: newTop)
38    ChangeAttributeValue(element: theCMSILContainer, attributeName: "Width", newValue:
39      newWidth)
40    ChangeAttributeValue(element: theCMSILContainer, attributeName: "Height", newValue:
41      newHeight)
42    NewTrace(theCMSMLContainer as "CMS-ML Container", theCMSILContainer as "CMS-IL
43      Container" ) ]

```

```

32 Rule "WST: Change Container name" [
33   GetTrace(from: theCMSILContainer, role: "CMS-ML Container", result: theCMSMLContainer)
34   ChangeAttributeValue(model: cmsml_model, class: SELECT "WebSiteTemplate.Container",
    element: theCMSMLContainer, attributeName: "Name", oldValue: oldName, newValue:
    newName)
35   <->
36   GetTrace(from: theCMSMLWebComponent, role: "CMS-IL Container", result: theCMSILContainer)
37   ChangeAttributeValue(model: cmsil_model, class: SELECT "WebSiteTemplate.Container",
    element: theCMSILContainer, attributeName: "Name", oldValue: oldName, newValue:
    newName) ]
38
39 Rule "WST: Create Role" [
40   CreateModelElement(model: cmsml_model, class: SELECT "WebSiteTemplate.Role", element:
    theCMSMLRole)
41   NewTrace(theCMSMLRole as "CMS-ML Role", theCMSILRole as "CMS-IL Role")
42   ChangeAttributeValue(element: theCMSMLRole, attributeName: "Name", newValue: newName)
43
44   cmsmlRoleTypes => SELECT WHERE model: cmsml_model, class: SELECT "EnumerationValue",
    name: newCMSILRoleType.Name
45   newCMSMLRoleType => SELECT end2 WHERE end1: SELECT "WebSiteTemplate.RoleType",
    end2: cmsilRoleTypes
46   ChangeAttributeValue(element: theCMSMLRole, attributeName: "Type", newValue:
    newCMSMLRoleType)
47   <->
48   CreateModelElement(model: cmsil_model, class: SELECT "WebSiteTemplate.Role", element:
    theCMSILRole)
49   NewTrace(theCMSMLRole as "CMS-ML Role", theCMSILRole as "CMS-IL Role")
50   ChangeAttributeValue(element: theCMSILRole, attributeName: "Name", newValue: newName)
51
52   cmsilRoleTypes => SELECT WHERE model: cmsil_model, class: SELECT "EnumerationValue",
    name: newCMSMLRoleType.Name
53   newCMSILRoleType => SELECT end2 WHERE end1: SELECT "WebSiteTemplate.RoleType",
    end2: cmsilRoleTypes
54   ChangeAttributeValue(element: theCMSILRole, attributeName: "Type", newValue:
    newCMSILRoleType) ]
55
56 Rule "WST: Establish Role Delegation" [
57   CreateModelElement(model: cmsml_model, class: SELECT "WebSiteTemplate.RoleDelegation",
    element: theCMSMLRoleDel)
58   NewTrace(theCMSMLRoleDel as "CMS-ML Role Delegation", theCMSILRoleDel as "CMS-IL Role
    Delegation")
59
60   relRoleDelDelegatee => SELECT WHERE model: cmsml_model, type: Association, end1: SELECT
    "WebSiteTemplate.Role", end1.Name: "delegatee", end2: SELECT
    "WebSiteTemplate.RoleDelegation"
61   GetTrace(from: theCMSILDelegatee, role: "CMS-ML Role", result: theCMSMLDelegatee)
62   EstablishRelationship(model: cmsml_model, class: relRoleDelDelegatee, end1:
    theCMSMLDelegatee, end2: theCMSMLRoleDel)
63
64   relRoleDelDelegator => SELECT WHERE model: cmsml_model, type: Association, end1: SELECT
    "WebSiteTemplate.Role", end1.Name: "delegator", end2: SELECT
    "WebSiteTemplate.RoleDelegation"
65   GetTrace(from: theCMSILDelegator, role: "CMS-ML Role", result: theCMSMLDelegator)
66   EstablishRelationship(model: cmsml_model, class: relRoleDelDelegator, end1: theCMSMLDelegator,
    end2: theCMSMLRoleDel)

```

```

67 <->
68 CreateModelElement(model: cmsil_model, class: SELECT "WebSiteTemplate.RoleDelegation",
69 element: theCMSILRoleDel)
70 NewTrace(theCMSMLRoleDel as "CMS-ML Role Delegation", theCMSILRoleDel as "CMS-IL Role
71 Delegation")
72
73 relRoleDelDelegatee => SELECT WHERE model: cmsil_model, type: Association, end1: SELECT
74 "WebSiteTemplate.Role", end1.Name: "delegatee", end2: SELECT
75 "WebSiteTemplate.RoleDelegation"
76 GetTrace(from: theCMSMLDelegatee, role: "CMS-IL Role", result: theCMSILDelegatee)
77 EstablishRelationship(model: cmsil_model, class: relRoleDelDelegatee, end1: theCMSILDelegatee,
78 end2: theCMSILRoleDel)
79
80 relRoleDelDelegator => SELECT WHERE model: cmsil_model, type: Association, end1: SELECT
81 "WebSiteTemplate.Role", end1.Name: "delegator", end2: SELECT
82 "WebSiteTemplate.RoleDelegation"
83 GetTrace(from: theCMSMLDelegator, role: "CMS-IL Role", result: theCMSILDelegator)
84 EstablishRelationship(model: cmsil_model, class: relRoleDelDelegator, end1: theCMSILDelegator,
85 end2: theCMSILRoleDel) ]
86
87 Rule "WST: Create WebComponent Permission" [
88 CreateModelElement(model: cmsml_model, class: SELECT
89 "WebSiteTemplate.WebComponentPermission", element: theCMSMLWCP)
90 NewTrace(theCMSMLWCP as "CMS-ML WebComponent Permission",
91 theCMSILWCP as "CMS-IL WebComponent Permission")
92 ChangeAttributeValue(element: theCMSMLWCP, attributeName: "View", newValue: newView)
93 ChangeAttributeValue(element: theCMSMLWCP, attributeName: "EditContent", newValue:
94 newEditContent)
95 ChangeAttributeValue(element: theCMSMLWCP, attributeName: "Configure", newValue:
96 newConfigure)
97
98 GetTrace(from: theCMSILRole, role: "CMS-ML Role", result: theCMSMLRole)
99 EstablishRelationship(model: cmsml_model, end1: theCMSMLWCP, end2: theCMSMLRole)
100
101 GetTrace(from: theCMSILWebComponent, role: "CMS-ML WebComponent", result:
102 theCMSMLWebComponent)
103 EstablishRelationship(model: cmsml_model, end1: theCMSMLWCP, end2:
104 theCMSMLWebComponent)
105
106 <->
107 CreateModelElement(model: cmsil_model, class: SELECT
108 "WebSiteTemplate.WebComponentPermission", element: theCMSILWCP)
109 NewTrace(theCMSMLWCP as "CMS-ML WebComponent Permission",
110 theCMSILWCP as "CMS-IL WebComponent Permission")
111 ChangeAttributeValue(element: theCMSILWCP, attributeName: "View", newValue: newView)
112 ChangeAttributeValue(element: theCMSILWCP, attributeName: "EditContent", newValue:
113 newEditContent)
114 ChangeAttributeValue(element: theCMSILWCP, attributeName: "Configure", newValue:
115 newConfigure)
116 ChangeAttributeValue(element: theCMSILWCP, attributeName: "Manage", newValue: newManage
117 OR (SELECT False))
118
119 GetTrace(from: theCMSMLRole, role: "CMS-IL Role", result: theCMSILRole)
120 EstablishRelationship(model: cmsil_model, end1: theCMSILWCP, end2: theCMSILRole)

```

```

104     GetTrace(from: theCMSMLWebComponent, role: "CMS-IL WebComponent", result:
105         theCMSILWebComponent)
106     EstablishRelationship(model: cmsil_model, end1: theCMSILWCP, end2: theCMSILWebComponent) ]
107
108 Rule "WST: Delete WebComponent Permission" [
109     GetTrace(from: theCMSMLWCP, role: "CMS-ML WebComponent Permission", result:
110         theCMSILWCP)
111
112     relWCPWC => SELECT WHERE model: cmsml_model, type: Association, end1: SELECT
113         "WebSiteTemplate.WebComponentPermission", end2: SELECT
114         "WebSiteTemplate.WebComponent"
115     instanceRelWCPWC => SELECT WHERE class: relWCPWC, end1: theCMSILWCP
116     DeleteElement(element: instanceRelWCPWC)
117
118     relWCPR => SELECT WHERE model: cmsml_model, type: Association, end1: SELECT
119         "WebSiteTemplate.WebComponentPermission", end2: SELECT "WebSiteTemplate.Role"
120     instanceRelWCPR => SELECT WHERE class: relWCPR, end1: theCMSILWCP
121     DeleteElement(element: instanceRelWCPR)
122
123     DeleteElement(model: cmsml_model, class: SELECT
124         "WebSiteTemplate.WebComponentPermission", element: theCMSMLWCP)
125     RemoveTrace(theCMSMLWCP)
126 <->
127     GetTrace(from: theCMSMLWCP, role: "CMS-IL WebComponent Permission", result:
128         theCMSILWCP)
129
130     relWCPWC => SELECT WHERE model: cmsil_model, type: Association, end1: SELECT
131         "WebSiteTemplate.WebComponentPermission", end2: SELECT
132         "WebSiteTemplate.WebComponent"
133     instanceRelWCPWC => SELECT WHERE class: relWCPWC, end1: theCMSILWCP
134     DeleteElement(element: instanceRelWCPWC)
135
136     relWCPR => SELECT WHERE model: cmsil_model, type: Association, end1: SELECT
137         "WebSiteTemplate.WebComponentPermission", end2: SELECT "WebSiteTemplate.Role"
138     instanceRelWCPR => SELECT WHERE class: relWCPR, end1: theCMSILWCP
139     DeleteElement(element: instanceRelWCPR)
140
141     DeleteElement(model: cmsil_model, class: SELECT
142         "WebSiteTemplate.WebComponentPermission", element: theCMSILWCP)
143     RemoveTrace(theCMSILWCP) ]
144
145 Rule "Toolkit: Establish Interaction Trigger" [
146     When
147     CreateModelElement(model: cmsml_model, class: SELECT "Toolkit.Trigger", element: theTrigger)
148     ChangeAttributeValue(element: theTrigger, attributeName: "Type", newValue: newType)
149
150     EstablishRelationship(model: cmsml_model, end1: theTrigger, end2: theCMSMLWebElement)
151
152     EstablishRelationship(model: cmsml_model, end1: theTrigger, end2: theTask) WHEN theTask.class
153         IS "Toolkit.Task"
154     EstablishRelationship(model: cmsml_model, end1: theTrigger, end2: theActionTrnst) WHEN
155         theActionTrnst IS "Toolkit.ActionTransition"
156     do
157     GetTrace(from: theCMSMLWebElement, role: "CMS-IL WebElement", result:
158         theCMSILWebElement)

```

```

145
146 WHEN theCMSILWebElement IS "Toolkit.InteractionWebElement" (
147     relIWEH => SELECT WHERE model: cmsil_model, end1: SELECT
148         "Toolkit.InteractionWebElement", end2: SELECT "Toolkit.EventHandler"
149     CreateModelElement(model: cmsil_model, class: SELECT "Toolkit.EventHandler", element:
150         theHandler)
151     EstablishRelationship(model: cmsil_model, type: Association, class: relIWEH, end1:
152         theCMSILWebElement, end2: theHandler)
153     NewTrace(theTrigger as "CMS-ML Trigger", theHandler as "CMS-IL Event Handler")
154
155     relEHPL => SELECT WHERE model: cmsil_model, end1: SELECT "Toolkit.EventHandler",
156         end2: SELECT "Toolkit.PureLambda"
157     CreateModelElement(model: cmsil_model, class: SELECT "Toolkit.PureLambda", element:
158         theLambda)
159     EstablishRelationship(model: cmsil_model, type: Association, class: relEHPL, end1: theHandler,
160         end2: theLambda)
161
162     relPLS => SELECT WHERE model: cmsil_model, end1: SELECT "Toolkit.PureLambda", end2:
163         SELECT "Toolkit.Statement"
164     CreateModelElement(model: cmsil_model, class: SELECT "Toolkit.Statement", element:
165         theStmt)
166     EstablishRelationship(model: cmsil_model, type: Association, class: relPLS, end1: theLambda,
167         end2: theStmt)
168
169     relEHE => SELECT WHERE model: cmsil_model, end1: SELECT "Toolkit.EventHandler",
170         end2: SELECT "Toolkit.Event"
171
172     WHEN newType.Name = "View" (
173         cmsilWCEvents => SELECT WHERE model: cmsil_model, class: SELECT
174             "EnumerationValue", name: "Initial Display"
175     )
176     WHEN newType.Name = "Click" (
177         cmsilWCEvents => SELECT WHERE model: cmsil_model, class: SELECT
178             "EnumerationValue", name: "Click"
179     )
180
181     WHEN theActionTrnst (
182         relActTrnstTarget => SELECT WHERE model: cmsml_model, type: Association, end1:
183             SELECT "Toolkit.Action", end1.Name: "target", end2: SELECT
184                 "Toolkit.ActionTransition"
185         theAction => SELECT end1 WHERE model: cmsml_model, class: relActTrnstTarget, end2:
186             theActionTrnst
187         ChangeAttributeValue(element: theStmt, attributeName: "Value", newValue: ("CMS.call " +
188             theAction.Name))
189     )
190     WHEN theTask (
191         ChangeAttributeValue(element: theStmt, attributeName: "Value", newValue: ("CMS.call " +
192             theTask.Name))
193     )
194
195     theEvent => SELECT end2 WHERE end1: SELECT "Toolkit.WebComponentEvent", end2:
196         cmsilWCEvents
197     EstablishRelationship(model: cmsil_model, type: Association, class: relEHE, end1: theHandler,
198         end2: theEvent)
199 ) ] ]

```

We consider these **Sync Rules** to be adequately illustrative for this validation’s purpose, as they depict: (1) straightforward synchronization rules between CMS-ML and CMS-IL elements (in lines 4–37); (2) creating elements in another model and converting enumeration values between them (lines 39–54); (3) creating an instance of a concept that connects other concepts (as shown in the **Sync Rules** in lines 56–105); (4) the removal of an element (lines 107–132); and (5) the synchronization of an aspect – CMS-ML’s **Trigger** vs. CMS-IL’s **Events** and **Event Handlers** – that is handled differently by the two languages (defined in lines 134–180).

After defining these **Sync Rules**, we tested each of them by: (1) starting with the CMS-ML and CMS-IL models of WebC-Docs (which were defined in Sections 10.2 and 10.3, respectively); (2) making small changes (reflecting the **Changes** expected by that **Sync Rule**) to one of those models; and (3) determining whether the other model was correctly updated. Although we did not find insurmountable problems in this validation effort, we did notice the following issues with the current version of MYNK:

- Defining a set of **Sync Rules** that *completely* covers the possible **Changes** to both CMS-ML and CMS-IL would be a very extensive undertaking; and
- The lack of possibility to define helper functions could have made the definition of the **Create Role Sync Rule** (see line 39) much more complicated than it was. The reason why this **Sync Rule** is relatively simple is that the values defined by CMS-ML and CMS-IL **Role Type** enumerations have the *same names*, a circumstance which may not be found in synchronization scenarios involving other modeling languages.

These are problems that we consider should be given priority in future work.

10.4.2 Synchronizing CMS-ML and UML Changes

In addition to the **Synchronization Spec** presented above, we have also validated MYNK by defining a simple **Sync Spec** – depicted in Listing 10.4 – that synchronizes changes between CMS-ML and UML models. It is important to highlight that UML (or even MOF) can be specified using the ReMMM metamodel, which is a necessary characteristic for this **Sync Spec**’s definition to be possible (as per the requirements described in Chapter 9).

Although the definition of UML could be simpler if it was based on ReMMM (because some of its accidental complexity could be removed), in this **Sync Spec** we assume that UML is defined in the same manner as is presented in its specification document [OMG 11_e]. To simplify this validation effort, we limited this **Sync Spec** to handle only changes pertaining to CMS-ML’s Domain view, in turn corresponding to a typical UML class diagram. Furthermore, the names of the UML **Model Elements** and **Attributes** are the ones provided by the UML specification [OMG 11_e].

Listing 10.4: Synchronization Spec: CMS-ML and UML.

