

Dragan Gašević
Dragan Djurić
Vladan Devedžić

Model Driven Architecture and Ontology Development

 Springer

Model Driven Architecture and Ontology Development

Dragan Gašević · Dragan Djurić
Vladan Devedžić

Model Driven Architecture and Ontology Development

Foreword by Bran Selic

With 153 Figures and 7 Tables

 Springer

Authors

Dragan Gašević
School of Interactive Arts
and Technology
Simon Fraser University Surrey
13450 102 Ave.
Surrey, BC V3T 5X3
Canada
dgasevic@sfu.ca
and dgasevic@acm.org

Dragan Djurić
Vladan Devedžić
FON – School of
Business Administration
University of Belgrade
Jove Ilića 154
11000 Belgrade
Serbia and Montenegro
dragandj@gmail.com
devedzic@fon.bg.ac.yu

Library of Congress Control Number: 2006924633

ACM Computing Classification (1998): D.2.10, D.2.11, H.4.0, I.2.4

ISBN-10 3-540-32180-2 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-32180-4 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable for prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media
springer.com

© Springer-Verlag Berlin Heidelberg 2006
Printed in Germany

The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Typeset by the Authors
Production: LE-TeX Jelonek, Schmidt & Vöckler GbR, Leipzig
Cover design: KünkellOpka Werbeagentur, Heidelberg

Printed on acid-free paper 45/3100/YL - 5 4 3 2 1 0

To our families

Foreword

The first time I paid attention to the term “ontology” was in the late 1980s when I was part of an engineering team that was responsible for defining what we would now call a domain-specific modeling language. In our case, the domain was telecommunications software and the purpose of our language was to give system architects the ability to describe the high-level structure of their software in the most direct and most expressive manner possible.

The team members were all experienced designers with deep knowledge of the domain so that we had no trouble putting together the initial list of key language concepts. We knew that we needed to include standard architectural modeling constructs such as components, ports, connectors, and the like. We also wanted our language to be object-oriented, so notions such as class, objects, and inheritance were added to the list. However, soon after this very promising start, all progress ground to a halt. Somehow, the definition of the seemingly trivial fine-grain details of these constructs kept eluding us despite long, passionate, and occasionally acrimonious discussions that can only be compared to medieval theological debates.

It was our good fortune that at that point we met Professor Doug Skuce of the University of Ottawa. He had a method and a tool that helped us develop an explicit ontology for our domain. From that exercise we learned that our difficulties stemmed from the fact that, although we shared a general intuition for the chosen constructs of our language, there were numerous subtle and *unstated* differences in our individual conceptualizations that were a barrier to mutual understanding. Furthermore, we discovered that certain commonly used terms had multiple meanings – all equally valid – but which we had not differentiated adequately, leading to much confusion. Only after we had defined our ontology, which included semi-formal definitions of all key terms and their relationships, were we able to finish our task successfully.

Ever since I’ve felt that defining a formal domain ontology is a useful and often necessary step in almost any software project. This is because software deals principally with ideas rather than physical artifacts. Whereas the nature of physical artifacts is generally self-evident, this is not

the case with conceptual entities, which are products of the mind. As we all know, different minds see the same thing differently.

The definition and application of ontologies for developing software systems is a central theme of this book. However, the book is about much more than that. It explains, in a clear and didactic manner, how a variety of recent buzzword developments in software theory and practice (intelligent agents, Model Driven Architecture, metamodeling, etc.) can be combined, and brings us to the threshold of the next step in the evolution of the World Wide Web: the *Semantic Web*. Like the Internet before it, the Semantic Web promises to introduce a significant and qualitatively new phenomenon into our lives. This is because it endows the network of disparate information that is currently accessible on the Internet with *meaning*. Because this meaning can be gleaned and processed *automatically* by software, the Semantic Web opens up the exciting and awe-inducing possibility of a unified global intelligence accessible to all.

In the first half of the book, the authors navigate deftly through the prolific and highly confusing *gemischt* of technologies, tools, and standards such as XML, RDF, OWL, MDA, or UML and explain how they relate to each other in the context of the idea of the Semantic Web. They introduce the notion of *modeling spaces*, which provides a conceptually simple yet comprehensive framework for understanding and addressing issues within the domain considered. Using that framework, the second half of the book describes a practical strategy for realizing key elements of the Semantic Web based on existing industry standards.

The book is equally suited to those who merely want to be informed of the relevant technological landscape, to practitioners dealing with concrete problems, and to researchers seeking pointers to potentially fruitful areas of research. The writing is technical yet clear and accessible, and is illustrated throughout with useful and easily digestible examples.

I would also highly recommend this book to sociologists studying the interplay between society and technology. It clearly demonstrates that the core technologies required for constructing the Semantic Web are available and moving forward inexorably. Society must be prepared to deal with something so ripe with potential. We must understand not only how the Semantic Web can be useful but also what dangers lurk within it.

Ottawa, Canada

Bran Selic

December 2005

Preface

The idea of ontologies emerged in applied artificial intelligence some time ago as a means for sharing knowledge [Gruber, 1993]. Following the development of ontologies and related Web technologies (e.g., HTML and XML), Tim Berners-Lee, Jim Hendler, and Ora Lassila envisioned the next generation of the Web, called the Semantic Web [Berners-Lee et al., 2001]. Being based on ontologies, the Semantic Web has the potential for semantically richer representations of things (e.g., Web pages, applications, and persons) and their relations on the Web, and thus should provide us with more intelligent services. That idea might have initially sounded very futuristic and too enthusiastic, but it has recruited a lot of important players from both academia and industry into very extensive and well-funded research efforts. Today, we have quite impressive results, manifested by standards that have been adopted (RDF and OWL), development frameworks (Jena), best-practice and deployment recommendations, and many applications (e.g. PiggyBank).

Of course, researchers are still facing many challenges in their efforts to accomplish the full vision of the Semantic Web. Probably the first and most important goal is to persuade many industrial developers and software engineers to use and develop ontologies in their everyday practice. However, ontologies rely on well-defined and semantically powerful concepts in artificial intelligence such as description logics, reasoning, and rule-based systems. Since software engineers are largely unfamiliar with these concepts, ontologies have a price that must be paid for the benefits that they provide.

Trying to address the above problems, researchers have started exploring the potential of some widely adopted software engineering tools and methodologies for ontology development. Stephen Cranefield did the pioneering research by proposing that UML, a well-known software modeling language, should be used for ontology development [Cranefield, 2001a]. After him, several researchers have explored further the similarities, differences, and equivalences between UML and ontology languages, as well as the potential of the most recent software engineering initiative called the Model Driven Architecture (MDA), and its accompanying standards (the Meta-Object Facility (MOF) and XML

Metadata Interchange (XMI)) for ontology development [Baclawski et al., 2002a; Djurić et al., 2005a; Falkovych et al., 2003]. This resulted in the initiation of a process for adopting an MDA-based ontology standard by the Object Management Group (OMG), a software engineering standardization consortium [OMG ODM RFP, 2003]. The standard is intended to define the Ontology Definition Metamodel (ODM) using the MOF (used for specifying UML as well), a UML extension (the Ontology UML Profile, or OUP) to allow UML tools to be used to fully develop ontologies, and a set of transformations between the ODM, the OUP, UML, and Semantic Web ontology languages (e.g., RDF(S) and OWL). When completed, the ODM specification is expected to be in the form of an OMG language, like UML and CWM.

In this book we try to fill the gap in the literature covering the subject of applications of the MDA for ontology development on the Semantic Web. Other books cover either the MDA initiative [Kleppe et al., 2003; Mellor et al., 2003b] or the Semantic Web (i.e., ontology development) [Fensel, 2004; Stuckenschmidt & van Harmelen, 2005; Zhong et al., 2003] only. This book gives a comprehensive overview of both themes, with the main emphasis on how we can employ MDA-related standards to develop Semantic Web ontologies. The book is closely related to the recent OMG initiative for the ODM. The book is the first description of that new language.

The book is based on our experience obtained from a series of tutorials entitled “MDA Standards for Ontology Development” that we have given at several international conferences on the Semantic Web (the International Semantic Web Conference and the European Semantic Web Conference) and on software engineering (the International UML Conference and the International Conference of Web Engineering).

Organization and Structure

The book is divided into three parts. Part I covers the basics of both the main topics – ontologies and the MDA. First, Chap. 1 gives a brief overview of the field of knowledge representation in artificial intelligence. Chapters 2 and 3 introduce the main concepts of ontologies, the Semantic Web, standards, applications, tools, and some open research questions. Next, Chap. 4 explains the Model Driven Architecture, and its main standards (the MOF and XMI) and mechanisms (UML profiles). Part I is concluded by Chap. 5 with modeling spaces, a conceptual framework

defined to provide an easier understanding of approaches to modeling such as ontologies and MOF-defined modeling languages (UML and the ODM).

Part II is the central part of the book. It starts with Chap. 6, which presents a comprehensive review of several approaches and tools that aim to bridge the gap between ontology development and software engineering methodologies. This chapter also lists the relations between UML and ontology languages. Chapter 7 explains the motivation for the forthcoming OMG ontology development standard for the ODM, and the requirements the standard has to fulfill. Next, Chaps. 8 and 9 describe the current specifications of the OMD and the Ontology UML Profile, respectively. Finally, Chap. 10 analyzes the mappings between MDA-based languages (the ODM and the OUP) and Semantic Web ontology languages.

Part III is dedicated to applications that will support the practical use of languages that conform to the OMG ontology development standard, and to some practical aspects of how to develop ontologies using those MDA-based languages. First, Chap. 11 is a short tutorial showing how to develop ontologies using the OUP in two state-of-the-art UML tools (MagicDraw and Poseidon for UML). Chapter 12 describes an implementation of an ontology-building platform called AIR, developed entirely following MDA principles. Finally, Chap. 13 discusses two examples of ontologies developed using the OUP and MDA standards.

Throughout the book, we use many ontologies, UML and other MDA-based models, and transformations between them. In order to allow you to try them out and use them in practice, we have created a Web page containing supplementary resources. You can reach this Web page at <http://www.modelingspaces.org>. Besides the resources referred to in the book, this Web page contains the slide handouts of the tutorials that we have given at many international conferences.

Acknowledgments

Some of the content of this book has been previously published by organizations that are not part of the Springer group. We thank these organizations for their kind permission to use the material for this book, in particular

- IEEE Press
- Rinton Press
- Journal of Educational Technology & Society, and
- Journal of Object Technology.

We are very grateful to Jelena Jovanović, Nenad Krdžavac, and Colin Knight for a huge amount of help in preparing this book and for giving us many useful comments and suggestions. We would also like to thank all members of the GOOD OLD AI research group (<http://goodoldai.org.yu>) at the University of Belgrade, Serbia and Montenegro for providing us with a great environment and many useful ideas that enormously improved the quality of the book. Dragan Gašević is very grateful to Marek Hatala, Tom Calvert, Griff Richards, and other colleagues from the LORNET research network at Simon Fraser University in Vancouver, Canada, for their great support and understanding.