```

1 Synchronization
2   between (cmsml_model in "CMS-ML", uml_model in "UML")
3   consists of {
4     Rule "Create Entity" [
5       CreateModelElement(model: cmsml_model, class: SELECT "Toolkit.Entity", element:
6         theCMSMLEntity)
7       NewTrace(theCMSMLEntity as "CMS-ML Entity", theUMLClass as "UML Class")
8       ChangeAttributeValue(element: theCMSMLEntity, attributeName: "Name", newValue: newName)
9       ChangeAttributeValue(element: theCMSMLEntity, attributeName: "IsAbstract", newValue:
10        newIsAbstract)
11      <->
12      CreateModelElement(model: uml_model, class: SELECT "UML.Class", element: theUMLClass)
13      NewTrace(theCMSMLEntity as "CMS-ML Entity", theUMLClass as "UML Class")
14      ChangeAttributeValue(element: theUMLClass, attributeName: "name", newValue: newName)
15      ChangeAttributeValue(element: theUMLClass, attributeName: "isAbstract", newValue:
16        newIsAbstract) ]
17
18   Rule "Create Association between Entities" [
19     CreateModelElement(model: cmsml_model, class: SELECT "Toolkit.Association", element:
20       theCMSMLAssoc)
21     NewTrace(theCMSMLAssoc as "CMS-ML Association", theUMLAssoc as "UML Association")
22     ChangeAttributeValue(element: theCMSMLAssoc, attributeName: "Name", newValue:
23       newAssocName)
24
25     relAssocAssocRole => SELECT WHERE model: cmsml_model, type: Association, end1: SELECT
26       "Toolkit.Association", end2: SELECT "Toolkit.AssociationRole"
27     relAssocRoleEntity => SELECT WHERE model: cmsml_model, type: Association, end1: SELECT
28       "Toolkit.AssociationRole", end2: SELECT "Toolkit.Entity"
29
30     CreateModelElement(model: cmsml_model, class: SELECT "Toolkit.AssociationRole", element:
31       theCMSMLAssocRole1)
32     ChangeAttributeValue(element: theCMSMLAssocRole1, attributeName: "Name", newValue:
33       newAssocRole1Name)
34     ChangeAttributeValue(element: theCMSMLAssocRole1, attributeName: "Multiplicity_Lower",
35       newValue: newAssocRole1MultLower)
36     ChangeAttributeValue(element: theCMSMLAssocRole1, attributeName: "Multiplicity_Upper",
37       newValue: newAssocRole1MultUpper)
38     ChangeAttributeValue(element: theCMSMLAssocRole1, attributeName: "ContainsOtherEntity",
39       newValue: newAssocRole1ContainsOther)
40     EstablishRelationship(model: cmsml_model, class: relAssocAssocRole, end1: theCMSMLAssoc,
41       end2: theCMSMLAssocRole1)
42     EstablishRelationship(model: cmsml_model, class: relAssocRoleEntity, end1:
43       theCMSMLAssocRole1, end2: theCMSMLEntity1)
44
45     CreateModelElement(model: cmsml_model, class: SELECT "Toolkit.AssociationRole", element:
46       theCMSMLAssocRole2)
47     ChangeAttributeValue(element: theCMSMLAssocRole2, attributeName: "Name", newValue:
48       newAssocRole2Name)
49     ChangeAttributeValue(element: theCMSMLAssocRole2, attributeName: "Multiplicity_Lower",
50       newValue: newAssocRole2MultLower)
51     ChangeAttributeValue(element: theCMSMLAssocRole2, attributeName: "Multiplicity_Upper",
52       newValue: newAssocRole2MultUpper)
53     ChangeAttributeValue(element: theCMSMLAssocRole2, attributeName: "ContainsOtherEntity",
54       newValue: newAssocRole2ContainsOther)

```

```

36     EstablishRelationship(model: cmsml_model, class: relAssocAssocRole, end1: theCMSMLAssoc,
37         end2: theCMSMLAssocRole2)
38     EstablishRelationship(model: cmsml_model, class: relAssocRoleEntity, end1:
39         theCMSMLAssocRole2, end2: theCMSMLEntity2)
40     <->
41     CreateModelElement(model: uml_model, class: SELECT "UML.Association", element:
42         theUMLAssoc)
43     NewTrace(theCMSMLAssoc as "CMS-ML Association", theUMLAssoc as "UML Association")
44     ChangeAttributeValue(element: theUMLAssoc, attributeName: "name", newValue: newAssocName)
45
46     relAssocProperty => SELECT WHERE model: uml_model, end1: SELECT "UML.Association",
47         end2: SELECT "UML.Property", end2.Name: "ownedEnd"
48     relTET => SELECT WHERE model: uml_model, end1: SELECT "UML.TypedElement", end2:
49         SELECT "UML.Type"
50
51     CreateModelElement(model: uml_model, class: SELECT "UML.Property", element:
52         theUMLAssocProp1)
53     ChangeAttributeValue(element: theUMLAssocProp1, attributeName: "name", newValue:
54         newAssocRole1Name)
55     EstablishRelationship(model: uml_model, class: relAssocProperty, end1: theUMLAssoc, end2:
56         theUMLAssocProp1)
57     (...)
58
59     GetTrace(from: theCMSMLEntity1, role: "UML.DataType", result: theUMLType1)
60     EstablishRelationship(model: uml_model, class: relTET, end1: theUMLAssocProp1, end2:
61         theUMLType1)
62
63     CreateModelElement(model: uml_model, class: SELECT "UML.Property", element:
64         theUMLAssocProp2)
65     ChangeAttributeValue(element: theUMLAssocProp2, attributeName: "name", newValue:
66         newAssocRole2Name)
67     EstablishRelationship(model: uml_model, class: relAssocProperty, end1: theUMLAssoc, end2:
68         theUMLAssocProp2)
69     (...)
70
71     GetTrace(from: theCMSMLEntity2, role: "UML.DataType", result: theUMLType2)
72     EstablishRelationship(model: uml_model, class: relTET, end1: theUMLAssocProp2, end2:
73         theUMLType2) ]
74
75 Rule "Create Specialization" [
76     CreateModelElement(model: cmsml_model, class: SELECT "Toolkit.Specialization", element:
77         theCMSMLSpec)
78     NewTrace(theCMSMLSpec as "CMS-ML Specialization", theUMLGen as "UML Generalization")
79
80     relSpecBase => SELECT WHERE model: cmsml_model, type: Association, end1: SELECT
81         "Toolkit.Entity", end1.Name: "base", end2: SELECT "Toolkit.Specialization"
82     GetTrace(from: theUMLGeneral, role: "CMS-ML Class", result: theCMSMLBase)
83     EstablishRelationship(model: cmsml_model, class: relSpecBase, end1: theCMSMLBase, end2:
84         theCMSMLSpec)
85
86     relSpecInheritor => SELECT WHERE model: cmsml_model, type: Association, end1: SELECT
87         "Toolkit.Entity", end1.Name: "inheritor", end2: SELECT "Toolkit.Specialization"
88     GetTrace(from: theUMLSpecific, role: "CMS-ML Class", result: theCMSMLInheritor)
89     EstablishRelationship(model: cmsml_model, class: relSpecInheritor, end1: theCMSMLInheritor,
90         end2: theCMSMLSpec)

```

```

73 <->
74   CreateModelElement(model: uml_model, class: SELECT "UML.Generalization", element:
75     theUMLGen)
76   NewTrace(theCMSMLSpec as "CMS-ML Specialization", theUMLGen as "UML Generalization")
77
78   relClassGenSpecific => SELECT WHERE model: uml_model, end1: SELECT "UML.Classifier",
79     end1.Name: "specific", end2: SELECT "UML.Generalization"
80   GetTrace(from: theCMSMLInheritor, role: "UML Class", result: theUMLSpecific)
81   EstablishRelationship(model: uml_model, class: relClassGenSpecific, end1: theUMLSpecific, end2:
82     theUMLGen)
83
84   relGenClassGeneral => SELECT WHERE model: uml_model, end1: SELECT
85     "UML.Generalization", end2: SELECT "UML.Classifier", end2.Name: "general"
86   GetTrace(from: theCMSMLBase, role: "UML Class", result: theUMLGeneral)
87   EstablishRelationship(model: uml_model, class: relGenClassGeneral, end1: theUMLGen, end2:
88     theUMLGeneral) ]
89
90 Rule "Create Attribute" [
91   CreateModelElement(model: cmsml_model, class: SELECT "Toolkit.Attribute", element:
92     theCMSMLAttr)
93   NewTrace(theCMSMLAttr as "CMS-ML Attribute", theUMLProp as "UML Property")
94   ChangeAttributeValue(element: theCMSMLAttr, attributeName: "Name", newValue: newName)
95   ChangeAttributeValue(element: theCMSMLAttr, attributeName: "IsIdentificationCriteria",
96     newValue: newIsID)
97   ChangeAttributeValue(element: theCMSMLAttr, attributeName: "Multiplicity_Lower", newValue:
98     newMultLower)
99   ChangeAttributeValue(element: theCMSMLAttr, attributeName: "Multiplicity_Upper", newValue:
100     newMultUpper)
101   ChangeAttributeValue(element: theCMSMLAttr, attributeName: "Type", newValue:
102     newCMSMLType)
103
104 <->
105   CreateModelElement(model: uml_model, class: SELECT "UML.Property", element: theUMLProp)
106   NewTrace(theCMSMLAttr as "CMS-ML Attribute", theUMLProp as "UML Property")
107   ChangeAttributeValue(element: theUMLProp, attributeName: "name", newValue: newName)
108   ChangeAttributeValue(element: theUMLProp, attributeName: "isID", newValue: newIsID)
109
110   relMEVSLower => SELECT WHERE model: uml_model, end1: SELECT
111     "UML.MultiplicityElement", end2: SELECT "UML.ValueSpecification", end2.Name:
112     "lowerValue"
113   CreateModelElement(model: uml_model, class: SELECT "UML.Expression", element:
114     theMultLowerSpec)
115   ChangeAttributeValue(element: theMultLowerSpec, attributeName: "symbol", newValue:
116     newMultLower)
117   EstablishRelationship(model: uml_model, class: relMEVSLower, end1: theUMLProp, end2:
118     theMultLowerSpec)
119
120   relMEVUpper => SELECT WHERE model: uml_model, end1: SELECT
121     "UML.MultiplicityElement", end2: SELECT "UML.ValueSpecification", end2.Name:
122     "upperValue"
123   CreateModelElement(model: uml_model, class: SELECT "UML.Expression", element:
124     theMultUpperSpec)
125   ChangeAttributeValue(element: theMultUpperSpec, attributeName: "symbol", newValue:
126     newMultUpper)
127   EstablishRelationship(model: uml_model, class: relMEVUpper, end1: theUMLProp, end2:
128     theMultUpperSpec)

```

```

108     relTET => SELECT WHERE model: uml_model, end1: SELECT "UML.TypedElement", end2:
109         SELECT "UML.Type"
110     GetTrace(from: newCMSMLType, role: "UML.DataType", result: theUMLType)
111     EstablishRelationship(model: uml_model, class: relTET, end1: theUMLProp, end2: theUMLType) ]
112
113     Rule "Change Attribute Name" [
114         GetTrace(from: theUMLProp, role: "CMS-ML Attribute", result: theCMSMLAttr)
115         ChangeAttributeValue(model: cmsml_model, class: SELECT "Toolkit.Attribute", element:
116             theCMSMLAttr, attributeName: "Name", oldValue: oldName, newValue: newName)
117         <->
118         GetTrace(from: theCMSMLAttr, role: "UML Property", result: theUMLProp)
119         ChangeAttributeValue(model: uml_model, class: SELECT "UML.Property", element: theUMLProp,
120             attributeName: "Name", oldValue: oldName, newValue: newName) ] }

```

These **Sync Rules** are defined like the ones in the **CMS-ML↔CMS-IL Sync Spec** (presented in the previous section). More precisely, it establishes a correspondence between the following CMS-ML and UML concepts: (1) **Entities** and **Classes** (lines 4–13); (2) **Associations** (in both languages, see lines 15–60); (3) **Specializations** and **Generalizations** (lines 62–83); and (4) **Attributes** and **Properties** (lines 85–118).

The omitted fragments of lines 49 and 57 correspond to the configuration of the **Property’s** multiplicity information, an example of which can be viewed in the **Create Attribute Sync Rule** (in lines 85–111).

10.5 Research Work Assessment

To conclude the description of the efforts that were performed to validate our research results, we now present an assessment of the research questions that were established at the beginning of this work, in Chapter 1. We also take the opportunity to assess the research objective and the Thesis Statement that is defended in this dissertation.

10.5.1 Research Questions

The following paragraphs provide answers for the research questions that were defined in Chapter 1. These answers are derived from the results of the research work that was described in this dissertation.

How can an MDE approach effectively address the perspective that each kind of stakeholder will have regarding the web application to be developed? In order to properly support multiple kinds of stakeholder, an MDE approach needs to address: (1) the usage of a *set* of modeling languages; and (2) a model synchronization mechanism that enables the synchronization of changes between a set of models specified

in different languages (but which should all represent the same web application). The proposed MDE development approach is an example of this, by providing support for the CMS-ML and CMS-IL languages, and the MYNK model synchronization framework.

How can a modeling language (or an MDE approach) address the various abstraction levels that are necessary to specify a web application in practice?

A trivial example of this need for multiple abstraction levels is that the definition of components that can be reused throughout a website should be performed in a level that is above the one in which the website itself is modeled. Although it is possible for a modeling language to concentrate elements of various abstraction levels within a single metalevel, this ultimately leads to *accidental complexity* (as was explained in Chapter 2). This leads to the need of defining modeling languages that effectively support *multiple metalevels*, so that each abstraction level can be correctly placed in its corresponding metalevel.

One of the results from this research work is the ReMMM metamodeling language, which provides the `Instantiation` element (among others) and thus supports the definition of multiple metalevels. ReMMM, in turn, was used to define the CMS-ML and CMS-IL languages, which take advantage of this facility to provide different abstraction levels (realized in both languages by the WebSite Template and Toolkit models).

Another perspective for supporting multiple abstraction levels is the existence of multiple kinds of stakeholder, who deal with different levels of detail (e.g., technical and non-technical details). The proposed development approach handles this issue by providing support for multiple languages (for the Business Designer and the System Designer), as described in the answer to the previous research question.

Is it really necessary to have a single language adopt a compromise between the support for low-level details and the ease of its learning and use?

Every modeling language is ultimately a *compromise* between (1) the number of modeling elements it provides (which, in turn, may be considered as being related to its degree of expressiveness, syntactic sugar notwithstanding), (2) the abstraction level(s) supported, and (3) how easy it is for its target audience to learn to read and use the language.

However, the development approach proposed in this dissertation demonstrates that this modeling language compromise does not necessarily mean that an *approach* should also be a result of that same kind of compromise; instead, it can leverage the use of *multiple languages* to address details of various abstraction levels.

Would an approach that supports multiple abstraction levels be more productive than a traditional (i.e., manual) development approach? Considering

that productivity can be defined as a measure of production output vs. the cost of the resources (and effort) needed to create that output⁶, we consider that the proposed MDE approach is more productive than traditional development approaches because: (1) the web application’s high-level models (e.g., wireframes of its structure, or its domain model) are used to automatically obtain models in a lower level of detail; (2) it reduces – or eliminates, depending on how stakeholders actually apply the approach – the need for multiple *discuss*→*develop*→*test*→*correct* cycles; and (3) the various models of the web application are automatically kept in sync with each other, instead of requiring that developers (or other stakeholders) manually update each model whenever they change something. It should be noted, nevertheless, that this approach does not remove the need to define a web application’s requirements, or the need for the web application’s stakeholders to continuously discuss its roadmap.

Is it feasible to have models as the project’s *de facto* main artifact, instead of source code (the more traditional artifact)? As was explained in Chapter 2, source code can be considered as a model, in which the modeling language assumes a textual concrete syntax.

However, we consider that the proposed development approach – and its supporting languages – demonstrates that source code (to be processed by a typical compiler, such as C’s `gcc` or Java’s `javac`) does *not* need to be the project’s main artifact. Instead, it is demonstrated that the project’s important artifacts can in fact be a set of models specified in different modeling languages, as long as (1) those models are semantically equivalent among themselves, and (2) the expressiveness that results from that set of modeling languages is enough to cover the needs of the desired web application. The set of CMS-oriented modeling languages proposed in this dissertation provides such a degree of expressiveness, because (1) CMS-ML only addresses some high-level modeling aspects and is not particularly expressive, but (2) CMS-IL offers a degree of expressiveness comparable to that found in programming languages typically used for web application development.

Considering the aforementioned issues, is it possible for an MDE approach to adequately support the development of CMS-based web applications? We consider that our research work has yielded an affirmative answer for this question, as the proposed MDE approach allows technical and non-technical stakeholders to provide their own perspectives regarding the desired CMS-based web application. Furthermore, the CMS-IL language (namely its Toolkit Modeling model, see Chapter 8) provides all

⁶“What is productivity? definition and meaning”, *BusinessDictionary.com*, <<http://www.businessdictionary.com/definition/productivity.html>> (accessed on June 4th, 2012)

of the extensibility constructs that are supported by most CMS systems [SS 08_b]; the importance of these constructs comes from the fact that they are often used by developers, during their manually-performed coding activities, to define this kind of web applications.

10.5.2 Research Objective and Thesis Statement

Regarding the *research objective* established in Chapter 1 – to improve the manner in which CMS-based web applications are developed –, we consider that it is successfully accomplished by our research work. The reason for this statement is that the proposed development approach provides modeling facilities for both non-technical *and* technical stakeholders (the Business and System Designers, see Chapter 6), and it also allows them to collaboratively edit their respective models (without the possible loss of information that could result from mismatches between those models).

In turn, this proposal contrasts with the typical development process – which sees non-technical stakeholders defining the web application’s requirements and afterward relinquishing its evolution to developers, regaining it only when the software is tested and accepted –, because the latter makes the web application mostly dependent on the *developer’s interpretation* of the requirements, while the former allows the non-technical stakeholder to quickly intervene and make corrections to the web application’s implementation.

Following this same rationale, we consider that the research work presented in this dissertation proves our original *thesis statement* to be *true*.

Summary

This chapter has presented the validation efforts that were performed in the context of our research work. More specifically, it described the case studies for the validation of the CMS-ML, CMS-IL, and MYNK languages. An assessment of the initial research questions and objective (which were laid out in Chapter 1) was also performed, in order to determine whether the results of our research work did accomplish that objective. These case studies have demonstrated that these languages (and the proposed development approach) can be used to create CMS-based web application of a relative degree of complexity.

In the next chapter, we conclude this dissertation by recapitulating the proposed development approach, and the corresponding scientific contributions that are provided by this research work. We also take the opportunity to point out some future work topics that, although not fundamental for this dissertation, would improve the proposed approach and make it more adequate for a wider spectrum of development scenarios.

Chapter 11

Conclusion

Now this is not the end. It is not even the beginning of the end. But it is, perhaps, the end of the beginning.

SIR WINSTON CHURCHILL

The Internet has triggered a shift in the way most applications are developed and deployed. The developer now creates web applications, and to do so it is necessary to deal with different concepts, such as *request*, *cookie*, or *hyperlink*, instead of *window* or *widget*. Furthermore, the new deployment platform of choice is typically a web server or similar, instead of a desktop system. This has also paved the way for a variety of frameworks to support the development of web applications.

A particular kind of web application that has been gaining popularity is the Content Management System (CMS). This kind of application typically provides a reasonable degree of configurability and extensibility that system administrators and end-users tend to favor, as it mitigates the need to develop whole new software systems to support content distribution and/or user-specific tasks; instead, the system itself is already available, and customizations consist in the development of *components* (or *modules*, depending on the system's nomenclature) that will be installed on top of the CMS system. Thus, these systems can be considered both as web applications and as web application frameworks.

On the other hand, *Model-Driven Engineering (MDE)* approaches seek to facilitate the development process, by focusing on *models* and concepts instead of source code. More specifically, the MDE paradigm aims to achieve a higher level of abstraction by advocating the models should be the main artifacts in the software development process, while other artifacts (e.g., source code and documentation) can be obtained from those models in an automatic manner, through model transformations.

CMS systems have the potential for becoming the next standard of web application frameworks. They can also benefit from the advantages provided by the MDE paradigm, as these approaches can significantly accelerate the development and deployment of web applications and features, as well as simplify their maintenance.

In this dissertation, we have presented our proposal for a new MDE-oriented approach to address the development of CMS-web applications. This proposal is focused on the development of web applications that are based (and deployed) on CMS systems, and distinguishes itself from others because of its use of multiple modeling languages, as well as the use of a model synchronization mechanism, MYNK, to ensure that the different views of the system are all consistent with each other. The proposed approach is based on two CMS-oriented languages, CMS-ML and CMS-IL, that are situated at different levels of abstraction. CMS-ML addresses modeling in a high-level manner; on the other hand, CMS-IL uses low-level concepts (although the language is not specific to a particular CMS system) to provide a common ground for the building of higher-level languages, such as CMS-ML.

Using this approach, some stakeholders (the *Business Designers*, as they are called in Chapter 6 of this dissertation) can use the CMS-ML language to provide their own perspective on what the system should be like. When the various stakeholders have a view of the system with which they agree, developers (which in Chapter 6 are referred to as the *System Designers*) can refine the corresponding CMS-IL model, namely by adding or customizing features that could not be specified by the Business Designers (because of the lack of expressiveness of the CMS-ML language). After the CMS-IL model is satisfactory, it can then be deployed to a CMS system, either by (1) generating source code or (2) supplying the CMS-IL model as input to a CMS Model Interpreter component that will handle the runtime execution of the modeled web application. Although this approach does not remove the need for an iterative development process (namely in projects that involve the rapid development of prototypes to present to the application's end-users), it does provide a way to mitigate the additional work that is necessary for dealing with mismatches between the stakeholders' perspectives and the developer's understanding of those perspectives.

11.1 Contributions and Conclusions

Regarding the research work and the scientific contributions that have been presented in this dissertation, we were able to draw some conclusions, which are now mentioned.

We consider that the thesis statement (presented in Chapter 1) was successfully proven, as this dissertation proposes a development approach for CMS-based web applications

that not only addresses the needs of *multiple* kinds of stakeholders, but it also allows them to edit their web application models in a collaborative manner, and without the potential loss of information that would follow the occurrence of conflicts. We consider this to be an improvement over the typical development process, in which non-technical stakeholders would define the web application's requirements, and then relinquish control of its evolution to developers, regaining it only at the phase of software testing and (possible) acceptance. Thus, a developer's *misinterpretation* of a requirement could lead to a significant waste of effort and time, because it would become necessary to once again develop – or change – parts of the web application which should have already been addressed.

Besides the proposed development approach, this dissertation also provides a number of scientific contributions that are worth mentioning. This set of contributions begins with *the ReMMM metamodel* (which is derived from the MoMM metamodel [AK 01]). ReMMM provides a simple modeling language, consisting of elements – such as **Concept**, **Association**, and **Instantiation** – that enable the definition and representation of not only models but also *metamodels*, by applying the technique of Level Compaction [AK 05].

Another contribution is *the CMS-ML modeling language*, which allows the aforementioned Business Designers to actively participate in the design and development of the intended CMS-based web application. CMS-ML provides a set of concepts (such as **Task**, **Role**, or **WebSite**) that this kind of stakeholder is often familiarized with. Its inclusion in the proposed development approach also enables development scenarios of a more agile nature, in which the Business Designer can change or annotate the web application's model in a real-time manner and using non-technical terms.

On the other hand, *the CMS-IL language* endows System Designers with the set of computational- and CMS-oriented concepts that they need to perform their development tasks. It could be considered as a programming language for CMS systems, considering its usage of source code (platform-independent or otherwise), which makes it adequate for its target audience (developers with technological know-how regarding CMS systems and web application development). The facilities provided by CMS-IL are mainly derived from: (1) the functionalities provided by most CMS systems currently available; (2) imperative programming languages, such as C [KR 88], Ruby [FM 08], or Python [Lut 09]; and (3) functional languages, such as Lisp or Scheme [AS 96], from which it borrows fundamental concepts (e.g., **Lambda** or **Function**) that are also becoming quite popular in mainstream programming languages like Java 7 [Sch 11] or C# [NEG⁺ 10].

It should be mentioned that the CMS-ML and CMS-IL languages were obtained by following a small *set of guidelines*, which resulted from a compilation of related work (regarding metamodeling and language design) as well as of the results of our own experiences when developing the XIS2 modeling language [SSSM 07, SSF⁺ 07]. Although

simple in nature, these guidelines were essential to (1) quickly ensure that each language would have the elements necessary for its audience’s modeling tasks, while simultaneously (2) avoiding accidental complexity by gold plating the languages with a large number of unnecessary features.