We thank Bran Selic of IBM Rational Software for agreeing to write the book's foreword. We are also very grateful to Jean Bézivin of the University of Nantes whose research in the field of model-driven engineering inspired us to start exploring relations between MDA and ontologies. A special mention should be given to Ralf Gerstner of Springer-Verlag, who invited us to write this book and had a lot of patience in working with us. We are very thankful to Evan Wallace, Elisa Kendall, Bob Colomb, Anna Gerber, Dan Chang, Lewis Hart, and other colleagues from the OMG's Ontology PSIG for giving us up-to-date information about the OMG's ODM standardization efforts in the last three years.

Vancouver, Canada
Belgrade, Serbia and Montenegro
Belgrade, Serbia and Montenegro

Dragan Gašević
Dragan Djurić
Vladan Devedžić

March 2006

Contents

Part I Basics

1. Knowledge Representation.....	3
1.1 Basic Concepts	4
1.2 Cognitive Science	7
1.3 Types of Human Knowledge	11
1.4 Knowledge Representation Techniques.....	14
1.4.1 Object–Attribute–Value Triplets	15
1.4.2 Uncertain Facts	15
1.4.3 Fuzzy Facts	16
1.4.4 Rules	17
1.4.5 Semantic networks.....	18
1.4.6 Frames	19
1.5 Knowledge Representation Languages.....	19
1.5.1 Logic-Based Representation Languages	20
1.5.2 Frame-Based Representation Languages	27
1.5.3 Rule-Based Representation Languages.....	29
1.5.4 Visual Languages for Knowledge Representation	32
1.5.5 Natural Languages and Knowledge Representation	35
1.6 Knowledge Engineering	36
1.7 Open Knowledge Base Connectivity (OKBC)	39
1.8 The Knowledge Level	41
2. Ontologies	45
2.1 Basic Concepts	46
2.1.1 Definitions	46
2.1.2 What Do Ontologies Look Like?	48
2.1.3 Why Ontologies?	50
2.1.4 Key Application Areas.....	55
2.1.5 Examples.....	57
2.2 Ontological Engineering.....	58
2.2.1 Ontology Development Tools.....	58
2.2.2 Ontology Development Methodologies	65
2.3 Applications.....	69
2.3.1 Magpie	69
2.3.2 Briefing Associate.....	70

2.3.3	Quickstep and Foxtrot.....	71
2.4	Advanced Topics.....	72
2.4.1	Metadata, Metamodeling, and Ontologies.....	72
2.4.2	Standard Upper Ontology.....	74
2.4.3	Ontological Level.....	76
3.	The Semantic Web.....	79
3.1	Rationale.....	80
3.2	Semantic Web Languages.....	81
3.2.1	XML and XML Schema.....	81
3.2.2	RDF and RDF Schema.....	84
3.2.3	DAML+OIL.....	87
3.2.4	OWL.....	90
3.2.5	SPARQL.....	92
3.3	The Role of Ontologies.....	95
3.4	Semantic Markup.....	96
3.5	Semantic Web Services.....	100
3.6	Open Issues.....	104
3.7	Quotations.....	107
4.	The Model Driven Architecture (MDA).....	109
4.1	Models and Metamodels.....	109
4.2	Platform-Independent Models.....	110
4.3	Four-Layer Architecture.....	112
4.4	The Meta-Object Facility.....	114
4.5	Specific MDA Metamodels.....	117
4.5.1	Unified Modeling Language.....	117
4.5.2	Common Warehouse Metamodel (CWM).....	118
4.5.3	Ontology Definition Metamodel.....	119
4.6	UML Profiles.....	120
4.6.1	Examples of UML Profiles.....	121
4.7	An XML for Sharing MDA Artifacts.....	123
4.8	The Need for Modeling Spaces.....	126
5.	Modeling Spaces.....	127
5.1	Modeling the Real World.....	128
5.2	The Real World, Models, and Metamodels.....	129
5.3	The Essentials of Modeling Spaces.....	131
5.4	Modeling Spaces Illuminated.....	134
5.5	A Touch of RDF(S) and MOF Modeling Spaces.....	137
5.6	A Touch of the Semantic Web and MDA Technical Spaces.....	139
5.7	Instead of Conclusions.....	141

Part II The Model Driven Architecture and Ontologies

6. Software Engineering Approaches to Ontology Development.....	145
6.1 A Brief History of Ontology Modeling	145
6.1.1 Networked Knowledge Representation and Exchange Using UML and RDF.....	145
6.1.2 Extending the Unified Modeling Language for Ontology Development	150
6.1.3 The Unified Ontology Language	155
6.1.4 UML for the Semantic Web: Transformation-Based Approach.....	156
6.1.5 The AIFB OWL DL Metamodel.....	159
6.1.6 The GOOD OLD AI ODM Proposal	160
6.2 Ontology Development Tools Based on Software Engineering Techniques	160
6.2.1 Protégé.....	161
6.2.2 DUET (DAML UML Enhanced Tool)	164
6.2.3 An Ontology Tool for IBM Rational Rose UML Models.....	165
6.2.4 Visual Ontology Modeler (VOM)	167
6.3 Summary of Relations Between UML and Ontologies	168
6.3.1 Summary of Approaches and Tools for Software Engineering-Based Ontology Development	169
6.3.2 Summary of Differences Between UML and Ontology Languages	169
6.3.3 Future Development	172
7. The MDA-Based Ontology Infrastructure.....	173
7.1 Motivation	173
7.2 Overview	174
7.3 Bridging RDF(S) and MOF.....	176
7.4 Design Rationale for the Ontology UML Profile.....	178
8. The Ontology Definition Metamodel (ODM).....	181
8.1 ODM Metamodels	181
8.2 A Few Issues Regarding the Revised Joint Submission	183
8.3 The Resource Description Framework Schema (RDFS) metamodel.....	184
8.4 The Web Ontology Language (OWL) Metamodel.....	190
9. The Ontology UML Profile	201
9.1 Classes and Individuals in Ontologies	201
9.2 Properties of Ontologies	204
9.3 Statements.....	206
9.4 Different Versions of the Ontology UML Profile.....	207

10. Mappings of MDA-Based Languages and Ontologies	211
10.1 Relations Between Modeling Spaces.....	211
10.2 Transformations Between Modeling Spaces	214
10.3 Example of an Implementation: an XSLT-Based Approach	217
10.3.1 Implementation Details.....	218
10.3.2 Transformation Example	219
10.3.3 Practical Experience	222
10.3.4 Discussion.....	225

Part III Applications

11. Using UML Tools for Ontology Modeling	229
11.1 MagicDraw	230
11.1.1 Starting with MagicDraw	230
11.1.2 Things You Should Know when Working with UML Profiles	232
11.1.3 Creating a New Ontology	234
11.1.4 Working with Ontology Classes	237
11.1.5 Working with Ontology Properties.....	240
11.1.6 Working with Individuals	244
11.1.7 Working with Statements	246
11.2 Poseidon for UML.....	247
11.2.1 Modeling Ontology Classes in Poseidon	249
11.2.2 Modeling Ontology Individuals and Statements in Poseidon	250
11.3 Sharing UML Models Between UML tools and Protégé Using the UML Back End.....	251
12. An MDA Based Ontology Platform: AIR	255
12.1 Motivation	255
12.2 The Basic Idea	256
12.3 Metamodel – the Conceptual Building Block of AIR	258
12.4 The AIR Metadata Repository.....	259
12.5 The AIR Workbench	262
12.6 The Role of XML Technologies.....	264
12.7 Possibilities.....	265
13. Examples of Ontology	267
13.1 Petri Net Ontology.....	267
13.1.1 Organization of the Petri Net Ontology	269
13.1.2 The Core Petri Net Ontology in the Ontology UML Profile	272
13.1.3 Example of an Extension: Upgraded Petri Nets.....	275

13.2 Educational Ontologies.....	278
13.2.1 Conceptual Solution.....	279
13.2.2 Mapping the Conceptual Model to Ontologies	281
References.....	291
Index	305

Part I Basics

1. Knowledge Representation

Knowledge is understanding of a subject area [Durkin, 1994]. It includes concepts and facts about that subject area, as well as relations among them and mechanisms for how to combine them to solve problems in that area.

The branch of computer science that studies, among other things, the nature of human knowledge, understanding, and mental skills is artificial intelligence (AI). The goal of AI is to develop computer programs to do the things that humans usually call “intelligent”.

There are dozens of definitions of AI in the literature, none of them being complete and all-encompassing. The reason for such incompleteness is the complexity of the phenomenon of intelligence. Still, the study of human knowledge and its representation in computers is so very central in AI that even some definitions of the discipline recognize that fact. For example:

AI is the branch of computer science that attempts to approximate the results of human reasoning by organizing and manipulating factual and heuristic knowledge. [Bandwidth Market, 2005]

Artificial intelligence. The range of technologies that allow computer systems to perform complex functions mirroring the workings of the human mind. Gathering and structuring knowledge, problem solving, and processing a natural language are activities possible by an artificially intelligent system. [e-Learning Guru, 2005]

It is said that AI aims at making programs that represent, encode, and process knowledge about problems – facts, rules, and structures – rather than the data of problems.

The objective of this chapter is to survey some of the most important concepts and principles of knowledge representation. The chapter takes a pragmatic approach – rather than providing a comprehensive study of all aspects of knowledge representation, it only covers those that lay the foundations for understanding the rest of the book.

1.1 Basic Concepts

In AI, *knowledge storing* is the process of putting knowledge, encoded in a suitable format, into computer memory. *Knowledge retrieval* is the inverse process – finding knowledge when it is needed. *Reasoning* means using knowledge and problem-solving strategies by means of an intelligent program to obtain conclusions, inferences, and explanations. An important prerequisite for these processes is *knowledge acquisition* – gathering, organizing, and structuring knowledge about a topic, a domain, or a problem area, in order to prepare it for putting into the system.

AI studies the above processes starting from observations of the human mind's intelligent activities. For example, a person who has never had an experience with handheld computers may happen to take one in his/her hand. The person may examine it for a while, learn how to “play” with it, and memorize the experience, as well as the handheld's features. Memorizing (knowledge storing) involves abstraction and modeling. The person's mind will somehow store all the essential features of the handheld computer, and will be able to retrieve that knowledge later when he/she takes the handheld computer to work with it again. Moreover, if he/she has had experience with desktop or laptop computers before, he/she will probably get along with the handheld more easily by reasoning about its possible features.

Unlike the human mind, computers do not have such a transparent mechanism for acquiring and representing knowledge internally, just by themselves [Arnold & Bowie, 1985]. They rely on humans to put the knowledge into their memories. It is then the task of humans to decide on how to represent knowledge inside computers.

So far, AI has come up with a number of different representations, or models of various *types of human knowledge* (see Sect. 1.3). Each one is associated with a method to structure and encode knowledge in an intelligent system, and a specific related data structure (such as a table, a tree, or a link). None of them is perfect or the best. Each representation and each data structure has some deficiencies that make it inadequate for representing all kinds of knowledge. Moreover, careful selection of a knowledge representation may simplify problem solving, whereas an unfortunate selection may lead to difficulties or even failure to find a solution. Complex problems require a combination of several different representations.

The above paragraphs raise the question: What is the theory behind knowledge representation? An incomplete and vague, but good approximate answer is: cognitive science. *Cognitive science* is the

interdisciplinary study of mind and intelligence, embracing philosophy, psychology, artificial intelligence, neuroscience, linguistics, and anthropology [Stanford Encyclopedia of Philosophy, 2005]. One of the key issues in cognitive science is the study of human thinking in terms of representational structures in the mind, and computational procedures that operate on those structures [Hofstadter, 1994]. An important hypothesis is that the human mind has *mental representations* analogous to computer data structures. Moreover, cognitive science assumes that the computational procedures of the mind are similar to computational algorithms. There are a number of approaches in this field of study, and there is no consensus about the nature of the representations and computations that constitute thinking.

Different mental representations of the human mind, as proposed by cognitive theorists – such as logical propositions, rules, concepts, images, and analogies – constitute the basis of different *knowledge representation techniques* (e.g., rules, frames, and logic; see Sect. 1.4). Each technique has advantages and disadvantages for capturing particular types of human knowledge. This is so because each representation technique emphasizes certain information about a topic or a problem, while ignoring other information [Hofstadter, 1994].

A special category of techniques is used to represent *inexact and uncertain knowledge* about the world. Human expressions such as “That’s probably true” imply a certain *degree of belief* in a fact or event that is being talked about. In other words, much human knowledge is not absolutely certain; on the contrary, it is often inexact, imprecise, and incomplete. The techniques used to represent such knowledge are typically based on probability theory, the theory of evidence, and fuzzy-logic theory.

In order to be practical, every knowledge representation technique needs a *notation*. A formal notation often used for representing knowledge is *first-order predicate calculus*. Of course, it is not suitable for all types of knowledge. The notation of *description logics* has also proven expressive enough to represent knowledge formally. More recently, the trend has been to use XML syntax for representing knowledge structures.

Starting from various notations, AI researchers have developed a number of *knowledge representation languages* to support various representation techniques. Being artificial languages, knowledge representation languages usually have precisely defined grammars to help the parsing of expressions in these languages and ease the machine processing of the knowledge represented. However, ease of parsing sometimes comes at the expense of human readability.

Note that knowledge representation and intelligent reasoning are always intertwined, both in the human mind and in AI [Minsky, 1985; Russell & Norvig, 2002]. The mind uses mental representations of the world and mental procedures such as deduction, search, matching, rotation, and retrieval to make inferences about the world. Likewise, AI reasoners use encoded knowledge and inference procedures to solve problems.

This intertwining makes it possible to answer the fundamental question: What is knowledge representation? Davis et al. [Davis et al., 1993] answer this question by discussing five different roles of a knowledge representation.

The first role of a knowledge representation is to serve as a *surrogate*, i.e., a *substitute* for things from the real world inside an intelligent entity. Typically, things from the real world are external to the reasoner, whereas the corresponding representations are internal to it. Internal representations enable reasoning to substitute for direct interaction with the world. Operations on representations substitute for operations on real-world things. Being surrogates and only approximations/models of reality, all knowledge representations are inevitably imperfect in terms of completeness, which may lead to incorrect reasoning.

The second role of a knowledge representation is *making a set of ontological commitments* about the real world. By expressing only a part of the information about the world and ignoring the rest, knowledge representation also makes a kind of selection: it determines what the intelligent system can “see” in the world, and to what it remains “blind”. In fact, it is the knowledge representation that brings in focus the aspects of the world that are believed to be relevant. For example, a representation of human hand may include details of its anatomy, but omit completely any knowledge of its function. This representation may be encoded in various languages and using various notations; however, the essential part is not the form but the content.

This is not to say that the form is irrelevant to ontological commitment. On the contrary, the very selection of the representation used to encode a piece of knowledge implies commitments and constraints. Rules, logic, frames, etc. all embody a point of view about the kinds of things that are important in the world, i.e., commitments about what to see in it. Rules imply inference logic about object–attribute–value triplets, whereas frames bring objects and their interactions into view.

The third role of a representation is as a *fragmentary theory of intelligent reasoning*. Every representation indicates at least a partial belief about how people reason intelligently and what it means to reason intelligently. Furthermore, there is a set of conclusions that an intelligent system is permitted to draw for each particular representation; this set is

usually very large and insufficiently constrained for practical purposes. On the other hand, there is also an indication of which inferences are appropriate to draw for each representation, i.e. the intelligent inferences. Finally, every representation typically incorporates only a part of the complex phenomenon of intelligent reasoning.

The fourth role of a knowledge representation is as a *medium for efficient computation*. That is to say, to use a representation a machine must compute with it, which makes computational efficiency a central issue in the notion of representation. To this end, it should be noted that every representation, by its nature, provides some guidance about how to organize knowledge in order for computation with it to be most efficient. For example, frames are suitable for taxonomic reasoning. Hence organizing them into taxonomic hierarchies increases the reasoning efficiency. Other representations may suggest different organizations of knowledge.

A knowledge representation is also a *medium for human expression*, which is its fifth role. Recall that, in the course of knowledge acquisition, humans (knowledge engineers) gather, organize, and store knowledge in the knowledge base of an intelligent system. To do so, humans have to create and communicate representations to machines and to each other. In that process, questions such as the following arise: How general is the representation used? How precise is it? How expressive? How suitable is it as a language for communication? How easy is it for humans to think or “talk” in that language? What kinds of things are easy to express in that language? And so on.

1.2 Cognitive Science

In brief, cognitive science studies the nature of the mind. More precisely, it studies cognition – mental states and processes such as thinking, reasoning, remembering, language understanding and generation, visual and auditory perception, learning, consciousness, and emotions, [Clark, 2001]. In addition to the theoretical goal of understanding human thinking, cognitive science can have the practical goal of improving it, which requires normative reflection on what we want thinking to be [Stanford Encyclopedia of Philosophy, 2005].

Historically, cognitive science and AI have developed in parallel. In the mid 1950s, experimental psychology was dominated by behaviorism, a view that psychology should restrict itself to examining the relation between observable stimuli and observable behavioral responses. Internal

mental states and processes were completely neglected by behaviorists, which was a view that virtually denied the existence of the mind. Cognitive science appeared as a reaction to that position – its view was that mental states and processes actually mediate between input stimuli and output responses. It was suggested that information in the mind is encoded in *chunks*, mental representations that require mental procedures for encoding and decoding the information. From the very beginning, the mind–computer analogy became dominant in cognitive science:

The mind is to the brain as software is to hardware. Mental states and processes are like computer programs implemented in brain states and processes. [Rapaport, 2000].

The interdisciplinary nature of cognitive science is best reflected in its diverse research methodology. For cognitive psychologists, the primary method is *experimentation with human participants*. People (usually undergraduate students) are brought to the laboratory to study their mental processes (e.g., the mistakes they typically make in deductive reasoning, their performance when solving problems using analogies, the way they form and apply concepts, and so on) under controlled conditions. Another set of methods has been developed by researchers interested in *computational models* that simulate aspects of human performance: these researchers form and test models intended to be analogous to mental operations. The third kind of contribution to the rich methodology of cognitive science originated from linguists. Linguists are interested in *identifying grammatical principles* that provide the basic *structure of human languages*. They start from noticing subtle differences between grammatically correct and incorrect phrases. Neuroscientists' research is based on *observing what is happening in various parts of the brain* while people are doing various mental tasks. In their experiments with nonhuman subjects, neuroscientists can insert electrodes and record the firing of individual neurons. With humans, the methods are noninvasive and include various kinds of magnetic and positron-based scanning devices to observe what is happening in different parts of the brain while people are doing mental tasks such as mental imagery and word interpretation. Cognitive anthropologists examine human thinking to consider *how thought works in different cultural settings* [Stanford Encyclopedia of Philosophy, 2005].

It should be stressed that the philosophy of the mind is also important to cognitive science, although it does not have a distinct method. Philosophy shares its theoretical work with that in other fields, and is also concerned with empirical results. It deals with questions such as the nature of representation and computation, the relation between mind and body, and the nature of explanations.

The best work in cognitive science comes from combining methods from different disciplines. For example, cognitive psychology and AI can be combined through computational models of how people behave in experiments.

The best way to grasp the complexity of human thinking is to use multiple methods, especially psychological and neurological experiments and computational models. Theoretically, the most fertile approach has been to understand the mind in terms of representation and computation. [Stanford Encyclopedia of Philosophy, 2005]

Two of the most widely used computational paradigms for describing mental representations and the processes of the mind in cognitive science are the *symbolic* and *connectionist* paradigms. In the symbolic paradigm, the mind exists as a physically implemented “symbol system”, i.e., an implementation of an effectively computable procedure (a universal machine). An important hypothesis here is that a physical system is capable of intelligent behavior if and only if it is a physical symbol system. In other words, intelligent systems can manipulate symbols through a variety of mental states and procedures, and vice versa, physical symbol systems are capable of being intelligent systems.

The connectionist paradigm is inspired by physical neurons and their connections in the human brain. Physical neurons are modeled as specific interconnected data structures and processors – artificial neurons – and there are corresponding algorithms that model the firing of neurons and the spreading of activation in an *artificial neural network*. The aim is to propagate information along the links between the (large number of) neurons from the input to the output. The network can be trained to exhibit a desired intelligent behavior on the output, which is equivalent to the learning capability of the human brain. Technically, the learning of an artificial neural network is achieved by adjusting the weights associated with the links between the neurons. The point here is that intelligence is achieved without symbolic computation; it *emerges* over time, through the training procedure.

Combinations of the two major paradigms have also been explored, in two ways. One way is based on the fact that some cognitive processes are easy to implement symbolically (e.g., problem solving, reasoning, and game playing), whereas some others are more easily implemented using the connectionist approach (e.g., some aspects of visual perception and learning). The other way views connectionism as a lower level of cognitive processing, i.e. as a way of implementing symbolic processes [Rapaport, 2000]. For example, logical reasoning may be explained in terms of

“connectivity patterns” that correspond precisely to symbolic representations.

All these paradigms suggest that there is no single computational model of the mind. It is actually the mind–brain–computation triad that requires further exploration, where each member of the triad may be used to suggest new ideas about the others. Most of today’s computers are sequential processors, whereas the brain performs parallel processing of many operations at once.

Currently, there are six widely established cognitive theories about the nature of the representations and computations that can be used to explain how the mind works [Stanford Encyclopedia of Philosophy, 2005]: formal logic, rules, concepts, analogies, images, and neural connections.