Another relevant scientific contribution is that *CMS-ML* and *CMS-IL* are both extensible while adhering to the principle of *strict metamodeling* [AK 02, Küh 09]. The Toolkit extensibility mechanism that both languages provide, although a relatively simple solution in nature, illustrates an important advantage of using the ReMMM metamodel to define these languages. Furthermore, other ReMMM-based modeling languages can use this solution’s rationale to add further extensibility mechanisms and/or metalevels.

The MYNK model synchronization language, in turn, can be regarded as the “glue” between these CMS-oriented languages. Without MYNK, the changes between CMS-ML and CMS-IL models would have to be propagated manually, which would make the proposed multi-language approach impractical and error-prone. A MYNK synchronization consists of a set of **Sync Rules**, which take sets of **Changes** and map them to other sets of **Changes**; in other words, a **Rule** can be regarded as an answer to the following question: “if models M and N are equivalent, and M suffers some changes, how can model N be changed to become *again* equivalent to M ?”. This question is presented in a generic manner (and does not mention CMS-ML or CMS-IL), because MYNK only requires the usage of the ReMMM metamodel, which we believe is sufficiently expressive to model most current modeling languages.

We also consider that MYNK lays the groundwork for another avenue of research regarding model synchronization and transformations. The usage of model changes (instead of the model itself) as the driver for the transformation of models – a strategy which, to our knowledge, has not been addressed yet in the research field of MDE, although it is common practice among software developers that use RCS systems – has some advantages over other existing model transformation techniques (e.g., QVT, ATL), such as:

- It becomes easier to define new model synchronization rules, because the model synchronization designer only has to think in terms of relatively small sets of **Changes**. The typical alternative is to define the criteria that determines whether two entire models are equivalent to each other. Although the end result would ideally always be the same – one would be able to obtain two equivalent models M and N –, the process of defining model synchronization rules would be simpler. Furthermore, defining simpler rules could avoid potential design flaws derived from the rules’ complexity;
- It avoids the loss of data in models that are being synchronized, because such situations can be considered as conflicts that must be manually resolved by the

person who is performing the synchronization (unlike what happens with QVT’s `enforce` mode, which changes a target model to become equivalent to the source model); and

- It effectively solves the problem of models becoming inconsistent when the corresponding metamodel is changed (e.g., when a concept is added or removed). This becomes a relatively easy issue to address, by defining a set of model changes for each metamodel change (see the beginning of Chapter 3 for further details).

Nevertheless, we still regard this development approach – and all of its supporting languages – as a work-in-progress. Although these languages have reached a state in which they could be used to develop web applications with a considerable degree of complexity, there is certainly still room for improvement on a number of issues, which we discuss in the following section.

11.2 Future Work

Although the work presented in this dissertation has yielded some significant research results, these topics still present avenues of research that are worthy of further pursuit. In this section, we highlight some possible research topics that we consider to be relevant for the work presented in this dissertation.

One of the most obvious issues to be addressed are the concrete syntaxes of the CMS-ML, CMS-IL, and MYNK languages. Although the concrete syntaxes presented in this dissertation are already defined while taking into consideration some of the provided user feedback, no language can hope to survive in the long term without evolving [KP 09]. Thus, it would be important to conduct further experiments with a statistically-significant segment of the languages’ target audiences, in order to ensure that each language’s audience can read, understand, and modify models in a satisfactory manner.

On the other hand, the languages’ abstract syntaxes should also be the subject of further refinement efforts, as we do not exclude the possibility that some modeling elements (e.g., CMS-ML’s `Bindings` and `Operations`) may ultimately be unnecessary for the modeling tasks that the stakeholders (Business or System Designers) will perform during the web application’s development. However, such changes should only be performed after careful evaluation of stakeholder feedback regarding those elements, to ensure that they are not actually the result of *essential complexity*, but rather of *accidental complexity* [Bro 87, AK 08] – which should be kept to a minimum.

Another avenue of research for CMS-ML and CMS-IL would be the modeling of `WebComponents` supported by RIA (Rich Internet Application) technologies. Nevertheless, the definition of such models would also require that CMS systems provide some degree

of support for RIA frameworks (such as Apache Flex¹ or Microsoft Silverlight²) and the elements that they offer. To the best of our knowledge, research efforts regarding this topic are scarce at the time of writing of this dissertation, although we have found some proposals [PLCS 07, MFR 09] for RIA-oriented extensions to the WebML and UWE modeling languages.

An interesting development path would be to further extend and refine CMS-IL so that it can effectively be used as a common ground (or middle language) for the deployment of CMS-based web applications, in the manner suggested in Figure 11.1. From this perspective, CMS-IL could be regarded as analogous to Java bytecode³ or Microsoft’s Common Intermediate Language (CIL)⁴, which are the cross-platform languages that respectively support the Java and .NET frameworks.

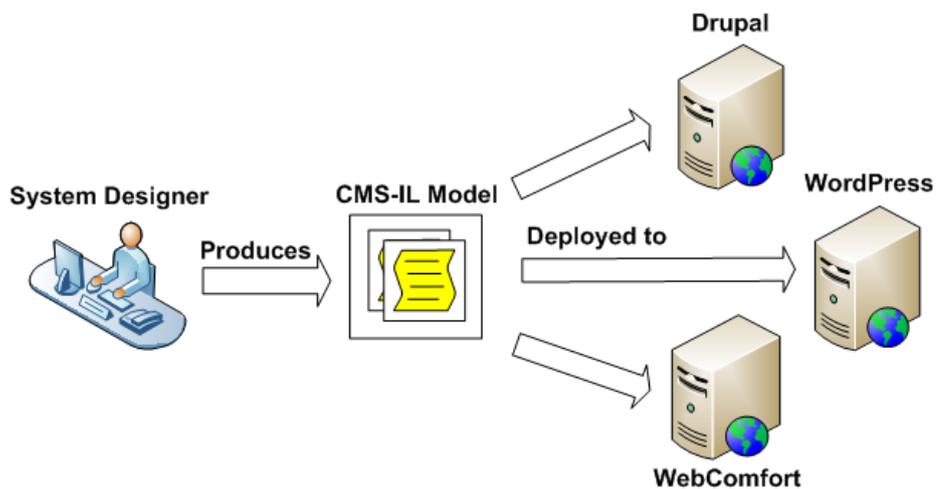


Figure 11.1: Using a CMS-IL model to deploy a web application in a variety of CMS systems.

The addition of further features to CMS-IL should also be considered. In particular, the various kinds of programmable hooks that are typically available in CMS systems (e.g., to determine what configuration options should be presented to users, or to change the HTML output that will be sent to the user’s web browser) could easily be reflected in CMS-IL as *Variability Point Types* and *CMS Events*, for which the CMS-IL Toolkit Developer could then provide *Event Handlers* that would extend those hooks with additional behavior. In fact, this is the manner in which most CMS systems are extended by third-parties.

¹ *Apache Flex*, <<http://incubator.apache.org/flex>> (accessed on May 22nd, 2012)

² *Microsoft Silverlight*, <<http://www.silverlight.net>> (accessed on May 22nd, 2012)

³ Oracle, “Java Virtual Machine Specification (Java SE 7 Edition)”, <<http://docs.oracle.com/javase/7/specs/jvms/se7/html/index.html>> (accessed on May 21st, 2012)

⁴ “Standard ECMA-335: Common Language Infrastructure (CLI), 5th edition”, <<http://www.ecma-international.org/publications/standards/Ecma-335.htm>> (accessed on May 21st, 2012)

Regarding MYNK, a worthwhile extension to consider would be the possibility of the synchronization developer including *semantic constraints* – rules that a certain model must *always* obey – within a **Synchronization Spec**. An example of a typical constraint could be that the same CMS-IL Toolkit cannot define two Domain **Entities** with the same **Name**; in other words, there should not be two **Model Elements** that (1) are included in the same **Toolkit**, (2) are connected to the **Entity** concept via the **Instantiation** relationship, and (3) whose value of the **Name Attribute** is equal. A possible way to address this feature would be to use OCL [OMG 10] to define such constraints, as ReMMM models could also be represented using MOF or UML (thus enabling the use of OCL for this purpose).

Finally, it would be desirable to extend the proposed web application development approach with languages other than CMS-ML and CMS-IL. Although the stakeholder types considered by our approach – Business Designer and System Designer – are broad enough to encompass most (if not all) of the stakeholders that participate in the design and development of a web application, the fact remains that this division into two categories is a simplification which we consider may not be adequate in some cases of greater complexity. The way to address this would be consider further kinds of stakeholder, and thus additional languages that support their modeling needs; however, this would also require that additional MYNK **Synchronization Specs** be defined, so that the models of all these languages can be kept in sync with each other.

Figure 11.2 provides a tentative illustration of this extended approach scenario, in which the web application’s various stakeholders use a set of languages – not only CMS-ML and CMS-IL, but also other languages that they find adequate for their modeling purposes, with possibly overlapping concepts (e.g., CMS-IL’s **Entity** concept is nearly equivalent to CMS-ML’s) – to model their own perspectives of the desired system. Each of those models will then reflect the corresponding stakeholder’s current perspective of the system, so that further changes can be made if the current system is not yet aligned with its goals.

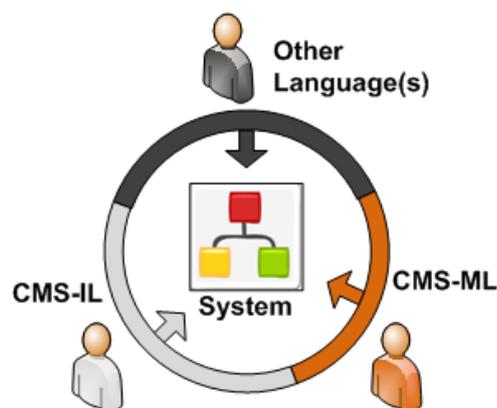


Figure 11.2: Proposed development approach, extended with additional languages.

Glossary

A

Abstract Syntax The set of concepts, and relationships between them, that reflect the structure of the information in a language's domain. It provides a syntax-independent definition of the relevant domain information that should be stored by the language's models, p. 31.

Application Programming Interface (API) An interface that a software program or library makes available, so that other software may interact with it. It determines the vocabulary and calling conventions that a programmer must use in order to use the functionality provided by the program or library, p. 5.

ATL Transformation Language (ATL) A model transformation language, developed and maintained by the AtlanMod research team, that provides a more natural approach than OMG's QVT. Available as a plugin for Eclipse's Model To Model (M2M) project. While QVT requires that the source and target metamodels conform to the MOF metamodel, ATL requires conformance to the *ECore* metamodel, p. 57.

C

Computer-Aided Software Engineering (CASE) The usage of computer-assisted tools and methods in the context of software development. Aims to improve the software development process by establishing a common ground (accessible to any particular software development role, such as programmers or managers) for the analysis of a software development project's status at any particular time, p. 20.

Content Management System (CMS) A system for managing content and providing it in various formats (such as documents, videos, or photos). Some authors distinguish explicitly between different kinds of CMS (e.g., Document Management

System, Enterprise Content Management System, Web Content Management System). Still, we use this term to refer to web-oriented content management systems, as we consider that these systems can address all of those other kinds of content management aspects, p. 83.

CMS Intermediate Language (CMS-IL) A textual modeling language for defining CMS-based web applications in a low-level but platform-independent manner. Similar to CMS-ML, especially regarding web application structural aspects, but behavioral aspects are specified using programming language-like elements, p. 146.

CMS Modeling Language (CMS-ML) A graphical modeling language that allows stakeholders to model CMS-based web applications in a high-level and platform-independent manner. Features two kinds of models, WebSite Templates and Toolkits, at different modeling levels, each of which with their own specific set of views. In order to be easy to learn, it does not define many elements, and so it is not as expressive as CMS-IL. It can, however, be synchronized with CMS-IL by means of the MYNK model synchronization mechanism, p. 105.

Concrete Syntax The set of visual symbols (either graphical or textual) that can be used to represent domain information in the language's models. Unlike the abstract syntax, a language can define multiple concrete syntaxes, in order to support multiple representations of the same model. Also known as *notation*, p. 32.

D

Domain-Specific Language (DSL) A language that endows its users with the ability of using problem-domain concepts to create models. Typically designed to be useful only for a specific set of tasks, as opposed to a general-purpose language, p. 28.

Domain-Specific Modeling (DSM) The usage of problem-domain concepts as the basic building blocks of models (as opposed to CASE tools which typically use programming language concepts), p. 28.

E

Enterprise Content Management (ECM) A system that uses Internet-based technologies and workflows to address the capture, management, preservation, and delivery of contents related to an organization and its processes, p. 4.

M

Model-Driven Architecture (MDA) The OMG approach to address the software development life cycle, based on OMG standards like UML, MOF, QVT, and XMI. It is focused on the importance of models in the development process, and defines two different kinds of model: the Platform-Independent Model (PIM) and the Platform-Specific Model (PSM), p. 27.

Model-Driven Engineering (MDE) A software development methodology which advocates the usage of models as the most important artifacts, rather than source code. It is meant to increase productivity by leaving repetitive tasks to automated mechanisms (such as model-to-model transformations and source code generation), and by promoting communication between individuals and teams working on the system, p. 20.

Metalevel Same as *Metalevel* or *modeling level*. The position of a model, in the context of a hierarchical metamodeling architecture, in relation to other models. Defining a set of metalevels provides a way to hierarchically structure a set of models in relation to their respective metamodels. A metamodel is often said to be in a metalevel above the corresponding model's metalevel, p. 21.

Metamodel A description of a relevant abstraction over the real world, through the precise definition of the constructs and rules that are needed for creating models. A metamodel provides a language that can be used to create a model. Likewise, a metamodel that defines the language in which another metamodel is specified is itself called a *metametamodel*, p. 21.

Microsoft Sketchflow (Sketchflow) A visual user interface builder used to create prototypes for Windows Presentation Foundation and Silverlight applications. The created prototypes are themselves valid applications, and so developers would only need to add business logic to the application. Included in Microsoft Expression Blend 3, p. 310.

Migration A strategy, supported by some ORM frameworks, that can be used to address changes to data and to the structure of data (depending on the ORM's

implementation). Typically consists of combining the ActiveRecord and the Command design patterns. Involves the definition of classes that encapsulate database transformation operations, by providing Execute and Undo methods, p. 64.

Model An interpretation of a certain problem-domain (i.e., a relevant fragment, the subject of modeling and system development tasks, of the real world) according to a determined structure of concepts, its metamodel, p. 21.

Model Synchronization A process that receives two (or more) models – which were semantically equivalent at some point – and their respective histories, and further manipulates them so that they become semantically equivalent again (i.e., so that they are in sync again). Presents the advantage of supporting scenarios in which *both* of the provided models have been changed, p. 93.

Model Transformation A process that receives a *source* model as input and outputs another artifact (which may, or may not, be semantically equivalent to the source model). Typical examples are source code generation mechanisms (such as MOFM2T, which generates source code text files from a source model) and model-to-model transformations (such as QVT, in which the process results in the production of another model). Some model transformation frameworks also consider the target artifact/model to be an input: in such cases, this target artifact is often manipulated to become semantically equivalent to the source model, p. 47.

Meta Object Facility (MOF) An OMG standard for the definition of metamodels. It consists of a reflexive language (i.e., with the capability to define itself) that can be used to create further modeling languages, typically domain-specific languages. Such DSLs can then be used in conjunction with other OMG facilities, namely model-to-model transformation and text generation mechanisms such as QVT and MOFM2T (respectively), p. 24.

MOF Model To Text Transformation Language (MOFM2T) A textual language, defined by the OMG, that can be used to specify transformations of MOF-derived models to text. It uses a template-based mechanism that is very similar to those found in typical source code generation tools. Also known as *Mof2Text*, p. 55.

Model sYNchronization framework (MYNK) A textual language that enables the specification of model synchronizations between CMS-ML models and CMS-IL

models. The MYNK mechanism is the cornerstone of the proposed CMS-based web application development approach (proposed in this dissertation), as it allows the future extension of the approach with further CMS-oriented languages, p. 203.

O

Object Constraint Language (OCL) An OMG standard, and also a component of the QVT model transformation language. OCL consists of a textual declarative language that can be used to specify object constraints and queries over MOF-based models, p. 25.

Object-Oriented Programming (OOP) A programming paradigm in which developers define data structures, called *objects*, and their behavior, as operations called *methods*. It is also possible to establish relationships between objects, such as inheritance (an object inherits characteristics from another object) or referencing (an object knows about another object, and can thus interact with it). This paradigm also provides some features such as encapsulation, modularity, polymorphism, and inheritance, p. 22.

Object-Relational Mapping (ORM) A framework that aims to address the mismatch between OOP concepts and the relational model, by establishing mappings between the two domains. Besides changes to data, some ORM frameworks also consider changes to the data's structure (typically by means of a Migration mechanism), p. 63.

OutSystems Agile Platform (Agile Platform) A product which addresses the full life cycle of delivering and managing web business applications. It includes the tools required to integrate, develop, deploy, manage, and change web business applications in a straightforward manner, p. 307.

Q

Query/View/Transformation (QVT) An OMG model-to-model transformation standard that is one of the cornerstones of the MDA approach. It consists of a set of languages – QVTRelation, QVTOperational, and QVTCore – that enable the specification of model-to-model transformations as sets of mappings between different MOF-based languages, p. 49.

R

Revision Control System (RCS) An application that supports the storage and tracking of the various changes to a set of files. To do so, it provides concepts such as *Repository*, *Working Copy*, *File*, and *Delta*. Also known as source control management or version control system, p. 66.

Revised Metamodel for Multiple Metalevels (ReMMM) A small metamodel, defined with the objective of being able to express the elements which constitute any model *and* its respective metamodel. Provides basic concepts such as *Concept*, *Association*, *Generalization*, and *Instantiation*. Used as the metamodel for CMS-ML and CMS-IL; MYNK also assumes the existence of the concepts defined by ReMMM. ReMMM is derived from the MoMM metamodel [AK 01], p. 97.

Rich Internet Application (RIA) Web applications that have most of the characteristics of desktop applications, typically delivered either via a standards-based web browser (through a plugin), or independently via sandboxes or virtual machines, p. 80.

S

Standard Query Language (SQL) A textual language that is used to access and manipulate relational databases (repositories of data structured according to the relational model). It provides concepts – such as *Database*, *Table*, *Column*, *Row*, and *Schema* – that allow developers to manipulate not only the data, but also the data's structure, p. 61.

U

Unified Modeling Language (UML) A general-purpose modeling language, originally designed to specify, visualize, construct, and document information systems, p. 22.

Universal Resource Locator (URL) An address through which users can access a specific website or document on the Internet, p. 85.

UML-based Web Engineering (UWE) A software engineering approach, based on OMG standards, for the development of web applications. It follows the separation of concerns principle, by building separate models for Requirements, Content, Navigation, Presentation, and Process, p. 301.

W

Web Application An application, supported by technologies such as the Internet, that can be accessed and used via a web browser. A web application typically consists of a client (web browser) coded in Dynamic HTML (DHTML) – namely HTML, Javascript, and CSS –, and may also include a server-side component that receives client requests and returns a corresponding response, p. 17.

Web Modeling Language (WebML) A visual language and software engineering approach for designing data-intensive web applications. It considers four different kinds of model, developed in an iterative manner: Data, Derivation, Hypertext, and Presentation, p. 297.

X

eXtreme modeling for Interactive Systems (XIS2) A UML profile oriented toward the development of interactive software systems. Its main goal is to allow the modeling of the various aspects of an interactive software system. It considers six different kinds of models: Domain, BusinessEntities, Actors, UseCases, InteractionSpace, and Navigation, p. 304.

XML Metadata Interchange (XMI) A standard for the exchange, definition, interchange, and manipulation of metadata information by using XML. Commonly used as a format to exchange UML models between tools, p. 25.

eXtensible Markup Language (XML) A markup language, defined by the W3C (World Wide Web Consortium), that is meant to encode information in a format that is readable by both humans and machines, p. 24.

References

- [AK 00] Atkinson, Colin and Kühne, Thomas. Meta-level Independent Modelling. In *International Workshop on Model Engineering at the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, pages 12–16. June 2000. [online] Available at: <http://homepages.mcs.vuw.ac.nz/~tk/publications/papers/level-indep.pdf> [Accessed March 27, 2012].
- [AK 01] Atkinson, Colin and Kühne, Thomas. The Essence of Multilevel Meta-modeling. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language, Modeling Languages, Concepts, and Tools, Fourth International Conference, Proceedings*, volume 2185 of *Lecture Notes in Computer Science*, pages 19–33. Springer, October 2001. ISBN 978-3540426677. doi:10.1007/3-540-45441-1_3.
- [AK 02] Atkinson, Colin and Kühne, Thomas. Profiles in a Strict Metamodeling Framework. *Science of Computer Programming*, 44(1):5–22, July 2002. ISSN 0167-6423. doi:10.1016/S0167-6423(02)00029-1.
- [AK 03] Atkinson, Colin and Kühne, Thomas. Model-Driven Development: A Metamodeling Foundation. *IEEE Software*, 20(5):36–41, September/October 2003. ISSN 0740-7459. doi:10.1109/MS.2003.1231149.
- [AK 05] Atkinson, Colin and Kühne, Thomas. Concepts for Comparing Modeling Tool Architectures. In Lionel Briand and Clay Williams, editors, *Model Driven Engineering Languages and Systems: 8th International Conference, MoDELS 2005*, volume 3713 of *Lecture Notes in Computer Science*, pages 398–413. Springer Berlin/Heidelberg, October 2005. ISBN 978-3540290100. ISSN 0302-9743. doi:10.1007/11557432_30.
- [AK 08] Atkinson, Colin and Kühne, Thomas. Reducing accidental complexity in domain models. *Software and Systems Modeling*, 7(3):345–359, July 2008. ISSN 1619-1366. doi:10.1007/s10270-007-0061-0.

- [ALC 08] Amar, Bastien, Leblanc, Hervé, and Coulette, Bernard. A Traceability Engine Dedicated to Model Transformation for Software Engineering. In Jon Oldevik, Gøran K. Olsen, Tor Neple, and Richard Paige, editors, *ECMDA Traceability Workshop Proceedings (ECMDA-TW 2008)*, pages 7–16. SINTEF, June 2008. ISBN 978-8214043969.
- [AS 96] Abelson, Harold and Sussman, Gerald Jay. *Structure and Interpretation of Computer Programs*. The MIT Press, second edition, July 1996. ISBN 978-0262011532.
- [BAGB 11] Barišić, Ankica, Amaral, Vasco, Goulão, Miguel, and Barroca, Bruno. Quality in Use of Domain Specific Languages: a Case Study. In Craig Anslow, Shane Markstrum, and Emerson Murphy-Hill, editors, *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2011)*, co-located with *SPLASH 2011*, pages 65–72. ACM, New York, NY, USA, October 2011. ISBN 978-1450310246. doi:10.1145/2089155.2089170.
- [BAGB 12] Barišić, Ankica, Amaral, Vasco, Goulão, Miguel, and Barroca, Bruno. Evaluating the Usability of Domain-Specific Languages. In Marjan Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, pages 386–407. Information Science Reference, September 2012. ISBN 978-1466620926. doi:10.4018/978-1-4666-2092-6.ch014.
- [BBH⁺ 08] Byron, Angela, Berry, Addison, Haug, Nathan, Eaton, Jeff, Walker, James, and Robbins, Jeff. *Using Drupal*. O’Reilly Media, December 2008. ISBN 978-0596515805.
- [BCFM 07] Brambilla, Marco, Comai, Sara, Fraternali, Piero, and Matera, Maristella. Designing Web Applications with WebML and WebRatio. In Gustavo Rossi, Oscar Pastor, Daniel Schwabe, and Luis Olsina, editors, *Web Engineering: Modelling and Implementing Web Applications*, Human-Computer Interaction Series, pages 221–261. Springer London, October 2007. ISBN 978-1846289231. doi:10.1007/978-1-84628-923-1_9.
- [BMA⁺ 12] Barišić, Ankica, Monteiro, Pedro, Amaral, Vasco, Goulão, Miguel, and Monteiro, Miguel. Patterns for Evaluating Usability of Domain-Specific Languages. In *Proceedings of the Pattern Languages of Programs Con-*

-
- ference (PLoP 2012), co-located with SPLASH 2012*. ACM, New York, NY, USA, October 2012.
- [Boi 01] Boiko, Bob. *Content Management Bible*. John Wiley & Sons, Hoboken, New Jersey, U.S.A., December 2001. ISBN 978-0764548628.
- [Bro 87] Brooks, Jr., Frederick P. No Silver Bullet – Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987. ISSN 0018-9162. doi:10.1109/MC.1987.1663532.
- [Car 06] Carmo, João Leonardo Vicente do. *Web Content Management Systems: Experiences and Evaluations with the WebComfort Framework*. Master’s thesis, Instituto Superior Técnico, Portugal, December 2006.
- [CFB⁺ 02] Ceri, Stefano, Fraternali, Piero, Bongio, Aldo, Brambilla, Marco, Comai, Sara, and Matera, Maristella. *Designing Data-Intensive Web Applications*. Morgan Kaufmann, December 2002. ISBN 978-1558608436.
- [CJKW 07] Cook, Steve, Jones, Gareth, Kent, Stuart, and Wills, Alan. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, June 2007. ISBN 978-0321398208.
- [CN 91] Cox, Brad J. and Novobilski, Andrew J. *Object-Oriented Programming: An Evolutionary Approach*. Addison-Wesley, second edition, 1991. ISBN 978-0201548341.
- [CRR 09] Cao, Lan, Ramesh, Balasubramaniam, and Rossi, Matti. Are Domain-Specific Models Easier to Maintain Than UML Models? *IEEE Computer*, 26(4):19–21, 2009. ISSN 0740-7459. doi:10.1109/MS.2009.87.
- [CS 07] Costa, Marco and Silva, Alberto Rodrigues da. RT-MDD Framework – A Practical Approach. In Jon Oldevik, Gøran K. Olsen, and Tor Neple, editors, *ECMDA Traceability Workshop Proceedings (ECMDA-TW 2007)*, pages 17–26. SINTEF, June 2007. ISBN 978-8214040562.
- [DGF 05] Ducasse, Stéphane, Gîrba, Tudor, and Favre, Jean-Marie. Modeling Software Evolution by Treating History as a First Class Entity. *Electronic Notes in Theoretical Computer Science*, 137(3):75–86, September 2005. ISSN 1571-0661. doi:10.1016/j.entcs.2004.08.035.
- [Dis 05_a] Diskin, Zinovy. Mathematics of Generic Specifications for Model Management, I. In Laura C. Rivero, Jorge Horacio Doorn, and Viviana E.
-

- Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*, pages 351–358. Idea Group, 2005. ISBN 978-1591405603.
- [Dis 05_b] Diskin, Zinovy. Mathematics of Generic Specifications for Model Management, II. In Laura C. Rivero, Jorge Horacio Doorn, and Viviana E. Ferraggine, editors, *Encyclopedia of Database Technologies and Applications*, pages 359–365. Idea Group, 2005. ISBN 978-1591405603.
- [DPFK 08] Drivalos, Nicholas, Paige, Richard F., Fernandes, Kiran J., and Kolovos, Dimitrios S. Towards Rigorously Defined Model-to-Model Traceability. In Jon Oldevik, Gøran K. Olsen, Tor Neple, and Richard Paige, editors, *ECMDA Traceability Workshop Proceedings (ECMDA-TW 2008)*, pages 17–26. SINTEF, June 2008. ISBN 978-8214043969.
- [EA 06] Essalmi, Fathi and Ayed, Leila Jemni Ben. Graphical UML View from Extended Backus-Naur Form Grammars. In *Proceedings of the 6th IEEE International Conference on Advanced Learning Technologies (ICALT 2006)*, pages 544–546. IEEE Computer Society, Los Alamitos, CA, USA, July 2006. ISBN 0769526322. doi:10.1109/ICALT.2006.178.
- [EEKR 99] Ehrig, Hartmut, Engels, Gregor, Kreowski, Hans-Jörg, and Rozenberg, Grzegorz, editors. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 2: Applications, Languages and Tools*. World Scientific Pub. Co. Inc., 1999. ISBN 978-9810240202.
- [Fav 04_a] Favre, Jean-Marie. Foundations of Meta-Pyramids: Languages vs. Meta-models – Episode II: Story of Thotus the Baboon. In Jean Bézivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, March 2004. ISSN 1862-4405.
- [Fav 04_b] Favre, Jean-Marie. Foundations of Model (Driven) (Reverse) Engineering: Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In Jean Bézivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, volume 04101 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, March 2004. ISSN 1862-4405.

-
- [Fav 05] Favre, Jean-Marie. Megamodeling and Etymology. In James R. Cordy, Ralf Lämmel, and Andreas Winter, editors, *Transformation Techniques in Software Engineering*, volume 05161 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany, April 2005. ISSN 1862-4405.
- [FGDS 06] France, Robert B., Ghosh, Sudipto, Dinh-Trong, Trung T., and Solberg, Arnor. Model-Driven Development Using UML 2.0: Promises and Pitfalls. *IEEE Computer*, 39(2):59–66, February 2006. ISSN 0018-9162. doi:10.1109/MC.2006.65.
- [Fia 04] Fiadeiro, José L. *Categories for Software Engineering*. Springer, November 2004. ISBN 978-3540209096.
- [FM 08] Flanagan, David and Matsumoto, Yukihiro. *The Ruby Programming Language*. O'Reilly Media, February 2008. ISBN 978-0596516178.
- [Fow 03] Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003. ISBN 978-0321127426.
- [GFD 05] Gîrba, Tudor, Favre, Jean-Marie, and Ducasse, Stéphane. Using Meta-Model Transformation to Model Software Evolution. *Electronic Notes in Theoretical Computer Science*, 137(3):57–64, September 2005. ISSN 1571-0661. doi:10.1016/j.entcs.2005.07.005.
- [GGA 10] Gabriel, Pedro, Goulão, Miguel, and Amaral, Vasco. Do Software Languages Engineers Evaluate their Languages? In Xavier Franch, Itana Gimenes, and Juan Pablo Carvallo, editors, *Proceedings of the XIII Iberoamerican Congress on Software Engineering (CIbSE 2010)*, pages 149–162. April 2010.
- [GHJV 95] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 978-0201633610.
- [Gho 11] Ghosh, Debasish. DSL for the Uninitiated. *Communications of the ACM*, 54(7):44–50, July 2011. ISSN 0001-0782. doi:10.1145/1965724.1965740.
- [GR 03] Gerber, Anna and Raymond, Kerry. MOF to EMF: There and Back Again. In Michael G. Burke, editor, *Proceedings of the 2003 OOPSLA*
-

- Workshop on Eclipse Technology eXchange*, pages 60–64. ACM Press, New York, NY, USA, October 2003. doi:10.1145/965660.965673.
- [GCK 04] Greenfield, Jack, Short, Keith, Cook, Steve, and Kent, Stuart. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004. ISBN 978-0471202844.
- [GSSF 04] Gaffar, Ashraf, Sinnig, Daniel, Seffah, Ahmed, and Forbrig, Peter. Modeling Patterns for Task Models. In Pavel Slavík and Philippe A. Palanque, editors, *Proceedings of the 3rd International Workshop on Task Models and Diagrams for User Interface Design (TAMODIA 2004)*, pages 99–104. ACM, New York, NY, USA, 2004. ISBN 1595930000. doi:10.1145/1045446.1045465.
- [GW 09] Giese, Holger and Wagner, Robert. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, 8:21–43, 2009. ISSN 1619-1366. doi:10.1007/s10270-008-0089-9.
- [Har 07] Harshbarger, Bill. Chaos, Complexity and Language Learning. *Language Research Bulletin*, 22, 2007. [online] Available at: <http://web.icu.ac.jp/lrb/vol_22/Harshbarger%20LRB%20V22.pdf> [Accessed December 31, 2010].
- [Hen 05] Henderson-Sellers, Brian. UML – the Good, the Bad or the Ugly? Perspectives from a panel of experts. *Software and Systems Modeling*, 4(1):4–13, February 2005. ISSN 1619-1366. doi:10.1007/s10270-004-0076-8.
- [HP 05] Haustein, Stefan and Pleumann, Jörg. A model-driven runtime environment for Web applications. *Software and System Modeling*, 4(4):443–458, 2005. doi:10.1007/s10270-005-0093-2.
- [HRW 09] Hammond, Christopher J., Renner, Patrick, and Walker, Shaun. *DotNet-Nuke 5 User’s Guide: Get Your Website Up and Running*. Wrox, June 2009. ISBN 978-0470462577.
- [Hug 90] Hughes, John. Why Functional Programming Matters. In David A. Turner, editor, *Research Topics in Functional Programming (The UT year of programming series)*, pages 17–42. Addison-Wesley Pub (Sd), June 1990. ISBN 978-0201172362.

-
- [IEEE 90] IEEE. IEEE Std 610.12-1990: IEEE Standard Glossary of Software Engineering Terminology, September 1990. doi:10.1109/IEEESTD.1990.101064.
- [IEEE 00] IEEE. IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, September 2000. doi:10.1109/IEEESTD.2000.91944.
- [JB 06] Jouault, Frédéric and Bézivin, Jean. KM3: A DSL for Metamodel Specification. In Roberto Gorrieri and Heike Wehrheim, editors, *Formal Methods for Open Object-Based Distributed Systems, 8th IFIP WG 6.1 International Conference Proceedings (FMOODS 2006)*, volume 4037 of *Lecture Notes in Computer Science*, pages 171–185. Springer Berlin/Heidelberg, June 2006. ISBN 978-3540348931. doi:10.1007/11768869_14.
- [Jen 05] Jenkins, Tom. *Enterprise Content Management Solutions: What You Need to Know*. Open Text Corporation, April 2005. ISBN 978-0973066265.
- [KK 02] Koch, Nora and Kraus, Andreas. The Expressive Power of UML-based Web Engineering. In *Proceedings of the Second International Workshop on Web-Oriented Software Technology (IWWOST'2002)*. June 2002.
- [KK 08] Kroiß, Christian and Koch, Nora. UWE Metamodel and Profile: User Guide and Reference. Technical Report 0802, Ludwig-Maximilians-Universität, February 2008. [online] Available at: <<http://uwe.pst.ifi.lmu.de/download/UWE-Metamodel-Reference.pdf>> [Accessed April 28, 2011].
- [KKH 01] Koch, Nora, Kraus, Andreas, and Hennicker, Rolf. The Authoring Process of the UML-based Web Engineering Approach. In *Proceedings of the First International Workshop on Web-Oriented Software Technology (IWWOST'2001)*. June 2001.
- [KP 09] Kelly, Steven and Pohjonen, Risto. Worst Practices for Domain-Specific Modeling. *IEEE Software*, 26(4):22–29, 2009. ISSN 0740-7459. doi:10.1109/MS.2009.109.
- [KR 88] Kernighan, Brian W. and Ritchie, Dennis M. *The C Programming Language*. Prentice Hall, second edition, April 1988. ISBN 978-0131103627.
-

REFERENCES

- [KT 08] Kelly, Steven and Tolvanen, Juha-Pekka. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, Hoboken, New Jersey, U.S.A., March 2008. ISBN 978-0470036662.
- [Küh 09] Kühne, Thomas. Contrasting Classification with Generalisation. In Markus Kirchberg and Sebastian Link, editors, *Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modelling (APCCM 2009)*, volume 96 of *CRPIT*, pages 71–78. Australian Computer Society, January 2009. ISBN 978-1920682774.
- [KWB 03] Kleppe, Anneke, Warmer, Jos, and Bast, Wim. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Reading, Massachusetts, U.S.A., April 2003. ISBN 978-0321194428.
- [Lew 46] Lewin, Kurt. Action Research and Minority Problems. *Journal of Social Issues*, 2(4):34–46, November 1946. ISSN 1540-4560. doi:10.1111/j.1540-4560.1946.tb02295.x.
- [LMB 92] Levine, John R., Mason, Tony, and Brown, Doug. *lex & yacc*. O’Reilly & Associates, Inc., second edition, 1992. ISBN 978-1565920002.
- [LTM 06] Lerdorf, Rasmus, Tatroe, Kevin, and MacIntyre, Peter. *Programming PHP*. O’Reilly Media, second edition, May 2006. ISBN 978-0596006815.
- [Lut 09] Lutz, Mark. *Learning Python*. O’Reilly Media, fourth edition, October 2009. ISBN 978-0596158064.
- [MCF 03] Mellor, Stephen J., Clark, Anthony N., and Futagami, Takao. Guest Editors’ Introduction: Model-Driven Development. *IEEE Software*, 20(5):14–18, September 2003. ISSN 0740-7459. doi:10.1109/MS.2003.1231145.
- [Mel 04] Mellor, Stephen J. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley Professional, March 2004. ISBN 978-0201788914.
- [Mey 06] Meyer, Eric A. *CSS: The Definitive Guide*. O’Reilly Media, third edition, November 2006. ISBN 978-0596527334.
- [MF 10] MacDonald, Matthew and Freeman, Adam. *Pro ASP.NET 4 in C# 2010*. Apress, June 2010. ISBN 978-1430225294.

-