- *Formal logic*. This theory hypothesizes that people make the inferences they do because they have mental representations similar to sentences in propositional and predicate logic, as well as deductive and inductive procedures that operate on those sentences. Many complex kinds of knowledge and many inferences can be understood in terms of applying deductive and inductive procedures to the sentences. For example, the rule of logic called *modus ponens* states that if A is true and “ A implies B ” is also known to be true, then B is true. Another example is the *resolution principle* of gathering specific information/evidence in order to determine the truth of an assertion or to prove some previously established goal. The assertion “ P is true” holds if it can be shown that “ $\neg P$ is true” cannot hold because it leads to a contradiction with the evidence. John Sowa [Sowa, 2000] generalizes the power of logic explicitly:

Everything that can be stated clearly and precisely in any natural language can be expressed in logic. There may be aspects of love, poetry, and jokes that are too elusive to state clearly. But anything that can be implemented on a digital computer in any programming language can be specified in logic. [Sowa, 2000]

- *Rules*. Much of human knowledge is naturally described in terms of rules of the form “IF ... THEN ...”. Moreover, much of human thinking and reasoning can be modeled by rule-based systems, because people have mental rules, and procedures for using those rules to search a space of possible solutions. They also have procedures for generating new rules. Applying these kinds of procedures produces intelligent behavior.
- *Concepts*. In cognitive science, concepts are viewed as sets of typical features (slots). Concepts correspond partially to the words in spoken and written language. Schemas and scripts are more complex than

concepts that correspond to words, but are also sets of features. Concepts usually belong to various kind-of and part-of hierarchies. Applying concepts in intelligent processing means obtaining an approximate match between concepts and things in the world, and using inheritance through kind-of hierarchies to infer about more general concepts.

- *Analogies.* People have verbal and visual representations of situations that can be used as cases or analogue in new situations. They also have retrieval, mapping, and adaptation processes that operate on those analogue and produce intelligent behavior. Computational models of analogical reasoning simulate how people retrieve, map, and apply source analogue to target situations.
- *Images.* People have visual images of objects, spaces, and situations, and are capable of processing and manipulating those images. An image is worth a thousand words, as the ancient Chinese said – images are often much more suitable in representations than lengthy verbal/textual descriptions. Computational image processing that supports human pictorial representations includes procedures such as scanning, rotation, focusing, zooming, and transformation.
- *Neural connections.* Some mental representations involve simple processing units of brain – neurons – linked to each other by excitatory and inhibitory connections. Some human neurophysiological processes spread activation between these units via their connections; some others modify the connections. Applying these processes produces various kinds of intelligent behavior, such as vision, decision making, explanation selection, and meaning making in language comprehension.

The central and appealing hypothesis of cognitive science, that the mind works by means of representations and computations, has been criticized as well. The critics complain that cognitive science neglects the importance of emotions, consciousness, and the physical environment in human thinking, as well as some social aspects of thought and the contribution of the body to thought and action. Some critics also doubt that the mind can be effectively described as a computational system in the standard sense, and suggest describing it as a dynamic system or system of quantum physics instead.

1.3 Types of Human Knowledge

Cognitive psychologists have identified a number of different types of knowledge that humans commonly use, and practical AI systems

implement some of these types using corresponding knowledge representation techniques. The existence of multiple types of knowledge in human minds indicates the capability of humans to organize their knowledge in a structured way and to use that knowledge to solve problems efficiently.

Table 1-1 summarizes the categories and types of knowledge that most humans typically use; nowadays, AI systems implement many of these types.

- *Procedural knowledge* is about *how* to do something, for example how a problem is solved. Some typical types of knowledge in this category are rules, problem-solving strategies, agendas and procedures.
- *Declarative knowledge* describes what is known about a topic or about a problem. For example, some statements of declarative knowledge may describe details of concepts and objects. Some other statements may express facts that are either true or false.

Table 1-1. Commonly used types of human knowledge (adapted from [Durkin, 1994])

Category	Explanation/type
Procedural knowledge	Rules Strategies Agendas Procedures
Declarative knowledge	Concepts Objects Facts
Metaknowledge	Knowledge about other types of knowledge and how to use them
Heuristic knowledge	Rules of thumb
Structural knowledge	Rule sets Concept relationships Concept-to-object relationships
Inexact and uncertain knowledge	Probabilities Uncertain facts, rules, relationships, and evidence Fuzzy sets and logic
Commonsense knowledge	Default propositions Approximate concepts and theories General hierarchies and analogies
Ontological knowledge	Concepts Relationships between concepts Axioms Constraints

- *Metaknowledge* is knowledge about knowledge. It is used to decide what other knowledge is best suited for solving the problem at hand, what other knowledge is irrelevant to the problem and should therefore not be considered at all, and how to direct the problem-solving process into the most promising areas of the solution space.
- *Heuristic knowledge* includes those rules of thumb that guide the problem-solving process on the basis of previous experience in solving problems, individual intuition and skills, and a good understanding of the problem. However, heuristic knowledge is not strict. Also, problem solving based on heuristic knowledge is not guaranteed to be successful (i.e., it is not guaranteed to find a solution to the problem). Heuristic knowledge is usually compiled empirically from fundamental, deep knowledge about the problem, and is expressed as simple heuristics that help guide the processes of solving the problem and moving through the solution space.
- *Structural knowledge* describes mental models and the organization of problems, solutions, and their respective spaces. Typically, relationships between different pieces of knowledge from other categories (such as kind-of, part-of, grouping, and sets) are described by structural knowledge.
- *Inexact and uncertain knowledge* characterizes problems, topics, and situations in which information is inherently imprecise, unavailable, incomplete, random, or ambiguous. Inexact information and knowledge is often described in terms of a priori, a posteriori, and conditional probabilities of events. Uncertain knowledge is associated with expressions about a subjective belief in the validity of the knowledge, as well as with fuzzy linguistic terms (such as little, warm, or high) rather than precise quantitative measures, and degrees of membership of sets ranging between 0 and 1 rather than strict true or false membership.
- *Commonsense knowledge* is the term used to denote a vast amount of human knowledge about the world which cannot be put easily in the form of precise theories [McCarthy, 1999]. In many situations, humans rely on this kind of knowledge when facing an imprecise or incomplete characterization of the problem they are trying to solve, or when they simply lack more appropriate knowledge. It is always difficult to know in advance what part of commonsense knowledge will become relevant in solving a problem. Also, the consequences of applying that knowledge cannot be fully determined. Applying commonsense knowledge requires using defaults, approximate concepts (concepts without clear definitions) and approximate theories (theories involving approximate concepts), as well as general hierarchies and analogies.

- *Ontological knowledge* is an essential supplement to knowledge about a specific domain – it describes the categories of things in that domain and the terms that people use to talk about them [Sowa, 2000]. For example, the ontological knowledge about an object-oriented language contains the concept of a class, and the ontological knowledge about a relational database includes tables and fields. Ontological knowledge also describes relations between such categories, and the axioms and constraints in the domain. Obviously, the types of ontological knowledge overlap with the types of the other categories of knowledge (declarative and structural knowledge). The important difference, however, is in the meanings of the various categories.

1.4 Knowledge Representation Techniques

There is no single “best” theory that explains all human knowledge organization. There is no single “best” technique for structuring data in all computer programs either. Likewise, there is no single ideal knowledge representation technique suitable for all applications. When building practical intelligent systems, developers should select the best knowledge representation technique to suit the application. To do this, they must have a good understanding of the various types of human knowledge and a palette of knowledge representation techniques to select from.

Note that the correspondence between the types of human knowledge discussed in the previous section and knowledge representation techniques is not necessarily one-to-one. Some techniques can be used to represent knowledge in two or more of the categories and types listed in Table 1-1. For example, rules of various forms, used as a knowledge representation technique, can represent parts of procedural knowledge, heuristic knowledge, metaknowledge, and inexact/uncertain knowledge. Likewise, facts, used as a declarative knowledge type, can be represented using several techniques – propositions, object–attribute–value triplets, multiple-valued facts, uncertain facts, etc.

The traditional AI techniques most frequently used to represent knowledge in practical intelligent systems include object–attribute–value triplets, uncertain facts, fuzzy facts, rules, semantic networks, and frames. Most of them are also important for understanding the rest of this book, and hence they are briefly surveyed in this section. More recently, ontologies have acquired major importance in knowledge representation as well. They are covered in detail from Chap. 2 onwards.

Simpler knowledge representation techniques can be used within more complex ones. For example, various techniques for representing facts can be used to describe parts of rules, frames, and semantic networks.

1.4.1 Object–Attribute–Value Triplets

Object–attribute–value (O–A–V) triplets) are a technique used to represent facts about objects and their attributes. More precisely, an O–A–V triplet asserts an attribute value of an object. For example, the English phrase “The color of the ball is yellow” can be written in O–A–V form as “Ball–color–yellow” and represented graphically as in Fig. 1-1.

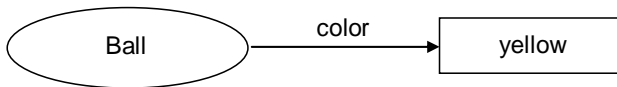


Fig. 1-1. An example of O–A–V triplet

Objects usually have more than one attribute, so in a graphical representation there may be multiple arrows (denoting attributes) and boxes (denoting attribute values) for each oval (representing an object). Some attributes may be single-valued as in the above example, whereas others may be multiple-valued (e.g., “Travel–purpose–business, pleasure”).

1.4.2 Uncertain Facts

A simple but frequently used extension of O–A–V triplets allows uncertainty of facts to be described. A *certainty factor* is a numeric value assigned to a statement that represents the degree of belief in the statement [Durkin, 1994]. Such numeric values are usually discrete and are taken from a previously specified interval (e.g., [0, 1], [–1, 1], [0–100], and the like). Discrete values usually have linguistic counterparts. For example, the certainty of a statement can be described using the following terms and their equivalent certainty factors: “definitely false” (–1.0), “probably false” (–0.5), “unknown” (0.0), “probably true” (0.5), and “definitely true”. The statement “The river is probably long” can be written as an uncertain O–A–V fact “River–length–long (CF = 0.5)”, and its graphical representation may look as in Fig. 1-2.

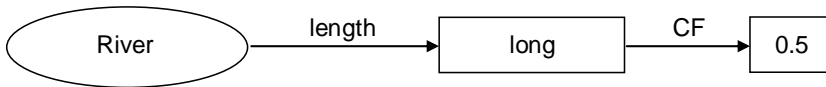


Fig. 1-2. An uncertain fact

The continuous interval of possible certainty factor values can be divided into continuous subintervals and each subinterval can be assigned a discrete value. More fine-grained subintervals, mean more variety in certainty factor values and more possibilities for expressing uncertainty precisely.

1.4.3 Fuzzy Facts

Fuzzy facts represent uncertainty using the imprecise and ambiguous terms commonly found in natural languages. For example, the statement “The person is old” may be interpreted differently by different people, who may have different understandings of the term “old”. The *fuzzy set* representing old people may look as in Fig. 1-3 and is defined by a corresponding *membership function*. This function can be used to calculate the actual numerical value of a membership in the fuzzy set, on a scale from 0 to 1.