- [MFR 09] Machado, Leonardo, Filho, Orlando, and Ribeiro, João. UWE-R: an extension to a web engineering methodology for rich internet applications. *WSEAS Transactions on Information Science and Applications*, 6(4):601–610, April 2009. ISSN 1790-0832.
- [MFV 06] Moreno, Nathalie, Fraternali, Piero, and Vallecillo, Antonio. A UML 2.0 profile for WebML modeling. In Nora Koch and Luis Olsina, editors, *Workshop Proceedings of the 6th International Conference on Web Engineering (ICWE 2006)*, volume 155 of *ACM International Conference Proceeding Series*. ACM, New York, NY, USA, July 2006. ISBN 1595934359. doi:10.1145/1149993.1149998.
- [MHS 05] Mernik, Marjan, Heering, Jan, and Sloane, Anthony M. When and How to Develop Domain-Specific Languages. *ACM Computing Surveys*, 37(4):316–344, December 2005. ISSN 0360-0300. doi:10.1145/1118890.1118892.
- [MPS 02] Mori, Giulio, Paternò, Fabio, and Santoro, Carmen. CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. *IEEE Transactions on Software Engineering*, 28(8):797–813, August 2002. ISSN 0098-5589. doi:10.1109/TSE.2002.1027801.
- [MV 11] Meyers, Bart and Vangheluwe, Hans. A framework for evolution of modelling languages. *Science of Computer Programming*, 76(12):1223–1246, December 2011. ISSN 0167-6423. doi:10.1016/j.scico.2011.01.002.
- [MWCS 11] Meyers, Bart, Wimmer, Manuel, Cicchetti, Antonio, and Sprinkle, Jonathan. A generic in-place transformation-based approach to structured model co-evolution. *Electronic Communications of the European Association of Software Science and Technology (ECEASST)*, 42, April 2011. ISSN 1863-2122.
- [NEG⁺ 10] Nagel, Christian, Evjen, Bill, Glynn, Jay, Watson, Karli, and Skinner, Morgan. *Professional C# 4.0 and .NET 4*. Wrox, March 2010. ISBN 978-0470502259.
- [NNC 06] Nóbrega, Leonel, Nunes, Nuno Jardim, and Coelho, Helder. The Meta Sketch Editor: a Reflexive Modeling Editor. In Gaëlle Calvary, Costin Pribeanu, Giuseppe Santucci, and Jean Vanderdonckt, editors, *Computer-Aided Design of User Interfaces V – Proceedings of the 6th International*
-

Conference on Computer-Aided Design of User Interfaces (CADUI 2006), pages 201–214. Springer Verlag, Berlin, Germany, June 2006. ISBN 978-1402058196. doi:10.1007/978-1-4020-5820-2_17.

- [OMG 02] OMG. Object Management Group – Common Warehouse Metamodel (CWM) Specification, Version 1.1. [online] Available at: <<http://www.omg.org/spec/CWM/1.1/PDF>> [Accessed November 14, 2011], March 2002.
- [OMG 03] OMG. Object Management Group – MDA Guide Version 1.0.1, June 2003. [online] Available at: <<http://www.omg.org/cgi-bin/doc?omg/03-06-01>> [Accessed November 14, 2011].
- [OMG 08] OMG. Object Management Group – MOF Model to Text Transformation Language (MOFM2T), v1.0, January 2008. [online] Available at: <<http://www.omg.org/spec/MOFM2T/1.0/PDF>> [Accessed November 14, 2011].
- [OMG 10] OMG. Object Management Group – Object Constraint Language (OCL) Specification, Version 2.2, February 2010. [online] Available at: <<http://www.omg.org/spec/OCL/2.2/PDF>> [Accessed November 14, 2011].
- [OMG 11_a] OMG. Object Management Group – Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, Version 1.1, January 2011. [online] Available at: <<http://www.omg.org/spec/QVT/1.1/PDF>> [Accessed November 14, 2011].
- [OMG 11_b] OMG. Object Management Group – Meta Object Facility (MOF) Core Specification, Version 2.4.1, August 2011. [online] Available at: <<http://www.omg.org/spec/MOF/2.4.1/PDF>> [Accessed November 14, 2011].
- [OMG 11_c] OMG. Object Management Group – MOF2 XMI (XML Metadata Interchange) Mapping Specification, Version 2.4.1, August 2011. [online] Available at: <<http://www.omg.org/spec/XMI/2.4.1/PDF>> [Accessed November 14, 2011].
- [OMG 11_d] OMG. Object Management Group – Unified Modeling Language: Infrastructure Specification, Version 2.4.1, August 2011. [online] Available at: <<http://www.omg.org/spec/UML/2.4.1/Infrastructure/PDF>> [Accessed November 14, 2011].
- [OMG 11_e] OMG. Object Management Group – Unified Modeling Language (UML) Superstructure Specification, Version 2.4.1, August 2011. [online] Avail-

-
- able at: <http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF>
[Accessed November 14, 2011].
- [OutSystems 10] OutSystems. Agile Platform Technical Overview. [online] Available at: <http://www.outsystems.com/RegisterAndGetResource.aspx?ResourceName=OutSystemsAgilePlatformTechnicalOverview> [Accessed March 15, 2012], 2010.
- [Pat 03] Paternò, Fabio. ConcurTaskTrees: An Engineered Notation for Task Models. In Dan Diaper and Neville Stanton, editors, *The Handbook of Task Analysis for Human-Computer Interaction*, chapter 24, pages 483–502. Lawrence Erlbaum Associates, May 2003. ISBN 978-0805844337.
- [Pie 02] Pierce, Benjamin C. *Types and Programming Languages*. The MIT Press, February 2002. ISBN 978-0262162098.
- [PLCS 07] Preciado, Juan Carlos, Linaje, Marino, Comai, Sara, and Sánchez-Figueroa, Fernando. Designing Rich Internet Applications with Web Engineering Methodologies. In Shihong Huang and Massimiliano Di Penta, editors, *Proceedings of the 9th IEEE International Symposium on Web Systems Evolution (WSE 2009)*, pages 23–30. IEEE Computer Society, Los Alamitos, CA, USA, October 2007. ISBN 978-1424414505. doi:10.1109/WSE.2007.4380240.
- [PMM 97] Paternò, Fabio, Mancini, Cristiano, and Meniconi, Silvia. ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models. In Steve Howard, Judith H. Hammond, and Gitte Lindgaard, editors, *Proceedings of INTERACT 97 – IFIP TC13 Sixth International Conference on Human-Computer Interaction*, volume 96 of *IFIP Conference Proceedings*, pages 362–369. Chapman & Hall, 1997. ISBN 978-0412809507.
- [RG 02] Ramakrishnan, Raghu and Gehrke, Johannes. *Database Management Systems*. McGraw-Hill, third edition, August 2002. ISBN 978-0072465631.
- [Ris 93] Ristad, Eric Sven. *The Language Complexity Game*. MIT Press, Cambridge, MA, USA, March 1993. ISBN 978-0262181471.
- [RLJ⁺ 03] Raible, Matt, Li, Sing, Johnson, Dave, Jepp, Daniel, Dalton, Sam, and Brown, Simon. *Pro JSP*. Apress, third edition, September 2003. ISBN 978-1590592250.
-

REFERENCES

- [RN 09] Russell, Stuart and Norvig, Peter. *Artificial Intelligence: A Modern Approach*. Prentice Hall, third edition, December 2009. ISBN 978-0136042594.
- [Roc 02] Rockley, Ann. *Managing Enterprise Content: A Unified Content Strategy (VOICES)*. New Riders Press, October 2002. ISBN 978-0735713062.
- [Roz 97] Rozenberg, Grzegorz, editor. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 1: Foundations*. World Scientific Publishing Company, 1997. ISBN 978-9810228842.
- [SAJ⁺ 02] Söderström, Eva, Andersson, Birger, Johannesson, Paul, Perjons, Erik, and Wangler, Benkt. Towards a Framework for Comparing Process Modelling Languages. In Anne Banks Pidduck, John Mylopoulos, Carson C. Woo, and M. Tamer Özsu, editors, *Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE 2002)*, volume 2348 of *Lecture Notes in Computer Science*, pages 600–611. Springer Verlag, London, UK, May 2002. ISBN 978-3540437383. doi:10.1007/3-540-47961-9_41.
- [SATE 03] Suh, Phil, Addey, Dave, Thiemecke, David, and Ellis, James. *Content Management Systems (Tools of the Trade)*. Glasshaus, October 2003. ISBN 978-1590592465.
- [SB 01] Schwaber, Ken and Beedle, Mike. *Agile Software Development with Scrum*. Prentice Hall, first edition, October 2001. ISBN 978-0130676344.
- [SBPM 08] Steinberg, David, Budinsky, Frank, Paternostro, Marcelo, and Merks, Ed. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, second edition, December 2008. ISBN 978-0321331885.
- [SC 01] Siau, Keng and Cao, Qing. Unified Modeling Language: A Complexity Analysis. *Journal of Database Management*, 12(1):26–34, 2001. ISSN 1063-8016.
- [SC 09] Severdia, Ron and Crowder, Kenneth. *Using Joomla: Building Powerful and Efficient Web Sites*. O'Reilly Media, December 2009. ISBN 978-0596804947.
- [Sch 06] Schmidt, Douglas C. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, February 2006. doi:10.1109/MC.2006.58.

-
- [Sch 08] Schou, Lasse. *Creating a model-driven Web application framework*. Master's thesis, Technical University of Denmark, Denmark, March 2008. [online] Available at: http://www.imm.dtu.dk/English/Research/Software_Engineering/Publications.aspx?lg=showcommon&id=213586 [Accessed May 31, 2010].
- [Sch 11] Schildt, Herbert. *Java The Complete Reference*. McGraw-Hill Osborne Media, 8th edition, June 2011. ISBN 978-0071606301.
- [Sco 09] Scott, Michael L. *Programming Language Pragmatics*. Morgan Kaufmann, third edition, April 2009. ISBN 978-0123745149.
- [SE 78] Susman, Gerald I. and Evered, Roger D. An Assessment of the Scientific Merits of Action Research. *Administrative Science Quarterly*, 23(4):582–603, December 1978. ISSN 0001-8392. doi:10.2307/2392581.
- [Sel 03] Selic, Bran. The Pragmatics of Model-Driven Development. *IEEE Software*, 20(5):19–25, September/October 2003. ISSN 0740-7459. doi:10.1109/MS.2003.1231146.
- [Sil 03] Silva, Alberto Rodrigues da. The XIS Approach and Principles. In *Proceedings of the 29th EUROMICRO Conference, New Waves in System Architecture*, pages 33–41. IEEE Computer Society, September 2003. ISBN 978-0769519968. doi:10.1109/EURMIC.2003.1231564.
- [SK 97] Sztipanovits, Janos and Karsai, Gabor. Model-Integrated Computing. *IEEE Computer*, 30(4):110–111, April 1997. ISSN 0018-9162. doi:10.1109/2.585163.
- [SK 09] Souer, Jurriaan and Kupers, Thijs. Towards a Pragmatic Model Driven Engineering Approach for the Development of CMS-based Web Applications. In *Proceedings of the 5th Model Driven Web Engineering Workshop (MDWE09)*, pages 31–45. June 2009.
- [SKR⁺ 08] Sousa, André, Kulesza, Uirá, Rummler, Andreas, Anquetil, Nicolas, Mitschke, Ralf, Moreira, Ana, Amaral, Vasco, and Araújo, João. A Model-Driven Traceability Framework to Software Product Line Development. In Jon Oldevik, Gøran K. Olsen, Tor Neple, and Richard Paige, editors, *ECMDA Traceability Workshop Proceedings (ECMDA-TW 2008)*, pages 97–109. SINTEF, June 2008. ISBN 978-8214043969.
-

- [SLMC 03] Silva, Alberto Rodrigues da, Lemos, Gonçalo, Matias, Tiago, and Costa, Marco. The XIS Generative Programming Techniques. In *Proceedings of the 27th International Computer Software and Applications Conference (COMPSAC 2003): Design and Assessment of Trustworthy Software-Based Systems*, pages 236–241. IEEE Computer Society, November 2003. ISBN 978-0769520209. doi:10.1109/CMPSAC.2003.1245347.
- [SS 08_a] Saraiva, João de Sousa and Silva, Alberto Rodrigues da. Evaluation of MDE Tools from a Metamodeling Perspective. *Journal of Database Management*, 19(4):21–46, October/December 2008. ISSN 1063-8016. doi:10.4018/jdm.2008100102.
- [SS 08_b] Saraiva, João de Sousa and Silva, Alberto Rodrigues da. The Web-Comfort Framework: An Extensible Platform for the Development of Web Applications. In *Proceedings of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2008)*, pages 19–26. IEEE Computer Society, September 2008. ISBN 978-0769532769. doi:10.1109/SEAA.2008.12.
- [SS 09_a] Saraiva, João de Sousa and Silva, Alberto Rodrigues da. CMS-based Web-Application Development Using Model-Driven Languages. In Kenneth Boness, João M. Fernandes, Jon G. Hall, Ricardo Jorge Machado, and Roy Oberhauser, editors, *Proceedings of the Fourth International Conference on Software Engineering Advances (ICSEA 2009)*, pages 21–26. IEEE Computer Society, September 2009. ISBN 978-0769537771. doi:10.1109/ICSEA.2009.12.
- [SS 09_b] Saraiva, João de Sousa and Silva, Alberto Rodrigues da. WebC-Docs: A CMS-based Document Management System. In Kecheng Liu, editor, *Proceedings of the International Conference on Knowledge Management and Information Sharing (KMIS 2009)*, pages 21–28. INSTICC Press, October 2009. ISBN 978-9896740139.
- [SS 10_a] Saraiva, João de Sousa and Silva, Alberto Rodrigues da. A Reference Model for the Analysis and Comparison of MDE Approaches for Web-Application Development. *Journal of Software Engineering and Applications*, 3(5):419–425, May 2010. ISSN 1945-3116. doi:10.4236/jsea.2010.35047.

-
- [SS 10_b] Saraiva, João de Sousa and Silva, Alberto Rodrigues da. Evaluation of MDE Tools from a Metamodeling Perspective. In Keng Siau and John Erickson, editors, *Principle Advancements in Database Management Technologies: New Applications and Frameworks*, pages 105–131. Information Science Publishing, December 2010. ISBN 978-1605669045. doi:10.4018/978-1-60566-904-5.ch005.
- [SS 10_c] Saraiva, João de Sousa and Silva, Alberto Rodrigues da. CMS-ML User’s Guide. Technical Report, Instituto Superior Técnico, April 2010. [online] Available at: <http://isg.inesc-id.pt/Modules/WebC_Docs/ViewDocumentDetails.aspx?DocumentId=93> [Accessed April 29, 2011].
- [SS 10_d] Saraiva, João de Sousa and Silva, Alberto Rodrigues da. Web-Application Modeling With the CMS-ML Language. In Luís S. Barbosa and Miguel P. Correia, editors, *Actas do II Simpósio de Informática (INForum 2010)*, pages 461–472. September 2010. ISBN 978-9899686304.
- [SS 11_a] Saraiva, João de Sousa and Silva, Alberto Rodrigues da. Design Issues for an Extensible CMS-based Document Management System. In Ana Fred, Jan L. G. Dietz, Kecheng Liu, and Joaquim Filipe, editors, *Knowledge Discovery, Knowledge Engineering and Knowledge Management: First International Joint Conference, IC3K 2009, Funchal, Madeira, Portugal, October 6-8, 2009, Revised Selected Papers*, volume 128 of *Communications in Computer and Information Science*. Springer Berlin/Heidelberg, 2011. ISBN 978-3642190322. doi:10.1007/978-3-642-19032-2_24.
- [SS 11_b] Saraiva, João de Sousa and Silva, Alberto Rodrigues da. CMS-IL User’s Guide. Technical Report, Instituto Superior Técnico, September 2011. [online] Available at: <http://isg.inesc-id.pt/Modules/WebC_Docs/ViewDocumentDetails.aspx?DocumentId=95> [Accessed April 29, 2011].
- [SS 11_c] Saraiva, João de Sousa and Silva, Alberto Rodrigues da. MYNK User’s Guide. Technical Report, Instituto Superior Técnico, November 2011. [online] Available at: <http://isg.inesc-id.pt/Modules/WebC_Docs/ViewDocumentDetails.aspx?DocumentId=96> [Accessed April 29, 2011].
- [SS 11_d] Schwaber, Ken and Sutherland, Jeff. The Scrum Guide – The Definitive Guide to Scrum: The Rules of the Game, October 2011. [online] Available at: <http://www.scrum.org/storage/scrumguides/Scrum_Guide.pdf> [Accessed March 21, 2012].
-

- [SSF⁺ 07] Silva, Alberto Rodrigues da, Saraiva, João, Ferreira, David, Silva, Rui, and Videira, Carlos. Integration of RE and MDE Paradigms: The ProjectIT Approach and Tools. *IET Software: On the Interplay of .NET and Contemporary Development Techniques*, 1(6):294–314, December 2007. doi:10.1049/iet-sen:20070012.
- [SSSM 07] Silva, Alberto Rodrigues da, Saraiva, João de Sousa, Silva, Rui, and Martins, Carlos. XIS – UML Profile for eXtreme Modeling Interactive Systems. In João M. Fernandes, Ricardo Jorge Machado, Ridha Khédri, and Siobhán Clarke, editors, *Proceedings of the 4th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2007)*, pages 55–66. IEEE Computer Society, Los Alamitos, CA, USA, March 2007. ISBN 0769527698. doi:10.1109/MOMPES.2007.19.
- [Sta 75] Standish, Thomas A. Extensibility in programming language design. In *American Federation of Information Processing Societies: Proceedings of the 1975 National Computer Conference*, volume 44 of *AFIPS Conference Proceedings*, pages 287–290. ACM, New York, NY, USA, May 1975. doi:10.1145/1499949.1500003.
- [SV 05_a] Silva, Alberto and Videira, Carlos. *UML, Metodologias e Ferramentas CASE, 2ª Edição, Volume 1*. Centro Atlântico, Portugal, May 2005. ISBN 9896150095.
- [SV 05_b] Stahl, Thomas and Voelter, Markus. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, Hoboken, New Jersey, U.S.A., May 2005. ISBN 978-0470025703.
- [Syr 11] Syriani, Eugene. *A Multi-Paradigm Foundation for Model Transformation Language Engineering*. Ph.D. thesis, School of Computer Science, McGill University, Canada, April 2011. [online] Available at: <<http://www.cs.mcgill.ca/~esyria/publications/dissertation.pdf>> [Accessed January 18, 2013].
- [Tho 04] Thomas, Dave A. MDA: Revenge of the Modelers or UML Utopia? *IEEE Software*, 21(3):15–17, May/June 2004. ISSN 0740-7459. doi:10.1109/MS.2004.1293067.
- [Vignette 09] Vignette Corporation. The Web Content Battle: Managing Pages Vs. Content – Which One Is Better For You? (whitepaper), June

-
2009. [online] Available at: <<http://www.vignette.co.uk/dafiles/docs/Downloads/WP-The-Web-Content-Battle.pdf>> [Accessed March 15, 2012].
- [VVP 08] Vlaanderen, Kevin, Valverde, Francisco, and Pastor, Oscar. Model-Driven Web Engineering in the CMS Domain: A Preliminary Research Applying SME. In Joaquim Filipe and José Cordeiro, editors, *10th International Conference on Enterprise Information Systems (ICEIS 2008), Revised Selected Papers*, volume 19 of *Lecture Notes in Business Information Processing*, pages 226–237. Springer, June 2008. ISBN 978-3642006692. doi:10.1007/978-3-642-00670-8_17.
- [Wri 09] Wright, Jevon M. A Modelling Language for Interactive Web Applications. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 689–692. IEEE Computer Society, November 2009. ISBN 978-0769538914. doi:10.1109/ASE.2009.44.
- [WSSK 07] Wimmer, Manuel, Schauerhuber, Andrea, Schwinger, Wieland, and Kargl, Horst. On the Integration of Web Modeling Languages. In Nora Koch, Antonio Vallecillo, and Geert-Jan Houben, editors, *Proceedings of the 3rd International Workshop on Model-Driven Web Engineering (MDWE 2007)*, volume 261 of *CEUR Workshop Proceedings*. CEUR-WS.org, July 2007. ISSN 1613-0073. [online] Available at: <<http://ceur-ws.org/Vol-261/paper05.pdf>> [Accessed August 18, 2011].
- [XLH⁺ 07] Xiong, Yingfei, Liu, Dongxi, Hu, Zhenjiang, Zhao, Haiyan, Takeichi, Masato, and Mei, Hong. Towards Automatic Model Synchronization from Model Transformations. In Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 164–173. ACM, New York, NY, USA, November 2007. ISBN 978-1595938824. doi:10.1145/1321631.1321657.

Appendix A

State of the Art: Model-Driven Approaches for Web Application Development

This appendix presents the state of the art that we consider relevant for our work, regarding MDE approaches and supporting languages for the development of web applications. The analyzed approaches are (1) WebML, (2) UWE, (3) XIS2, (4) OutSystems Agile Platform, and (5) Microsoft Sketchflow.

A.1 WebML (Web Modeling Language)

The WebML (Web Modeling Language)¹² [CFB⁺ 02] is a graphical language for modeling data-intensive web applications (i.e., with the purpose of publishing and maintaining large quantities of data). WebML is aimed at supporting designers in the modeling of advanced web application features, such as user-specific personalization of contents and the delivery of information on multiple kinds of devices (e.g., PCs, PDAs) [MFV 06]. Due to these characteristics, designers using WebML can express the main features of a site in a manner that is both high-level and platform-independent. Despite its academic origins, WebML is mainly supported by the commercial tool WebRatio³, which addresses the entire WebML development life cycle.

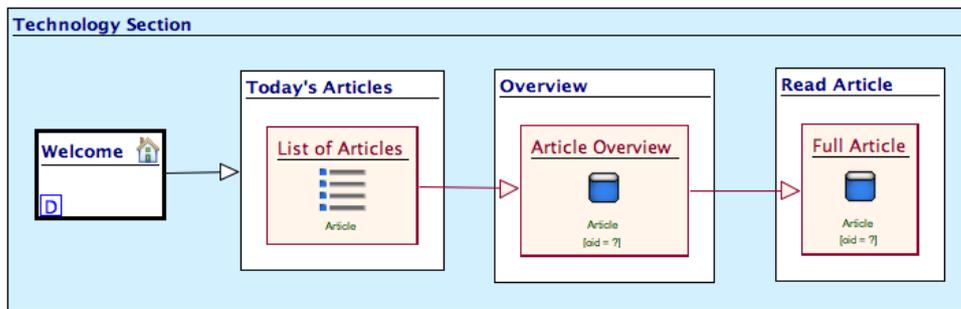
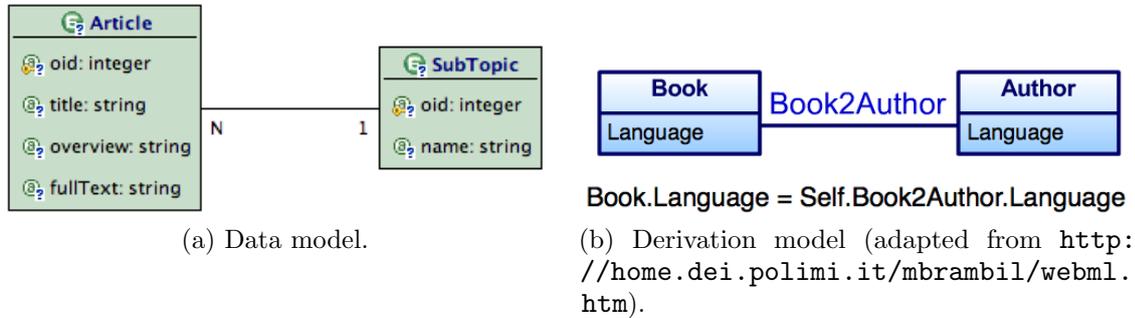
¹<http://www.webml.org> (accessed on March 15th, 2012)

²Marco Brambilla, “WebML tutorial”, <<http://home.dei.polimi.it/mbrambill/webml.htm>> (accessed on March 15th, 2012)

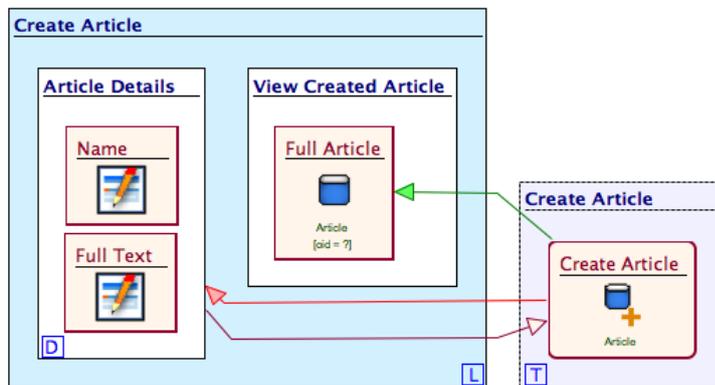
³<http://www.webratio.com> (accessed on March 15th, 2012)

A.1.1 Modeling Language

The specification of a website in WebML uses the following types of models: (1) the Data Model; (2) the Derivation Model; (3) the Hypertext Model; and (4) the Presentation Model. Some examples of WebML models are illustrated in Figure A.1.



(c) Hypertext model.



(d) Content-management operation.

Figure A.1: Examples of WebML models.

Data Model. The Data Model is used to express the content model of the website. It defines a small set of concepts (namely **Entity**, **Attribute**, **Relationship**, and **Is-A Hierarchy**) that are similar to what can be found in UML class diagrams [OMG 11.e] or Entity-Relation (ER) diagrams. **Entity** is conceptually similar to UML's **Class**, and represents a set of objects in the domain of the application. An **Attribute** is a property

of an **Entity**. A **Relationship** is a connection between **Entities**. An **Is-A Hierarchy** is very similar to UML's **Generalization**, and is used for the grouping and classification of **Entities**. Figure A.1a illustrates an example of a WebML Data model that uses most of these elements (except for **Is-A Hierarchy**).

Derivation Model. The Derivation Model can be considered as an increment of the Data Model, because it is used to define additional *redundant* data by means of derivation queries over the Data Model, expressed using WebML-OQL (WebML Object Query Language). This data is meant to support the operations defined in the user interface (e.g., a list of all **Books** written by a certain **Author**). Figure A.1b shows an example of a Derivation model, in which a **Book's Language** is derived from its **Author's Language**.

Hypertext Model. The Hypertext Model addresses two different (but closely related) perspectives, Composition and Navigation. These perspectives determine how hypertext (i.e., text to be displayed on a computer, with links to other text) is divided into smaller blocks and how those blocks are linked between themselves, respectively.

In the Composition perspective, the designer divides the website's hypertext into smaller pieces, called **Pages**, which are the basic unit of navigation in WebML. **Pages** can be grouped into **Areas**, which are sets of **Pages** that share a particular topic; **Areas** can be nested, allowing the designer to further divide the hypertext in a hierarchical manner. Furthermore, **Pages** are grouped into one or more **SiteViews**, which are also sets of **Pages** and/or **Areas**. While an **Area** is nothing more than a grouping of related **Pages** (and possibly other nested **Areas**), a **SiteView** provides a coherent view of the website (according to criteria such as the output device or the kind of user browsing the site). Thus, a certain kind of user (e.g., a manager) may be able to perceive the site (by browsing and exploring) in a different manner than a different kind of user (e.g., an anonymous user). **SiteViews** are either public or private, as a form of access control.

Each **Page** can contain **Content Units**, which are components that publish details regarding specific **Entities**; the instances whose details are to be published can be specified by providing selection conditions called *selectors*. WebML provides a set of **Content Units** (e.g., **DataUnit**, **IndexUnit**) that address patterns typically found in such data-intensive web applications (e.g., show a list of instances). Each **Content Unit** can define a set of input parameters – to determine which instances should be published – as well as a set of output parameters (e.g., the identifier of the selected instance). Data is published by means of an orchestration of **Content Units**, where **Content Units** provide the inputs for other **Content Units**. Figure A.1c shows an example of a very simple **Area** –

the **Technology** section of a news site – with a set of **Pages** that guide the user through the selection of an **Article** to read.

On the other hand, the **Navigation** perspective is used to establish directed links between the various pages and units. WebML defines two kinds of links, contextual and non-contextual. A **Non-Contextual Link** is just a link between two web pages, typically rendered as an HTML anchor, that indicates the navigation possibilities presented to the user and does not carry any context. A **Contextual Link** is a directed link between two **Content Units**, and has the following semantics: (1) move from one place to another (i.e., between different sets of information being published, and possibly between different **Pages**); (2) carry information – the navigation context – between those places; and (3) activate computations to publish/modify information. In Figure A.1c, the link between the **Pages Welcome** and **Today’s Articles** is a **Non-Contextual Link**, while the links between the **Content Units** are **Contextual Links**.

Finally, the Hypertext Model also allows the specification of basic content management workflows to manipulate data, by using **Operation Units**. An **Operation Unit** has the following characteristics: (1) it represents a built-in operation (create, update, or delete a certain instance of an **Entity** or **Relationship**) or a generic operation (to be implemented afterward by a developer); (2) it receives input parameters from one or more links; and (3) it has two kinds of output links, designated **OK** and **KO**, that will be followed if the operation succeeds or fails, respectively. Figure A.1d provides a simple example of such an operation, in which the user is to provide information (in a form) to create an **Article**. If the operation is successful, then the **OK** link (from **Create Article** to **Full Article**) will be followed, and the details of the new **Article** will be shown. Otherwise, the **KO** link (from **Create Article** to **Article Details**) will be followed, and the form for submitting **Article** information will be presented to the user again.

Presentation Model. Although WebML addresses user interface (UI) modeling in the Hypertext Model (but only at an abstract level, with no concepts like layout or color), the concrete aspects of the web application’s UI are actually specified in the Presentation Model (available in WebRatio). The Presentation Model is where the graphical layout and appearance of the **Pages** is specified, as well as the positioning of **Content Units**. These details are specified by providing a grid model of the **Page**, within which the various **Content Units** of the **Page** are placed (although WebRatio does provide an element, called *Custom Location*, that can be used for particular positioning needs). WebRatio also allows designers to specify the graphical appearance of **Pages** and **Units**, through the usage of CSS classes and XSLT stylesheets.

A.1.2 Development Approach

The WebML development approach follows the typical workflow: (1) model the website, by using the language described above; (2) generate source code; (3) customize the generated source code; and (4) deploy the website. It should be noted that this approach does not exclude the manual customization of generated source code.

A.2 UWE (UML-based Web Engineering)

The UWE (UML-based Web Engineering) [KK 08]⁴ is an object-oriented software engineering approach for web application development, based on OMG standards (e.g., UML, OCL, XMI), that focuses on models and model transformations. UWE comprises [KKH 01]: (1) a modeling language for the graphical representation of web application models; (2) a technique supporting semi-automatic generation; and (3) a process supporting the development life cycle of web applications. The main characteristic of UWE is the use of UML for all models [KKH 01], in particular using pure UML whenever possible (i.e., stereotypes are defined only when UML's semantics reveal themselves insufficient). UWE is defined as an extension to UML; it is also available as a UML profile [OMG 11_e, KK 08] tailored for web application modeling [KK 02].

A.2.1 Modeling Language

The UWE metamodel is structured according to the Separation of Concerns principle, in the package structure illustrated in Figure A.2. The Core package contains all the elements that are a part of the UWE language, while the Adaptivity package is intended to allow the language to be extended in a cross-cutting manner [KK 08]. The Core package itself is divided into the following packages (called *Models* in UWE): (1) Requirements, (2) Content, (3) Navigation, (4) Presentation, and (5) Process. Figure A.3 provides some examples of UWE models.

Requirements Model. The Requirements Model does not define any UWE language elements, as it only considers the usage of regular UML use case diagrams to specify the various processes and activities that the web application must consider.

Content Model. The Content Model also does not define any additional elements, as it resorts to plain UML class diagrams to model the web application's contents (e.g., strings).

⁴<http://uwe.pst.ifi.lmu.de> (accessed on March 15th, 2012)

APPENDIX A. STATE OF THE ART: MODEL-DRIVEN APPROACHES FOR WEB APPLICATION DEVELOPMENT

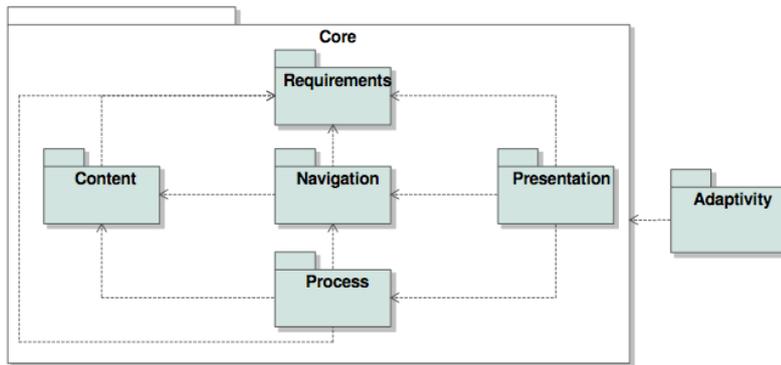
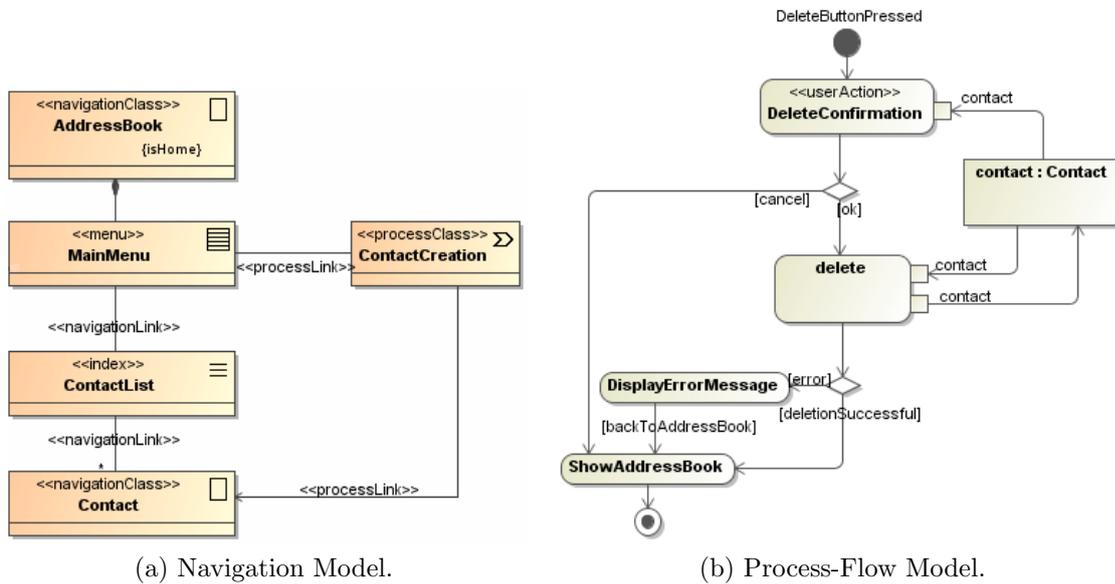
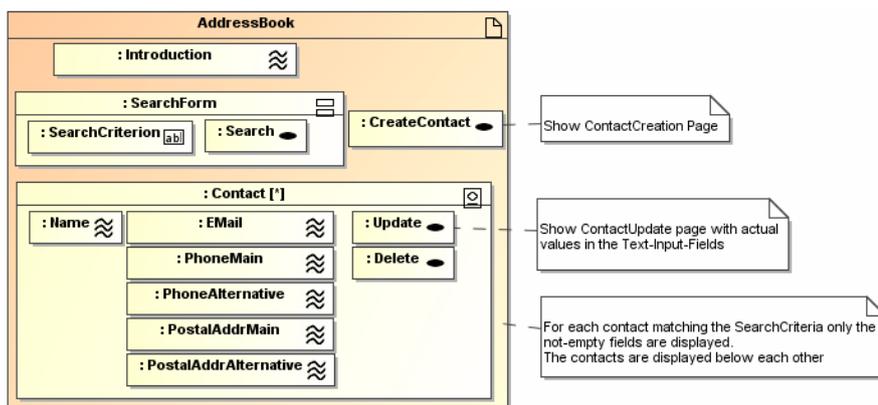


Figure A.2: UWE metamodel's structure (extracted from [KK 08]).



(a) Navigation Model.

(b) Process-Flow Model.



(c) Presentation Model.

Figure A.3: Examples of UWE models (extracted from <http://uwe.pst.ifi.lmu.de/teachingTutorial>).

Navigation Model. The Navigation Model, as the name suggests, is used to specify the possible navigation flows that the user can follow (i.e., the existing web pages and the navigation flows connecting them). This is done by means of UML class diagrams, decorated with UWE stereotypes representing nodes and links. Figure A.3a shows a small example of a navigation model for an address book (from the Address Book tutorial⁵), with a set of `navigationLinks` and `processLinks` connecting the nodes in the model.

Process Model. The Process Model specifies the behavior of the web application, namely user actions and the application's response to them. It is comprised of: (1) the Process Structure Model, which describes the relations between the various processes (or sets of activities) of the web application; and (2) the Process Flow Model, which consists of regular UML activity diagrams specifying how each process is to be performed (a UWE stereotype, `UserAction`, is used to identify activities requiring explicit interaction with the user). Figure A.3b illustrates an example of a process-flow model, which specifies the activities that must be performed whenever the user initiates the process of deleting a contact from an address book.

Presentation Model. The Presentation Model specifies an abstract view of the application's UI (i.e., its web pages). To this end, the Presentation Model defines stereotypes for various kinds of elements (e.g., text input boxes, anchors, and buttons), but without providing concrete details such as CSS styles or HTML elements. Figure A.3c shows an example of a presentation model for an address book page.

A.2.2 Development Approach

UWE-based development also follows the typical workflow: (1) model the website; (2) generate source code; (3) customize the generated source code; and (4) deploy the website. However, UWE defines a set of model-to-model transformations to enable the quick generation of additional models from the entities already defined; an example is the Content-to-Navigation transformation, which generates a possible Navigation model from the information defined in the Content model.

Like WebML, this approach also does not exclude the manual customization of generated source code. Nevertheless, in WebML's case such customization is typically necessary when specific features (that are not related to data manipulation) are necessary. On the other hand, UWE does not address relevant aspects such as the specification of business logic or the web application's look-and-feel, which makes such customization almost always necessary.

⁵<http://uwe.pst.ifi.lmu.de/teachingTutorial> (accessed on March 15th, 2012)

A.3 XIS2 (eXtreme Modeling for Interactive Systems)

The XIS2 (eXtreme modeling for Interactive Systems) [SSSM 07, SSF⁺ 07] modeling language is a UML profile [OMG 11.e], oriented toward interactive systems for desktop- and web-based platforms (instead of just considering web-based platforms).

A.3.1 Modeling Language

The XIS2 language follows the Separation of Concerns principle, and defines six concrete views: (1) the Domain View; (2) the BusinessEntities View; (3) the Actors View; (4) the UseCases View; (5) the InteractionSpace View; and (6) the Navigation View. Some XIS2 model examples are presented in Figure A.4.

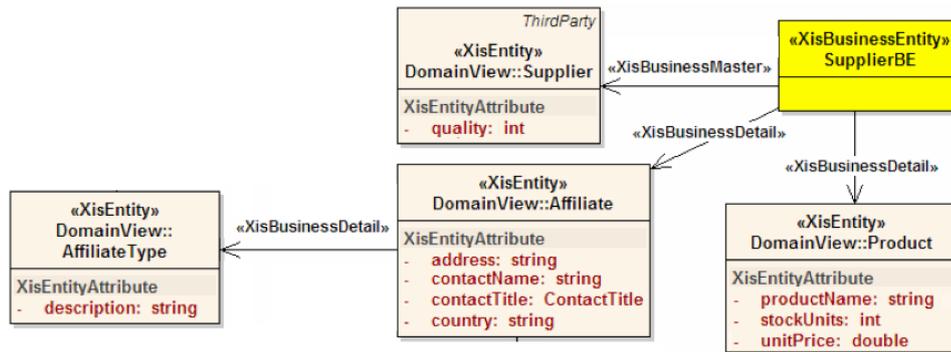
Domain View. The Domain View is used to model the relevant problem-domain entities in a traditional manner, by using regular UML class diagrams with some XIS2-specific stereotypes (which allow the designer to convey additional information or code generation constraints).

BusinessEntities View. The BusinessEntities View is used to organize the Domain View entities into higher-level entities (known as *business entities*), using a coarser level of granularity and compositions that express master–detail relationships. These business entities will be manipulated in the context of specific use cases (specified in the UseCases View). Figure A.4a provides an example of the BusinessEntities View.

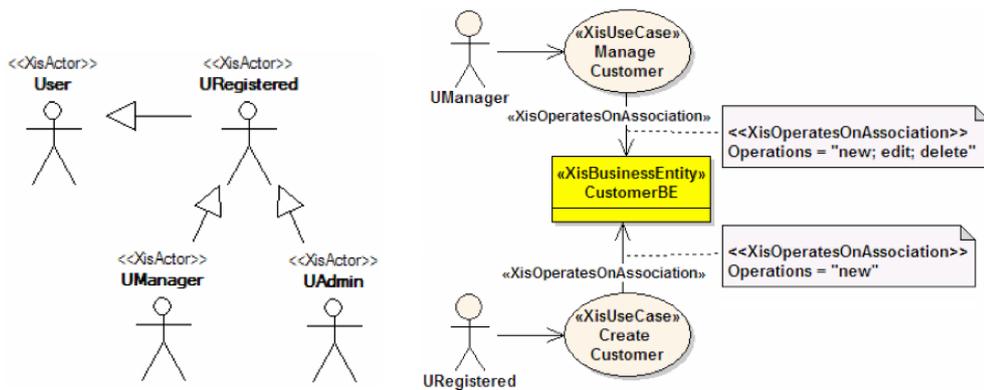
Actors View. The Actors View is used to specify the various types of user (i.e., actors) that can interact with the system; this view consists of a typical UML **Actor** hierarchy model, using regular UML **Actors** and **Generalizations**. Figure A.4b provides a simple example of the Actors View.

UseCases View. The UseCases View is used to define the responsibilities of the system’s actors (which were defined in the Actors View), by means of UML use cases that associate the actors with the business entities; the concrete responsibilities of each actor are specified by defining entity manipulation patterns – such as **new** or **delete** – or custom manipulation patterns (which must be manually implemented afterward). Figure A.4c provides a simple example of the UseCases View.

InteractionSpace View. The InteractionSpace View defines the existing interaction spaces (screens that receive and present information to the user) and their UI elements

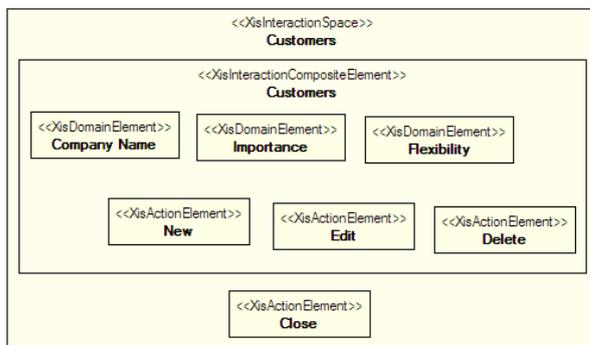


(a) BusinessEntities View.

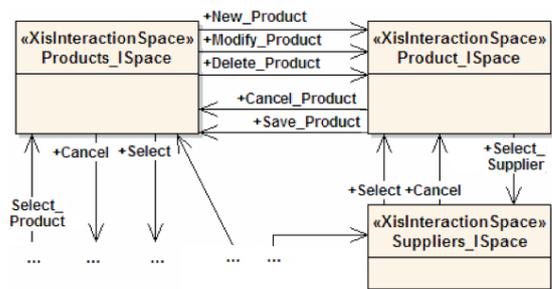


(b) Actors View.

(c) UseCases View.



(d) InteractionSpace View.



(e) Navigation View.

Figure A.4: Examples of XIS2 models (extracted from [SSSM 07] and [SSF⁺ 07]).

(e.g., buttons, images), using a WYSIWYG-oriented approach. This view is also used to establish access control between actors and those UI elements (e.g., to only show certain elements to specific actors). Figure A.4d illustrates an example of an interaction space.

Navigation View. The Navigation View determines the possible navigation flows between the system’s interaction spaces, and the causes that can trigger those navigation flows (i.e., what action must occur for the user to be presented with another interaction space). Figure A.4e provides an example of this view.

A.3.2 Development Approach

The XIS2 development approach also follows the typical workflow: (1) model the website; (2) generate source code; (3) customize the generated source code; and (4) deploy the website. The views presented above are meant as a guide for this approach, in which designers are to specify: (1) the Entities View (consisting of the Domain View and optionally the BusinessEntities View); (2) afterward, the Use-Cases View (consisting of the Actors View and the optional UseCases View), based on the concepts defined in the Entities View; and (3) finally, the User-Interfaces View (consisting of the InteractionSpace View and the Navigation View), either manually or by means of an adequate model-to-model transformation. Figure A.5 provides a high-level overview of the XIS2 development approach.

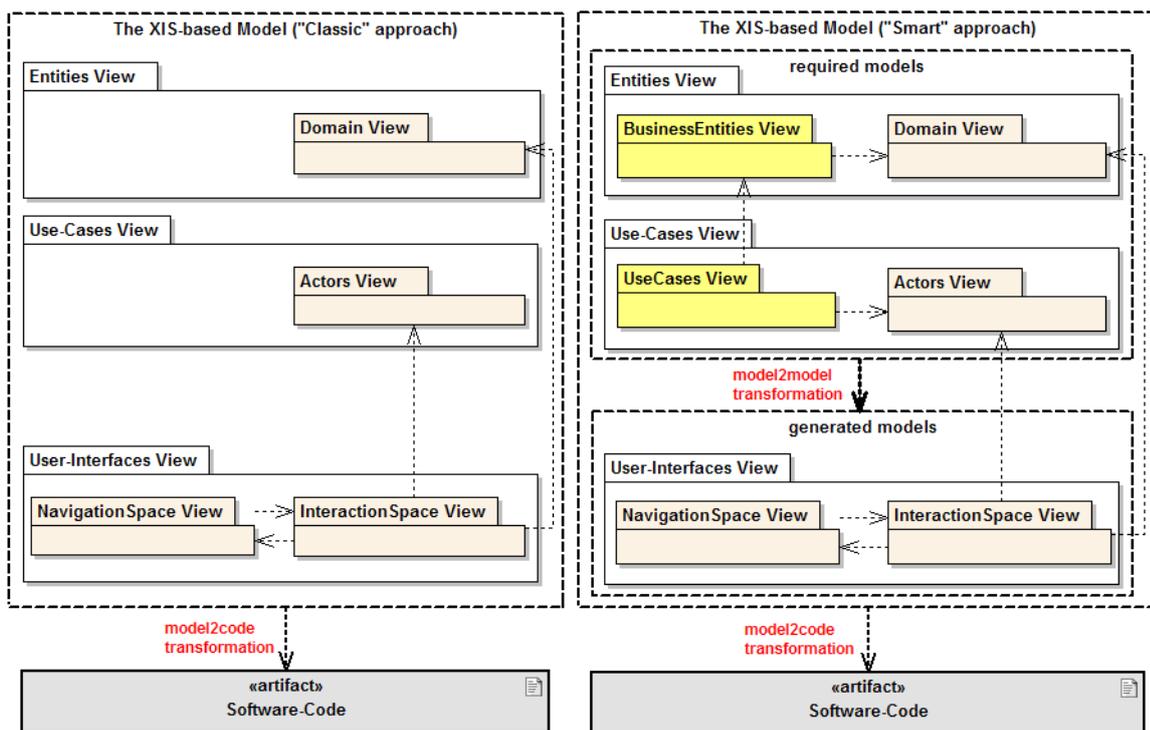


Figure A.5: The XIS2 views and design approaches (extracted from [SSF⁺ 07]).

Nevertheless, XIS2 is still not expressive enough to represent every possible feature that can be expected of interactive systems [SSSM 07]. Thus, specific aspects (such as custom business logic patterns or platform-specific UI details like the usage of CSS classes) must be manually implemented by programmers, after source code generation.

A.4 OutSystems Agile Platform

The OutSystems Agile Platform⁶ is a platform meant to support and integrate the deployment and management of web business applications, by means of agile methodologies. It provides a visual development environment from which all web application aspects (e.g., UI, business logic, database schema) can be defined without specifying low-level source code; integration with legacy systems, such as existing applications and databases, is also addressed by this tool. Those web applications can then be easily deployed in an automatic manner, by using 1-Click Publishing.

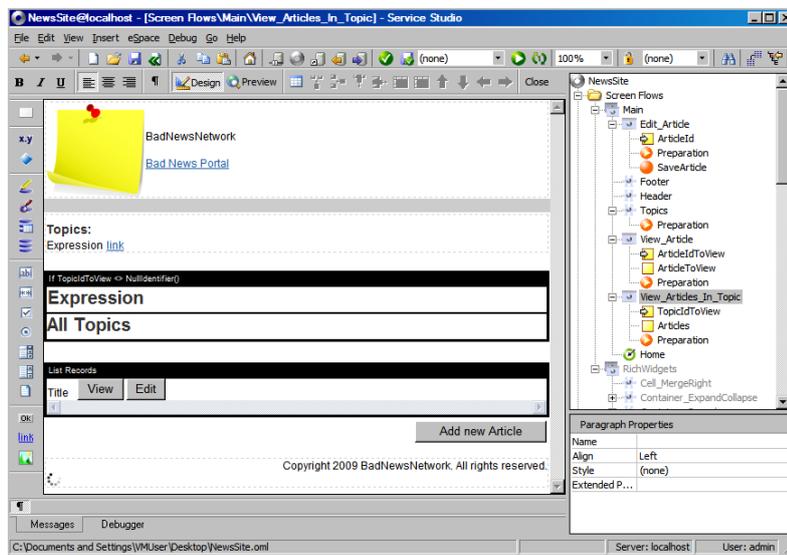
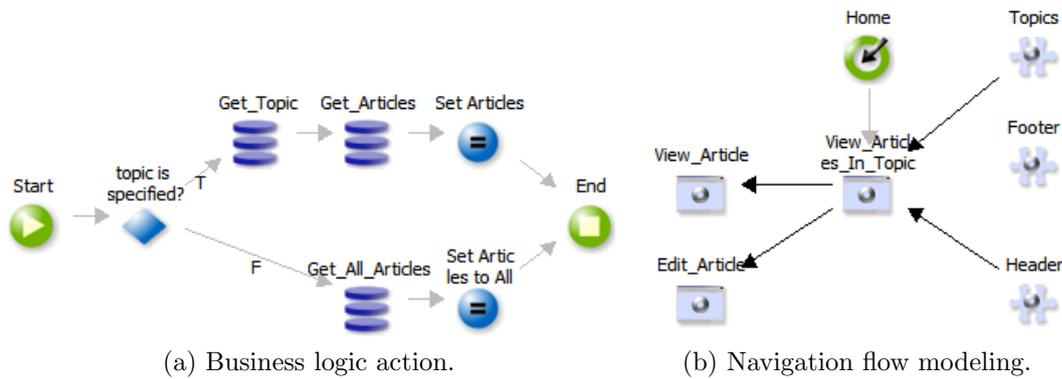
A.4.1 Modeling Language

The modeling language used by the Agile Platform addresses the functionality expected of web applications (persistence, business logic, and presentation). In OutSystems' nomenclature, the application or project itself is referred to as an **eSpace**. It should be noted that there is a high level of integration between the various aspects of the modeling language, as the specification of aspects such as business logic, UI, and navigation flow are all tightly coupled among themselves (e.g., a page, besides specifying its HTML contents, also specifies the input parameters that it requires), and so the language is not segmented into a set of views. Some examples of Agile Platform models are presented in Figure A.6.

Domain Modeling. Domain modeling is addressed by a mechanism very similar to traditional database modeling. This mechanism is based on the concept of **Entity**, which contains a set of **Attributes**, each having a specific type (e.g., integer, text, or an identifier of another **Entity**); these concepts correspond to database tables and columns, respectively. The tool also provides the concept of **Static Entity**, which is an **Entity** with a predefined set of instances or records that cannot be changed at runtime. It should be noted that the reason why this mechanism is akin to database modeling is because the Agile Platform will automatically convert the specified domain model into a database schema and deploy it to a target database. Thus, designers have complete control over the persistence layer, albeit at the cost of some loss of abstraction between the domain model and the persistence layer.