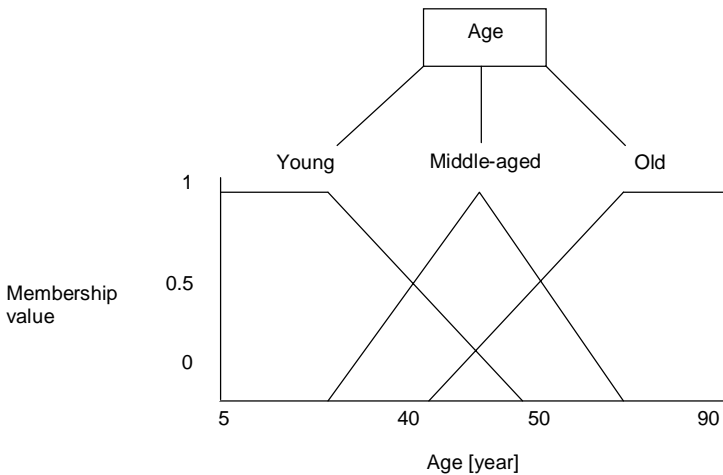


Fig. 1-3. Fuzzy sets on age

1.4.4 Rules

A rule is a knowledge representation technique and a structure that relates one or more *premises* (*conditions*, or *antecedents*), or a *situation*, to one or more *conclusions* (*consequents*), or *actions*. The premises are contained in the IF part of the rule, and the conclusions are contained in the THEN part, so that the conclusions may be inferred from the premises when the premises are true. For example:

IF	The time is after midnight
AND	I am hungry
THEN	I should not eat now

Premises are typically represented as facts, for example using O–A–V triplets. Actions can assert a new fact or can perform an operation (e.g., invoke a procedure).

Another interpretation of rules, premises, and conclusions is obtained from the state–transition view. There is a collection of states – situations that might occur in the application environment. A rule describes the logic of moving from a start state (a set of valid premises) to a goal state (described by the rule’s conclusions).

Uncertain rules may include certainty factors, both in the premises and in the conclusions, and imply uncertain inferences. For example:

IF	The time is after midnight (CF = 0.5)
AND	I am hungry
THEN	I should not eat now (CF = –0.5)

In uncertain rules, omitting an explicit value for a certainty factor in a premise means “take the previously inferred CF value” or “take the default CF value”. Conclusions made with uncertain premises cannot be absolutely true, but absolutely certain premises may lead to an uncertain conclusion.

Fuzzy rules contain fuzzy sets in their IF and THEN parts. They actually map fuzzy sets to one another:

IF	The meal is <i>hot</i>
THEN	I should wait a <i>little</i> before I eat it.

The knowledge of a practical intelligent system may be represented using a number of rules. In such a case, the rules are usually grouped into a hierarchy of *rule sets*, each set containing rules related to the same topic.

1.4.5 Semantic networks

Semantic networks (or concept maps) are a knowledge representation technique that attempts to reflect cognition [Barr & Feigenbaum, 1981; Rapaport, 2000]. Designed after the psychological model of the human associative memory, semantic networks are graphs made up of objects, concepts, and situations in some specific domain of knowledge (the nodes in the graph), connected by some type of relationship (the links/arcs). Figure 1-4 shows an example. The network represented expresses several inter-related facts: Billy *is a* Labrador, which is *a kind of* a dog. The other kinds of dogs shown are setters and bulldogs. A dog *has* legs, and the *number of* legs is 4.

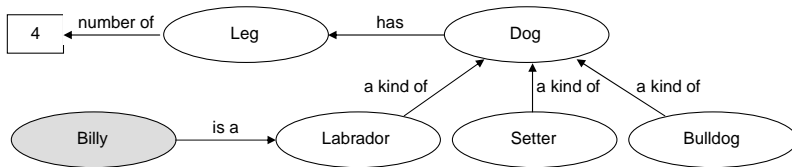


Fig. 1-4. A simple semantic network

Note that arcs can be labeled to denote an arbitrary relationship between concepts. However, kind-of, part-of, and is-a relationships are the most common. The graphical notation used to represent semantic networks is not standardized, but the things that can be typically represented in such networks are concepts (classes), their instances (objects), attributes of concepts, relationships between classes and objects, and values of attributes. As in object-oriented programming, objects/instances should normally have values for all attributes defined in the concept. There may be default values for some attributes (e.g., dogs normally have 4 legs). Kind-of relationships denote inheritance. Also, parts of semantic networks are easily recognized as O–A–V triplets (e.g., “Leg–number of–4”).

It is easy to insert new nodes, relationships, and values into a semantic network. The technique has proven to be intuitively clear and easy to learn, and is widely used by AI specialists and psychologists alike. However, semantic networks are known to be weak in representing and handling exceptions to the knowledge they represent. In large networks, all nodes must be carefully examined for exceptions.

1.4.6 Frames

A frame is a technique and a data structure for representing stereotypical knowledge of some concept or object. Frames are similar to classes and objects in object-oriented programming. There are *class frames*, which represent common features of a set of similar objects (e.g., *dog*, *student*, or *movie*). *Instance frames* represent specific instances of a class frame (e.g., instances of the above three class frames may be the instance frames Billy, Joan, and *The Legends of The Fall*, respectively).

Each frame is easy to visualize as a form with various fields to be filled. The fields correspond to *slots* (or *properties*) of the frame. A typical slot of a frame is its *name*. Instance frames also typically have a corresponding *class* slot. The other slots depend on the nature of the frame itself.

Instance frames inherit all slots of the corresponding class frame. However, an instance frame may change the values of some slots or even add new slots (and their values). There can be many instance frames corresponding to one class frame.

In fact, frames are tightly related to semantic networks. An important difference between the two techniques is that, unlike semantic networks, frames include procedural knowledge as well, in the form of *facets*. Facets are typically attached to slots and contain procedures (methods, or demons) that are invoked automatically when the value of the slot is changed, or when it is needed (read). For example, an *if-changed* method may be invoked for each slot for which it is necessary to check the numeric value that is to be written into the slot. Likewise, an *if-needed* method may be called to supply the slot's value from a database or from a spreadsheet.

Another knowledge representation technique related to frames is that of *scripts*, which describe activities involving the corresponding knowledge, and also supply possible outcomes and scenarios.

1.5 Knowledge Representation Languages

The central component of any knowledge-based intelligent system is its *knowledge base*. The knowledge base contains a set of *sentences* – the units of the knowledge represented using one or more knowledge representation techniques, i.e., assertions about the world [Russell & Norvig, 2002]. The sentences are expressed in a knowledge representation language.

New sentences can be added to the knowledge base, in order for the information and solutions provided by the system to be both accurate and

complete. Queries can be made to the knowledge base to obtain what the system currently knows about the world. Functions that implement these actions must rely on the details of the knowledge representation language.

Knowledge representation languages should be capable of both syntactic and semantic representation of entities, events, actions, processes, and time. However, not all of the existing knowledge representation languages have support for all of these things. Also, each existing language supports some, but not all, popular knowledge representation techniques. In addition, some knowledge representation languages are designed to provide support for knowledge communication and interchange between intelligent systems.

The dividing line between knowledge representation languages and programming languages is not strict. One may argue that traditional AI programming languages such as Lisp and Prolog support knowledge representation as well. Unlike popular programming languages such as C and Java, AI languages provide some built-in mechanisms for deriving facts from other facts, which is important for representing knowledge (as opposed to representing data). Others may argue that it is necessary for a knowledge representation language to have explicit support (in terms of data structures and language constructs) for knowledge representation techniques such as rules, frames, scripts, and semantic networks.

Ideally, a successful knowledge representation language should both be *expressive* enough to let the knowledge base developer structure and encode all the necessary knowledge easily, and make the represented knowledge *understandable* by humans. The language and the knowledge represented in it should cause a system using that knowledge to *behave* as if it knows it.

The subsections that follow briefly overview important design paradigms for knowledge representation languages, deliberately leaving out ontology representation languages, which are covered in the later chapters.

1.5.1 Logic-Based Representation Languages

Much knowledge representation can be done in a logic-based language (or “logic language” for simplicity). Moreover, many of the other frequently used knowledge representation paradigms are built on top of formal logics. Note the plural here – there are actually many formal logics. They partially overlap, but are also different in terms of how much they can express and in terms of notational and other details.

Both historically and pragmatically, logic-based languages are an important branch of AI.

Rationale, Pros and Cons

The popularity of formal logics as the basis for knowledge representation languages arises for practical reasons. They are all formally well founded and are suitable for machine implementation. Also, every formal logic has a clearly defined syntax that determines how sentences are built in the language, a semantics that determines the meanings of sentences, and an inference procedure that determines the sentences that can be derived from other sentences. Furthermore, most logic languages are designed to be precise and to avoid the incomplete and ambiguous expressions frequently found in natural languages. Even fuzzy logic has this property, in spite of the fact that it is suitable for representing ambiguity arising from linguistic terms.

On the other hand, all sentences in logics are assertions. Reasoning based on formal logics is limited to deriving truth values and proofs for such assertions. In formal logics, it is difficult to model interrogations and those kinds of human reasoning that involve assumption, likelihood, belief, disbelief, doubt, desires, etc. Also, designing a complete valid inference procedure is a very complex task for many logics.

Nevertheless, precise and formally clearly expressed assertions are a good starting point. It may be the case that the syntax and semantics of a logic are quite suitable for a certain application, and that only the reasoning needs to be specifically designed for that particular application. Some may complain that designing and developing a reasoner is not an easy task, but the alternatives (e.g., frame-based and rule-based languages) have proven not to be perfect either. Moreover, since logic-based roots can often be traced in many other formalisms, the shortcomings of logic cannot be eliminated easily. Using something completely different instead, such as neural networks, may only reverse the problem – things that are very easy to do with logic languages can be very hard to do with another formalism.

Note, however, that agreement to use a logic language does not solve other knowledge base development problems automatically. For example, setting up the vocabulary for building the sentences in the knowledge base can be a time-consuming task.

Propositional Logic

A proposition is a logical statement that is either true or false. For example, “The princess is in the palace.” An O–A–V triplet is a more complex form of proposition, since it has three distinct parts.

Propositional logic is a form of symbolic reasoning. It assigns a symbolic variable to a proposition, for example

A = The princess is in the palace

The truth value (*true* or *false*) of the variable represents the truth of the corresponding statement (the proposition). Propositions can be linked by logical operators (AND (\wedge), OR (\vee), NOT (\neg), IMPLIES (\rightarrow or \Rightarrow), and EQUIVALENCE (\Leftrightarrow)) to form more complex statements and rules:

IF	The princess is in the palace	(A)
AND	The king is in the garden	(B)
THEN	The king cannot see the princess	(C)

The symbolic representation of the above rule in propositional logic is

$A \wedge B \rightarrow C$

Thus, propositional logic allows formal and symbolic reasoning with rules, by deriving truth values of propositions using logical operators and variables. Finding the truth value of an arbitrary statement is a matter of examining the truth values of the variables and the truth tables of the logical operators in component parts of the statement. In fact, this procedure means checking the statement’s validity or satisfiability – the proof theory of propositional logic. Deduction systems based on propositional logic are syntax-based (i.e., the actual meaning of the symbols is unimportant in the reasoning process; what matters is the formal proof). A deduction system for propositional logic is called a propositional calculus.

However, this simple form of reasoning may be inadequate or too mechanistic. In many situations, it is difficult to assign a variable to a whole statement. Also, the truth value of a statement can often be subjective and prone to interpretation, and is generally computationally exponential. First-order logic offers a more fine-grained approach, in which parts of statements can be represented and reasoned about logically [Durkin, 1994; Russell & Norvig, 2002; Sowa, 2000].

First-Order Logic (First-Order Predicate Calculus)

First-order logic extends propositional logic by introducing the *universal quantifier*, \forall , and the *existential quantifier*, \exists (see below for details). It also uses symbols to represent knowledge and logical operators to construct statements. Its symbols may represent constants, variables, predicates, and functions.

Constants are symbols that begin with a lowercase letter. *Variables* are symbols that begin with an uppercase letter. Propositions are represented using *arguments* – objects of propositions – and *predicates* – assertions about objects. For example, the proposition “The princess is in the palace” would be written in first-order logic as

$\text{in}(\text{princess}, \text{palace})$

The predicate *in* can be used with variables to write more general propositions that capture various special cases (instantiations):

$\text{in}(X, Y)$

The above statement can be instantiated by propositions such as “The princess is in the palace” and “The king is in the garden”:

$\text{in}(\text{princess}, \text{palace})$

$\text{in}(\text{king}, \text{garden})$

A *function* is a mapping from entities of a set to a unique element of another set:

$\text{father}(\text{princess}) = \text{king}$

Using predicates, functions, and logical operators, it is possible to specify rules:

$\text{in}(\text{princess}, \text{palace}) \wedge \text{in}(\text{king}, \text{garden}) \rightarrow \neg \text{see}(\text{king}, \text{princess})$

The universal quantifier \forall and the existential quantifier \exists , in first-order logic, denote the range or scope of variables in statements. The combination “ $\forall x$ ” is interpreted as “for each x ” or “for all x ”. “ $\exists x$ ” means “there exists an x such that”. For example,

$\exists X \text{ see}(X, \text{princess})$ “Someone sees the princess”

$\forall X \text{ see}(X, \text{princess})$ “Everyone sees the princess”

The range of quantifiers in first-order logic is restricted to simple, unanalyzable individuals, hence the name “first-order”. In *higher-order logic*, function variables and predicate variables can be governed by quantifiers [Sowa, 2000].

Knowledge represented with first-order logic is written in the form of predicates and rules. Reasoning with first-order logic is performed using predicates, rules, and general rules of inference (such as modus ponens and resolution) to derive conclusions.

First-order logic is simple – it is two-valued logic with just two quantifiers and the basic Boolean operators. However, used as a knowledge representation language, it forms a sound basis upon which many other AI languages are built (e.g., Prolog). It is like an assembly language for knowledge representation [Durkin, 1994]. Higher-order logic, modal logic, fuzzy logic, and even neural networks can all be defined in first-order logic. In other words, it can be used as a metalanguage for defining other kinds of logic, and even more:

First-order logic has enough expressive power to define all of mathematics, every digital computer that has ever been built, and the semantics of every version of logic including itself. [Sowa, 2000]

Knowledge Interchange Format (KIF)

KIF is a particular logic language based on first-order logic, specifically designed to make it useful as a mediator in the translation of other languages – an “interlingua” [Genesereth & Fikes, 1992]. The idea is to have translators from other languages to KIF and to other languages from KIF. If it is necessary to perform a translation from a representation described in a language A to an equivalent representation in another language B, it requires only A-to-KIF and KIF-to-B translators; KIF is used as an intermediate language.

The description of KIF includes specifications of both its syntax and its semantics. The semantics is similar to that of first-order logic. KIF uses the prefix notation of first-order logic with extensions to support nonmonotonic reasoning and definitions. The following simple examples illustrate the syntax of KIF:

(forall ((?x P)) (Q ?x))	;"All P are B"
(exist ((?x P)) (not (Q ?x)))	;"Some P is not Q"
(president 802 thomas jefferson)	; a record in the database ;of presidents
(president 345 abraham lincoln)	; another record in the database of ; presidents

Another important extension of KIF with respect to first-order logic is the use of the backquote (`) and comma (,) operators, which enable the encoding of knowledge about knowledge. For example,

(notes mary `(president ,?x ,?y ,?z))

The above sentence represents the knowledge that Mary notes the sentences about presidents. The backquote operator is applied to the inner sentence, which is a knowledge statement itself; the entire inner statement is used as an argument in the outer statement. The question marks denote variables, but the comma operator applied to the variables ?x, ?y, and ?z specifies that they should not be taken literally. In other words, the commas specify that Mary notes instances of the sentence (president ?x ?y ?z), not the sentence itself.

KIF can be also used to describe functions, sets, and many other concepts.

Conceptual Graphs [Sowa, 2000] provide a suitable graphical notation for logic. They are based on existential graphs, augmented with features from linguistics and semantic networks. As a graphical notation, they are of interest for KIF because they have expressive power identical to that of KIF, and anything stated in one can be automatically translated to the other.

Description Logics

An entire group of languages for knowledge representation, based on description logics, is of particular interest in the context of ontology representation and development (Chap. 2 and onwards). The semantics of the underlying concepts of these languages identifies them as decidable fragments of first-order logic.

Developing a knowledge base using a description logic language means setting up a *terminology* (the vocabulary of the application domain) in a part of the knowledge base called the *TBox*, and *assertions* about named individuals (using the vocabulary from the TBox) in a part of the knowledge base called the *ABox*. The vocabulary consists of concepts and roles. *Concepts* denote sets of *individuals*. *Roles* are binary relationships between individuals. There are *atomic* concepts and roles (names of concepts and roles) and *complex* concepts and roles (terms for concepts and roles). The complex concepts are built using descriptions expressed in the corresponding description logic language and are assigned names in the TBox. For example, if *Person* and *Female* are atomic concepts and *hasChild* is an atomic role (a relation), part of a TBox defining complex concepts about family relationships may look like this (adapted from [Baader et al., 2003]):

Woman \equiv Person \sqcap Female

Man \equiv Person \sqcap \neg Woman

Mother \equiv Woman \sqcap \exists hasChild.Person

Father \equiv Man \sqcap \exists hasChild.Person

Parent \equiv Mother \sqcup Father

Grandmother \equiv Mother \sqcap \exists hasChild.Parent

MotherWithManyChildren \equiv Mother \sqcap ≥ 3 hasChild

MotherWithoutDaughter \equiv Mother \sqcap \forall hasChild. \neg Woman

Wife \equiv Woman \sqcap \exists hasHusband.Man

Husband \equiv Man \sqcap \exists hasWife.Woman

In the above concept definitions, each symbol of the form $\exists R.C$ or $\forall R.C$ denotes the set of those individuals from the set corresponding to the concept C for which the relation (role) R holds. The symbols \sqcap and \sqcup denote intersection and union, respectively. ≥ 3 hasChild is a value restriction on the cardinality of the corresponding set (in this case, the minimum; the set includes all those instances of the atomic role hasChild in the domain of interpretation for which the value restriction holds).

In fact, the TBox defines (in a general way) semantic relationships between individuals introduced in the ABox and their properties. In other words, the ABox describes a specific state of affairs in the world in terms of the concepts and roles defined in the TBox. An ABox corresponding to the above TBox might be

MotherWithoutDaughter(JANE)
hasChild(JANE, CHARLES)
Father(ABRAHAM)
hasChild(ABRAHAM, ISAAC)

An intelligent system with a knowledge base structured as a TBox–ABox pair can reason about its terminology and assertions. It can determine whether a description in the TBox is *satisfiable* (i.e., noncontradictory), and whether there is a *subsumption* relationship between two descriptions (i.e., one is more general than the other). For example, in the above TBox *Person* subsumes *Woman*, and both *Parent* and *Woman* subsume *Mother*. Two important reasoning tasks about assertions in the ABox are to determine whether the assertions imply that a particular individual is an *instance* of a given concept description, and whether a set of assertions is *consistent* (whether it has a model). From a pragmatic point of view, consistency checks of sets of assertions and satisfiability checks of concept descriptions help to determine whether a knowledge base is meaningful at all. Subsumption tests help to organize

the terminology into a meaningful hierarchy of concepts according to their generality. Each test can be also interpreted as a query about objects of interest; it retrieves the set of individuals that satisfies the test.

A subclass of description logic languages allows *rules* to be used in addition to the TBox and ABox to describe knowledge. The rules are of the operational form $C \Rightarrow D$ (“if an individual is an instance of C , then derive that it is also an instance of D ”), or of the declarative form $\mathbf{K}C \sqsubseteq D$ (the *inclusion axiom* form; the operator \mathbf{K} restricts the application of the axiom to those individuals that appear in the ABox and for which the ABox and TBox imply that they are instances of C). Such rules can be used to add assertions to the ABox in an operational environment (an intelligent system) containing a knowledge base. These rules can be interpreted logically. Alternatively, the environment may provide an application program interface (API) with appropriate functions through which programs can operate on the knowledge base in terms of adding, retracting, and modifying concepts, roles, and assertions.

1.5.2 Frame-Based Representation Languages

In all frame-based representation languages (or, for simplicity, “frame representation languages”), the central tenet is a notation based on the specification of frames (concepts and classes), their instances (objects and individuals), their properties, and their relationships to each other [Welty, 1996]. This object-orientedness is human-centered. For this reason, frame-based languages are favored by many over logic languages as more appealing and more effective from a practical viewpoint [Baader et al., 2003].

In addition to object-orientedness, frame-based languages also provide for expressing generalization/specialization, i.e., organizing concepts into hierarchies. They also enable reasoning, by making it possible to state in a formal way that the existence of some piece of knowledge implies the existence of some other, previously unknown piece of knowledge. With frame-based languages, it is possible to make classifications – concepts are defined in an abstract way and objects can be tested to see whether they fit such abstract descriptions.

To get a feeling for the nature and syntax of frame-based languages, consider following piece of code, in the KM language [Clark & Porter, 2004]:

```
(every Buy has
  (buyer ((a Agent)))
```

```
(object ((a Thing)))  
(seller ((a Agent)))
```

The above code partially defines the class *Buy*, saying something like “All buy events have a buyer and a seller (both of type *Agent*), and an object (of type *Thing*) which is bought.” In fact, it defines only the properties of the instances of the class, and not the properties of the class itself (things such as the parent class(es), the words or phrases used to denote objects of the class, and so forth). The latter may be defined as follows:

```
(Buy has  
  (superclasses (Transaction))  
  (terms (“buy” “purchase”)))
```

In KM, both of the above code snippets are referred to as frames. A KM class typically needs two frames to describe it, one for the properties of the class, and another for the properties of the members of the class.