The language also provides the concept of **Structure**, intended to provide information of a more coarser-grained nature (e.g., the result of a query over a set of **Entities**). This allows the designer to define complex, UI-oriented, data structures, without requiring particular adjustments to the data model itself. The main difference between **Entities** and **Structures** is that the former are persistent elements (i.e., they will survive between

⁶<http://www.outsystems.com> (accessed on March 15th, 2012)



(c) UI modeling.

Figure A.6: Examples of models in the OutSystems Agile Platform.

application restarts), while the latter are only temporary elements that will be lost when the application ends.

Business Logic Modeling. Business logic modeling is addressed through **Actions**, which are workflows to be executed when some event occurs (e.g., a **Screen Preparation** is an **Action** that will be performed when the page processing starts, and some UI elements may also trigger certain **Actions**). The Agile Platform includes a wide variety of **Actions** (such as **Entity** queries, assigning values to variables, evaluating variables to decide which path to follow, or navigation), and additional **Actions** (defined in other **eSpaces**) can be referenced.

Each **Action** is modeled in a manner similar to flowchart diagrams or UML Activity diagrams [OMG 11_e]: the workflow starts in the initial node and performs a set of operations (other **Actions**) until it reaches the end node. Additionally, **Actions** can be

composed by other (simpler) **Actions**. Figure A.6a shows a simple example of an **Action** (the **Screen Preparation** for the **Web Screen** illustrated in Figure A.6c), in which a set of articles will be loaded depending on whether a topic has been specified.

Navigation Flow Modeling. The Agile Platform addresses navigation flow modeling by the traditional means of defining a directed graph, where nodes correspond to **Web Screens** or **Web Blocks** (these two concepts are explained in the UI modeling aspect, further down this text) and the edges define the possible navigation flows that can be followed. Figure A.6b provides an example of navigation flow modeling, in which: (1) the **Topics** and **Header Blocks** both provide navigation links to the main **View Articles In Topic Screen**; and (2) the **View Articles In Topic Screen** can navigate to the **View Article** and **Edit Article Screens** (the navigation flows to return to the main **Screen** are not represented in those **Screens** because they are provided by the **Header Block**).

User Interface Modeling. UI modeling is addressed via a WYSIWYG editor that allows designers to specify **Web Screens** and **Web Blocks**. **Web Screens** correspond to HTML pages, while **Web Blocks** are components that can be reused in other pages or components (e.g., a website's banner and footer), in a fashion similar to ASP.NET's usage of pages and controls. **Web Screens** and **Web Blocks** are also composed of **Widgets**, which can be representations of HTML elements – input boxes, radio buttons – or typical interaction patterns, such as message boxes, contextual help, or transitions supported by AJAX (Asynchronous JavaScript And XML). Figure A.6c shows a basic example of the Agile Platform's UI modeling, in which a list of articles (loaded by the **Action** shown in Figure A.6a) is displayed according to the selected topic.

A.4.2 Development Approach

The Agile Platform development approach follows a very straightforward workflow: (1) model the website, by using the language described above; and (2) deploy the website with the 1-Click Publishing mechanism. All development is supposed to be done in a visual manner (including tasks like debugging, which in other approaches is usually done in a source code-oriented manner). It is important to note that, unlike the other analyzed approaches, the Agile Platform does not consider the need for developers to manually change the internal details of the application (e.g., generated code, database structures); in fact, this kind of change is not allowed in the development process, and so any such changes done by the developer will likely break the application.

A.5 Microsoft Sketchflow

The Microsoft Sketchflow design tool⁷, a part of Microsoft Expression Blend 3⁹, is used to quickly build application prototypes at the early stages of the application development process. Sketchflow is intended to address the following issues: (1) experiment with different UI alternatives; (2) communicate ideas between stakeholders and designers; and (3) gather and analyze feedback from stakeholders regarding design ideas.

Sketchflow supports two types of project, oriented toward either the Microsoft Windows Presentation Foundation (WPF)¹⁰ or Microsoft Silverlight¹¹ technologies respectively. Sketchflow prototypes themselves are valid WPF or Silverlight applications, allowing developers to use those prototypes as primary artifacts and refine throughout the development process.

A.5.1 Modeling Language

Sketchflow allows the modeling of the following aspects: (1) navigation flow, (2) user interface, and (3) behavior (more precisely, responses to user input). Domain modeling is not addressed explicitly, although Sketchflow does support the use of live data sources (or even just sample data) in a database-oriented fashion, the presence of which would imply that a domain model (or at least a persistence model) has been previously defined.

Prototypes for an application are designed in Sketchflow by drawing (1) screens and (2) navigation flows between them. Sketchflow also allows the specification of states and behavior. Figure A.7 presents some examples of Sketchflow models for a news site.

Navigation Flow Modeling. This is addressed by the *Sketchflow Map*, which specifies (1) the navigation flow between **Screens**, and (2) the composition relationships between **Screens** and **Composition Screens** (these two concepts are explained in the next paragraph). Figure A.7a illustrates a simple example of a Sketchflow Map.

User Interface Modeling. One of the focal points of Sketchflow, UI modeling is performed by drawing **Screens** and **Component Screens**. A **Screen** is a representation of an application window with which the user will interact, and is composed of: (1) regular Sketchflow controls, which represent WPF or Silverlight controls (depending on the nature

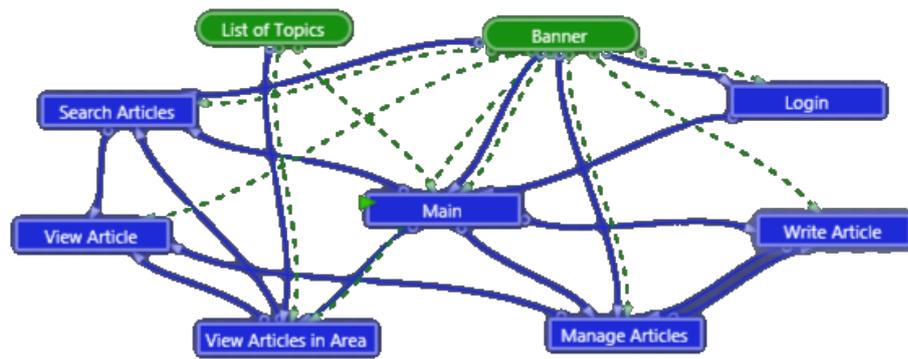
⁷http://www.microsoft.com/expression/products/Sketchflow_Overview.aspx (accessed on March 15th, 2012)

⁸Christian Schormann, “Sketchflow Concepts: An Overview”, <<http://electricbeach.org/?p=214>> (accessed on March 15th, 2012)

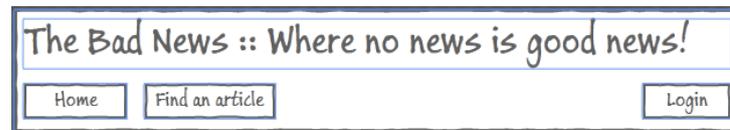
⁹<http://www.microsoft.com/expression> (accessed on March 15th, 2012)

¹⁰<http://windowsclient.net> (accessed on March 15th, 2012)

¹¹<http://www.silverlight.net> (accessed on March 15th, 2012)



(a) The Sketchflow Map.



(b) A Component Screen (banner for a news site).



(c) A Screen to search for articles.



(d) A Screen to write an article.

Figure A.7: Examples of Microsoft Sketchflow models.

of the project); and (2) **Component Screens**, which can be considered as *screen pieces* that can be reused across **Screens** (e.g., a website banner), allowing designers to create prototypes with a consistent interface. **Component Screens** themselves are composed of regular Sketchflow controls, and possibly of other **Component Screens**. Figure A.7b illustrates a **Component Screen** (a banner for the news site), while Figures A.7c and A.7d present two **Screens** addressing different functionalities. Note that the top component of both Figures A.7c and A.7d is actually the banner defined in Figure A.7b, reused in both **Screens**.

Behavior. Although Sketchflow does not allow the specification of business logic modeling in a traditional manner (such logic is supposed to be developed in a later stage of the development process, after the stakeholders are satisfied with the designed prototype), it

does allow designers to specify some behavior in their prototypes, by means of two mechanisms: **Behaviors** and **States**. **Behaviors** are modular reusable blocks that express some kind of interactivity between the user and the prototype (e.g., make a component support drag-and-drop, activate a state, or navigate to a screen), and can be applied to UI components in a visual drag-and-drop manner. On the other hand, **States** are non-visual components that allow the designer to specify a state machine (by defining states and transitions between them); UI components can then use **Behaviors** to trigger transitions in the state machine(s) available in the current **Screen**. **Behaviors** are extensively used within Sketchflow, as the possibility of controls triggering navigation or transitions in state machines (offered by the tool with only a few clicks) is actually a shortcut to specifying the corresponding **Behaviors**.

Also, as mentioned above, Sketchflow does allow data-binding to live data sources or to sample data (defined within Sketchflow); nevertheless, this binding is read-only, and so the prototype cannot modify data.

A.5.2 Development Approach

Microsoft Sketchflow does not provide a development approach *per se*, because Sketchflow prototypes are supposed to be a part of a larger (and more traditional) development process. Also, Sketchflow prototypes by themselves are not supposed to be sufficient input for a code generation mechanism, and so they will inevitably be involved in regular programming tasks at some point during the application's development.

Appendix B

State of the Art: CMS Systems

This appendix presents the state of the art regarding CMS systems that we consider relevant for our research work. The analyzed systems are (1) DotNetNuke, (2) Drupal, (3) Joomla, (4) Vignette Content Management, and (5) WebComfort.

B.1 DotNetNuke

DotNetNuke¹ [HRW 09] is an open-source CMS system with an extensive user community, based on Microsoft ASP.NET² technology. It is written in Visual Basic .NET (VB.NET), and requires Microsoft Internet Information Services (IIS) and Microsoft SQL Server in order to run.

The DotNetNuke CMS provides a set of out-of-the-box features that can be found in most CMS systems (e.g., forums, blogs, user and role management), allowing non-technical users to quickly create and customize websites. It also supports multi-tenancy, as users can define multiple sites within a single installation.

The system presents a page-centric approach to the management of the website, as users will make content available by defining tabs and modules, within the context of which they can then define the content that should be displayed.

DotNetNuke also provides the means for it to be extended and customized, namely through third-party skins, modules, and providers that enable additional custom functionality. The administrator of a DotNetNuke installation can also upload and install additional modules via DotNetNuke's administration pages; after installation, the module is available to any of the website's pages.

Furthermore, due to its skinning features, DotNetNuke can be customized to present a different layout without requiring any particular knowledge of ASP.NET. A DotNetNuke

¹<http://www.dotnetnuke.com> (accessed on March 15th, 2012)

²<http://www.asp.net> (accessed on March 15th, 2012)

Skin consists of simple HTML files (with placeholders for content, menus, and other features), as well as support files (e.g., images and stylesheets), packaged in a zip file.

B.1.1 Development Approach

Like most CMS systems, DotNetNuke does not provide any particular approach (model-driven or otherwise) to module development. Development of modules is done via traditional programming. This involves developing source code (in VB.NET or C#) that uses the features provided by the DotNetNuke framework and API.

B.2 Drupal

Drupal³ [BBH⁺ 08] is a widely used open-source CMS, written in PHP and capable of running in any web server that can run PHP, such as Apache or Microsoft IIS. Regarding persistence, it provides out-of-the-box support for MySQL and PostgreSQL databases.

The standard release of Drupal is known as *Drupal core*, and provides features that can typically be found in CMS systems (e.g., the possibility of customizing page layout, available syndication feeds, blogs, forums). Drupal also supports multi-tenancy (i.e., the possibility of defining multiple sites using a single installation).

Unlike DotNetNuke, Drupal follows a content-centric approach to website management. Although administrators can customize some parts of the website's structure, the main emphasis is placed on producing content; any pages that are defined by the administrator will only be a means through which users can access content (although there are other means, namely the Menu mechanism).

Furthermore, Drupal core can be extended with additional features and behavior, by means of plugin modules contributed by members of its user community. Such modules will be responsible for handling various issues, such as interpreting URL addresses to determine what content to show.

B.2.1 Development Approach

Drupal does not provide any particular approach to the development of plugin modules. Development of such modules is done via traditional programming, which requires knowledge of Drupal itself, the PHP programming language, and Drupal's module API.

³<http://drupal.org> (accessed on March 15th, 2012)

B.3 Joomla

Joomla⁴ [SC 09] is another open-source CMS, written in PHP and capable of running in the Apache or Microsoft IIS web servers. Regarding persistence, it supports only the MySQL database.

Unlike other CMS systems, the standard release of Joomla does not provide most typical CMS features (e.g., the possibility of customizing page layout, available syndication feeds, blogs, forums) out-of-the-box, although such features are available as free add-ons. Furthermore, Joomla itself does not support multi-tenancy yet, but some free add-ons do provide that feature and, at the time of writing of this dissertation, some work is being done within Joomla's core to support this feature natively.

In a manner that is very similar to Drupal, the Joomla CMS follows a content-centric approach to website management. Administrators can customize some parts of the website's structure, but nevertheless the main emphasis is placed on producing content.

Joomla can be extended with additional features and behavior, by means of plugin modules that are contributed by its community members (in fact, most of the basic CMS features provided by Joomla are obtained by means of such modules that must be installed by the administrator).

B.3.1 Development Approach

Like Drupal, Joomla does not consider any particular approach to the development of plugin modules: development of Joomla modules is done via traditional programming, by using the PHP programming language and Joomla's API.

B.4 Vignette

Vignette⁵, developed by the Vignette Corporation, is a widely used commercial suite of content management, portal, collaboration, document, and record management software tools. It supports the JavaEE and Microsoft .NET platforms, and runs on an IBM DB2 database. Vignette also supports multi-tenancy.

Of particular relevance to our work are two of its applications, (1) Content Management and (2) Portal. The Content Management application is essentially a back-office that allows non-technical users to create, edit, and track content within workflows. On the other hand, the Portal application enables the publishing of that content on a Vignette-powered website.

⁴<http://www.joomla.org> (accessed on March 15th, 2012)

⁵<http://www.vignette.com> (accessed on March 15th, 2012)

Although Vignette’s philosophy is to always focus on content instead of website structure, its approach to content management is actually a mix of content-centric and page-centric. Content is created and edited by users (sometimes by using workflows), but is published by using a template-based mechanism that receives a set of contents and returns a website, allowing users to access that content.

B.4.1 Development Approach

From the information that we could find in our analysis, Vignette does not consider any particular approach to the development of extensions: development is done via traditional programming, by using the Java programming language and Vignette’s API.

B.5 WebComfort

WebComfort⁶ [SS 08_b] is a web content and application management framework. It is based on Microsoft ASP.NET technology, and it can run on any Microsoft IIS server. Regarding persistence, it provides out-of-the-box support for Microsoft SQL Server, PostgreSQL, and Oracle databases. WebComfort does not support multi-tenancy, as multiple websites require the existence of multiple installations.

Although originally developed to be a CMS system, WebComfort has recently evolved to provide developers with an extensive API that allows them to control almost every aspect of the CMS. Nevertheless, it has not abandoned its CMS roots, and it also provides out-of-the-box support for a variety of features that can be found in typical CMS systems (e.g., announcements, links management, user and role management).

In a manner similar to DotNetNuke and many other CMS systems, WebComfort uses a page-centric approach to website management. Administrators and users have the define the website’s structure, which consists of tabs and modules, and content is made available within the context of those modules.

Third-party developers can extend WebComfort by defining toolkits, modules, and extenders. A module is WebComfort’s basic unit of content publishing, and it provides mechanisms to manage and layout a certain kind of content. A toolkit can be considered, from a simplistic point of view, as an integrated collection of modules that provide specific functionalities. Finally, an extender allows a developer to add behavior to certain aspects of the platform, or change their predetermined behavior (depending on the semantics of the aspects that are being extended).

⁶<http://www.siquant.pt/WebComfort> (accessed on March 15th, 2012)

B.5.1 Development Approach

Like DotNetNuke, WebComfort does not provide any particular approach to the development of modules or toolkits. Although WebComfort does provide some support for their development and deployment (namely via the WebComfort API and the Toolkits mechanism), these tasks are done via traditional programming and manual processing, by using the C# or VB.NET programming languages.