The traditional frame-based languages lacked precise characterization of their semantics. Moreover, discrimination between different types of knowledge embedded in a frame system was not clear. As a result, every frame-based system behaved differently from the others (which were developed using different frame-based languages), in many cases despite virtually identical-looking components and even identical relationship names [Baader et al., 2003]. As a consequence, different systems could not interoperate and share knowledge. Later frame languages introduced more formal semantics, retaining the hierarchical concept structures and ease of representation and simultaneously improving the efficiency of reasoning and the representational rigor. For example, the KL-ONE system of Brachman and Schmoltze [Brachman & Schmoltze, 1985] introduced description logic formalism into representing frames. KL-ONE not only greatly improved the representational rigor of frame-based languages, but also spawned an entire new class of frame systems. CLASSIC [Brachman et al., 1991] was one of them. The following example is the CLASSIC encoding of two concepts (*JAPANESE-CAR-MAKE* and *RICH-KID*) and one role (*thing-driven*):

```
define-concept[JAPANESE-CAR-MAKER,  
  (ONE-OF Mazda Toyota Honda)];  
define-role[thing-driven];  
define-concept[RICH-KID,  
  (AND STUDENT  
    (ALL thing-driven SPORTS-CAR)  
    (AT-LEAST 2 thing-driven))]
```

The first two definitions in the above example have obvious meanings. In the third one, the *ALL* construct is used to describe any object that is related by the *thing-driven* role solely to individuals that can be described by the concept *SPORTS-CAR* (“something all of whose *thing-driven* are *SPORTS-CARS*”). The third definition also uses the *AND* construct to make the composition of the three concept expressions that follow it. So, *RICH-KID* is defined to be “a *STUDENT* who drives *AT-LEAST 2* things, *ALL* of which are *SPORTS-CARS*”. The correspondence between *CLASSIC* constructs and operators and constructs from the underlying description logics is easy to notice (e.g., *AND* corresponds to \sqcap , *ALL* to \forall , and (*ALL thing-driven SPORTS-CAR*) to \forall thing-driven.SPORTS-CAR).

Note, however, that most frame languages are just a new syntax for parts of first-order logic [Welty, 1996]. This means that most of these languages do not offer anything new in expressiveness. For instance, the KM code shown in the example about buys, buyers, and sellers can be represented equivalently in KIF as

```
(forall (?b)
  (exists (?a) (=> (isa ?b Buy) (and (buyer ?b ?a) (isa ?a Agent))))
(forall (?b)
  (exists (?a) (=> (isa ?b Buy) (and (object ?b ?a) (isa ?a Thing))))
(forall (?b)
  (exists (?a) (=> (isa ?b Buy) (and (seller ?b ?a) (isa ?a Agent))))
```

Frame-based languages are usually suitable for representing knowledge that does not change. Only very few of them are able to deal with nonmonotonic changes in the knowledge being represented.

Another disadvantage of frame-based languages is inadequate way in which they deal with procedural knowledge. The way they do this is usually to use some limited arithmetic, plus calling procedures and functions written in a procedural language and attached to slot facets. The procedural knowledge encoded in this other language is not represented in a frame-based way – it is hard-coded in the corresponding function/procedure. As a consequence, the resulting systems can only reason *with* that knowledge, but not *about* it.

1.5.3 Rule-Based Representation Languages

Rule-based representation languages (or, simply, “rule languages”) are popular in commercial AI applications, such as expert systems [Durkin, 1994]. The rule-based paradigm is easy to understand, and there are a number of practical tools for building rule-based systems. Every rule-

based language has an appropriate syntax for representing the If–Then structure of rules. On the other hand, there are considerable syntactic differences between different languages in terms of how they actually represent If–Then rules.

Vianu [Vianu, 1997] notes that there are two broad categories of rule-based languages: *declarative (model-theoretic)* languages, which attempt to provide declarative semantics for programs, and *production system* languages, which provide procedural semantics based on forward chaining of rules. The criteria for other possible classifications of rule-based languages may be related to what is allowed in the rules. Some languages allow only the assertion of new facts in the knowledge base in the Then part of the rules. Other languages allow facts to be retracted from the knowledge base as an effect of rule firing. Yet another group of languages allows the embedding and/or invocation of procedural code (written in another, procedural language) in the rule syntax. Rule languages also differ in terms of to what extent they control rule ordering during the inference process. Further classifications of rule languages are possible.

However, systems built using traditional rule languages have been shown to have two important limitations [Welty, 1996]:

- they are quite restrictive when applied to large problems (i.e., when the rule base is large), as there is no ordering of the rules;
- inferences cannot be limited to those dealing only with the objects of interest.

More recent rule languages combine rules with frame-based features that provide object orientation, better structuring of knowledge through generalization/specialization and concept hierarchies, and natural inference capabilities such as classification and inheritance. The synergy between rule-based languages and frame-based languages brings about an important advantage: information about a specific concept can be stored together with that concept, as opposed to rule-based systems where information about one concept may be scattered throughout the rule base. For example, a set of rules can be stored with a frame slot and activated through an if-needed demon whenever the slot value is read. Alternatively, a set of rules may be written to reason about a certain concept (frame) in detail.

As an example of a language that combines rules and frames, consider the following code, written in the native language of Jess, a popular expert-system development tool [Friedman-Hill, 2003]:

```
(deftemplate person (slot age))
...
(defrule check-age
```

```
(person (age ?x))
(test (> ?x 30))
=>
(printout t ?x "is over 30!" crlf)
```

This example defines a frame (*person*) using the *deftemplate* construct, and a rule (*check-age*) using the *defrule* construct. The rule checks to see if an instance of the *person* frame has a value of the *age* slot greater than 30; if so, an appropriate print action is taken.

The rule-based representation formalism is recognized as an important topic not only in AI, but also in many other branches of computing. This is especially true for Web engineering. Rules are one of the core design issues for future Web development, and are considered central to the task of document generation from a central XML repository. Much of the interoperability between different applications in e-commerce also relies on rules and needs a Web interchange format. Rules can be also used for declarative specification of Web services.

In response to such practical demands from the world of the Web, the *Rule Markup Initiative* has taken steps towards defining *RuleML*, a shared *Rule Markup Language* [RuleML, 2005]. RuleML enables the encoding of various kinds of rules in XML for deduction, rewriting, and further inferential–transformational tasks. The Rule Markup Initiative’s approach to rule categorization is more detailed, more fine-grained, and more practically oriented than the categorization by Vianu described above [RuleML, 2005]. The rule categories currently considered by the Rule Markup Initiative include [Wagner et al., 2003]

- production rules – the If–Then rules commonly found in numerous AI systems, such as expert systems;
- integrity rules (or integrity constraints) – assertions that must hold in all evolving states of a discrete dynamic system;
- reaction rules – rules that observe triggering events, check conditions in the forward direction in a natural fashion, perform an action only when and if all the necessary events/conditions are observed/met, and may also include an optional postcondition (in other words, rules specifying the reactive behavior of a system in response to triggering events);
- derivation rules – rules with one or more conditions and a conclusion, which imply using a procedure to derive the conclusion from the condition(s) (e.g., rules for defining derived concepts);
- transformation rules – transformation constructs that consist of a transformation invoker, a condition, and a transformation return (e.g., conditional directed equations or term-rewriting rules).

An example rule in English may state that “if someone likes music, then John likes that guy”. Represented in RuleML, as an adaptation of a similar example from [Boley et al., 2001], this rule may look like this:

```
<rulebase label="likesMusic">
  <imp>
    <_head><atom>
      <rel>likes</rel>
      <ind>John</ind>
      <var>x</var>
    </atom></_head>
    <_body><atom>
      <rel>likes</rel>
      <var>x</var>
      <ind>music</ind>
    </atom></_body>
  </imp>
</rulebase>
```

Without going into details of RuleML tagging, we recall that such an XML-based representation can be translated into another XML-based format using an appropriate XSL transformation (XSLT). For example, the above RuleML rule can be translated into an equivalent Jess rule using the RuleML-to-Jess XSLT:

```
(defrule likesMusic
  (likes ?x music)
=>
  (likes John ?x))
```

The Rule Markup Initiative now covers a number of new developments, including Java-based rule engines, an RDF-only version of RuleML, and MOF-RuleML (an abstract syntax of RuleML considered as a MOF model). See Chaps. 3 and 4 for a detailed explanation and discussion of RDF (Resource Description Framework) and MOF (Meta Object Facility).

1.5.4 Visual Languages for Knowledge Representation

The use of visual languages for knowledge representations is compelling for many reasons [Kremer, 1998]. The most obvious one is that the domain may require representation in a way that is not possible with purely textual or symbolic languages. Even if another formal knowledge representation language is available and suitable, in preliminary knowledge acquisition

the appropriate formal structure for that language may not be apparent. Domain experts may have difficulties in communicating and articulating their domain knowledge in terms of formal knowledge structures; hence knowledge engineers may want to use visual representations for easier communication with the experts and for relaxation of formal discipline. Also, the syntax of logic, frame, and rule languages may often be complicated and may require knowledge of tiny details which may be quite unappealing even to knowledge engineers.

Visual languages for knowledge representation may be based on various kinds of graphs, forms (query by example), purely spatial relationships (iconic sentences), matrices (spreadsheets), and simple text layouts (outlining tools). Most of them facilitate knowledge representation by at least partially eliminating the need to use the rigorous syntax of symbolic or text-based languages. Under the surface, they often transform the represented knowledge into another language (such as first-order logic).

Note that many of the graphical user interfaces (GUIs) of knowledge acquisition and representation tools serve the purpose of collecting and transforming user input into the knowledge base. However, calling a GUI a visual language may provoke objections, especially in the context of visual languages for knowledge representation. In many cases, the GUI is there just to hide the details of the underlying representation language. On the other hand, this role of the GUI proves to be very useful in knowledge engineering. For example, JessGUI is a forms-based GUI that transforms a knowledge engineer's input into Jess syntax, thus making it possible for the knowledge engineer to avoid learning the Jess language [Jovanović et al., 2004].

In a more narrow sense of the term, visual languages rely on two- or three-dimensional graphics and always involve pictures of some sort, typically nodes and connecting arcs [Kremer, 1998]; text is involved as well, but for labeling and annotation purposes. However, more importantly, in order for visual languages to be interpreted reliably, it is necessary to specify their syntax and semantics formally. As with all other formal languages, such a specification involves precise definition of:

- terminal and nonterminal symbols;
- productions (derivation rules), i.e., the grammar;
- unambiguous semantics.

Specifying a visual language in this way must reflect its primary intent – to render knowledge into a form amenable to computational support, i.e., to straightforward, unambiguous interpretation by a computer program.

Myers [Myers, 1990] has provided a comprehensive taxonomy of all visual languages, including those for knowledge representation. It includes several categories of visual languages for knowledge representation and shows that all of them can be subsumed by the general notion of *concept maps*, i.e., semantic networks. For example, visual languages such as Petri nets and flowcharts can also be interpreted as concept maps. The common feature of all categories of visual languages for knowledge representation is that they may be described in terms of relational grammars. As a consequence, all visual languages for knowledge representation used in practice fall into the category of concept maps, or have concept map languages which implement them.

All concept map languages provide graphical elements, used to represent typed *nodes* and typed *arcs*, as well as their labels. Other visual indicators may be used to distinguish between the nodes and arcs of different types (e.g., different shapes, colors, and line types). In fact, concept maps implement a simple graph theory. Frequently used extensions include the implementation of *partitions* or *contexts*, usually in the form of a box drawn around a subgraph.

Like frames, concept maps are object-oriented, human-centered, and easy to comprehend and use. The way they enable the structuring of a body of knowledge is much more salient than other forms of knowledge representation, such as pure text and logic [Kremer, 1998]. They can be used both at an informal level, such as for “brainstorming”, and at a very formal level, such as for representing knowledge in the knowledge base of an intelligent system.

Well-known examples of visual languages for knowledge representation include KRS [Gaines, 1991], which is the visual counterpart of the CLASSIC frame language, and Conceptual Graphs [Sowa, 2000]. Conceptual Graphs are more elaborate, more complex, more accurate, more expressive, more detailed, and more fine-grained than KRS. Although originally intended to represent natural-language expressions in a graphical form, they can actually represent many more forms of knowledge, including first-order logic (recall from Sect. 1.5.1 that Conceptual Graphs actually represent a visual counterpart of logic). It is of particular importance that Conceptual Graphs also have their “linear form”, i.e., a pure text representation, evolved for ease of use by computer programs. For the sake of completeness, this feature of Conceptual Graphs should be noted here as an analogue of other graphical–text representation pairs, such as that in RDF, and that provided by the Unified Modeling Language (UML) tools that enable serialization of UML models to an XML representation using XML Metadata Interchange (XMI). Chapters 4 and 5 cover these topics in detail. In fact, much of this book argues that

UML itself is an extremely suitable visual language for knowledge representation, even though it was originally developed as a software engineering tool.

1.5.5 Natural Languages and Knowledge Representation

Natural languages can express almost everything related to human experience, and hence they can be conceived of as knowledge representation languages as well. Natural languages are very expressive, and are also suitably declarative for knowledge representation. Moreover, they can be thought of as metalanguages for explaining both other natural and artificial languages (such as programming languages, other knowledge representation languages, and all other formal languages) and themselves [Sowa, 2000].

However, the use of natural languages for knowledge representation in AI is very restricted, owing to the fact that they are extremely complex for machine processing. Parsing sentences in human languages successfully is one of the most difficult computing problems, and still remains a challenge. Even more important and more difficult is the problem of machine understanding of the meaning of natural languages.

On the other hand, there are several examples of research, experimentation, and practical developments in describing knowledge *about* natural languages using knowledge representation techniques. Such efforts are of high importance for ontological engineering (see Chap. 2 for a detailed explanation of ontological engineering). For example, DATR is a language designed for lexical knowledge representation, i.e. for representing knowledge about words taken from natural languages [Evans & Gazdar, 1996]. More precisely, DATR represents knowledge about *lexemes*, the fundamental units of the lexicon of a natural language; for instance, “bind”, “binds”, “bound”, and “binding” are forms of the English lexeme “bind”. In DATR, lexical knowledge is encoded in a rather “minimalist” semantic network of nodes, each node representing a word, a lexeme, or a class of lexemes. A set of attributes and their values is encoded with each node and represents the links. For example, the node describing the lexeme “bind” may be described by the following attribute–value pairs: *syntactic category* – *verb*; *syntactic type* – *main* (i.e., it is a main verb, not an auxiliary one); *syntactic form* – *present participle*; *morphological form* – *bind-ing*. In addition to such nodes and links, DATR has a number of constructs that are impossible to visualize in terms of simple inheritance hierarchies of semantic networks. Nevertheless, the semantics of DATR is defined in an explicit, mathematically rigorous way.

Another interesting approach is that of using *controlled languages* for knowledge representation [Pulman, 1996]. A controlled language is a restricted form of a natural language that can be systematically related to the underlying knowledge representation language of an intelligent system. The restrictions apply to both the vocabulary and the syntax of the natural language, and produce output that can be analyzed and processed both as a formal language and as a restricted natural language. The idea of using a controlled language for knowledge representation is that it may be easier to represent human knowledge in computers in two steps, rather than trying to encode it in a formal symbolic language directly from its natural-language form. The first step is the transformation from a natural language to its appropriate controlled-language counterpart. In the second step, the controlled-language sentences are transformed into sentences of a more formal knowledge representation language (such as first-order logic, possibly augmented with some higher-order constructs).

For example, the English rule “If John has 30 dollars, and he likes going to restaurants, then he will go out for dinner” may be represented in a controlled English as follows:

CONDITION:

P has Q dollars

P likes going to restaurants

ACTION

P will go out for dinner

It is comparatively easy even for people without knowledge-engineering skills to express their knowledge in this way, and it is then rather straightforward to translate it to, for example, first-order logic.

Note that controlled languages are different from canned text and macro-style templates such as “<person> has <number> dollars”, which involve replacing the variable parts with specific words at run time. Unlike controlled languages, templates do not support the interactive knowledge refinement that some types of applications may require, and are rigid, overly simplistic, mechanistic, and also prone to representational infidelity during processing.

1.6 Knowledge Engineering

In order to develop practical knowledge bases, it is necessary to acquire human knowledge (from human experts or from other sources), to understand it properly, to transform it into a form suitable for applying

various knowledge representation formalisms, to encode it in the knowledge base using appropriate representation techniques, languages, and tools, to verify and validate it by running the practical intelligent system that relies on it, and to maintain it over time. These activities are parts of the process called *knowledge engineering*, and the people who perform them are *knowledge engineers* [Barr & Feigenbaum, 1981; Durkin, 1994; Friedman-Hill, 2003; Russell & Norvig, 2002].

It is important to understand that knowledge-engineering activities never occur linearly, in sequence. On the contrary, they are all interwoven and require rigorous discipline. It is not uncommon for knowledge engineers to return many times to alternative knowledge representation techniques and tools if they realize that those they are currently using are not suitable. Unfortunately, it is not always easy to tell suitable from unsuitable techniques and tools before the knowledge has been at least partially encoded, i.e., before at least one cycle of the above time-consuming activities has been completed. Minimizing the number of cycles depends on many factors, including the structure of the problem, the skills and experience of the knowledge engineer(s), the representation formalisms and tools used, and, most importantly, how successful the communication is between the knowledge engineer(s) and the human expert(s) in the process of knowledge acquisition.

Knowledge acquisition from human experts is difficult for several reasons, some of which are:

- cognitive mismatch between the expert and the knowledge engineer – they often perceive, present, explain, and prioritize things differently;
- knowledge transformation problems – there are always difficulties in representing the knowledge that the expert has supplied with the available tools and techniques, no matter how proven and/or sophisticated they are;
- consistency maintenance – in order to maintain high quality and consistency in the knowledge represented, it is necessary to update the knowledge base whenever changes in the domain occur or new knowledge is revealed.

There is a whole range of techniques that knowledge engineers use in the acquisition of knowledge from human experts [Mitchell, 1998]. Direct techniques either rely on the expert's ability to articulate his/her knowledge directly, or collect the expert's behavior. In both cases, there is a more or less straightforward way to represent the acquired knowledge using some well-known knowledge representation technique(s). The direct methods include interviews, questionnaires, observation of task performance,

protocol analysis (asking the expert to “think aloud” while performing a task), interruption analysis (asking the expert to work without thinking aloud, and interrupting him/her when the knowledge engineer can no longer understand what the expert is doing), drawing closed curves, and inferential flow analysis (with various weights assigned to the links between concepts and processes). The indirect techniques make various assumptions about how experts represent their knowledge. Results from psychology and cognitive science show that experts recognize old patterns in new problems more effectively than do nonexperts, and that they use a variety of knowledge structures (tables, lists, hierarchies, networks, flow diagrams, and many more). Examples of indirect techniques include various kinds of concept grouping (clustering, sorting, and constructing hierarchies) that the expert is asked to perform explicitly, the use of case-based analysis, and repertory grids (where the expert is asked to evaluate various cases of solving a problem along a number of different “dimensions”, i.e., features). After collecting data using an indirect technique, the knowledge engineer is supposed to work on it with the expert and formulate the underlying knowledge explicitly using appropriate knowledge representation techniques.

Over the years, efforts in the field of knowledge engineering have resulted in the development of various structured methodologies. The best-known example is CommonKADS [Schreiber et al., 1994]. This advocates developing multiple models of the domain knowledge (such as a conceptual model, organizational model, application model, and task model) to cover various aspects of the domain knowledge and help cope with the complexities of building the knowledge base. Furthermore, CommonKADS suggests modeling four categories of knowledge – domain knowledge, task knowledge, inference knowledge, and strategic knowledge – as well as modeling with knowledge reuse in mind. The latter often involves building partial models that are then reused with appropriate refinement in each particular case. CommonKADS insists on strong correspondence between the knowledge models and their implementations – the resulting knowledge bases should clearly and explicitly reflect the models they rely on.

A number of graphical software tools have been developed to support various activities in the process of knowledge engineering, including visual knowledge representation languages. Some of these are covered in the subsequent chapters.

1.7 Open Knowledge Base Connectivity (OKBC)

Developers of intelligent systems use various techniques, languages, and tools to represent, manipulate, and process knowledge in the knowledge bases of their systems. A particular technique, language, or tool may be favored by a knowledge engineer for a number of reasons (e.g., suitability for the particular problem, ease of representation, processing efficiency, or previous experience with the underlying formalism(s)). If the resulting system is developed to operate as a stand-alone one, such preferences may prevail and it may not be necessary to worry about other issues.

However, the resulting system may need to interoperate with other systems and applications, both intelligent and conventional. The other systems and applications will then typically demand services from the intelligent system and access to its knowledge, which implies communication between them. Owing to the potential variety of the underlying knowledge representation formalisms, providing communication interfaces and ensuring reliable interoperability may be difficult. Implementation of full-scale communication and interoperation may be time-consuming and may become quite cumbersome. Even worse, supporting all the idiosyncrasies of the underlying knowledge representation formalisms for each particular application may turn out to mean an enormous multiplication of effort.

OKBC [Chaudhri et al., 1998] is a protocol for accessing knowledge bases stored in *knowledge representation systems* (intelligent systems based on knowledge representation techniques and languages described in Sect. 1.5), and in systems that can be viewed externally as knowledge representation systems (such as object-oriented database systems). It specifies a set of generic operations for access to knowledge bases from various applications. These operations serve as an interface to the knowledge bases. To the external applications demanding services, this interface layer provides some independence from the software and tools that implement and manipulate the knowledge base. Moreover, it enables the development of tools (e.g., graphical browsers, frame editors, analysis tools, and inference tools) that operate on many knowledge representation systems. It also supports knowledge base reuse and composition from multiple individual systems. Version changes of the underlying knowledge representation systems and even transitions to other knowledge representation systems should be transparent to the external applications that access knowledge bases through OKBC.

Just like application programming interfaces for database systems (e.g., Java Data Base Connectivity (JDBC)), OKBC is an application

programming interface for knowledge representation systems. Fig. 1-5 illustrates this idea. Each generic OKBC operation is invoked from the application using a method implemented in the appropriate programming language for the application. In turn, each knowledge representation system implements the protocol in its own way. Hence each knowledge base manipulated by the protocol is serviced directly by the underlying knowledge representation system. OKBC specifies its generic operations in a language-independent way, but also provides appropriate language bindings for Java, Lisp, and C to facilitate the use of generic operations from some of the most popular programming languages. The bindings establish language-specific conventions for naming, passing arguments, returning values, etc.

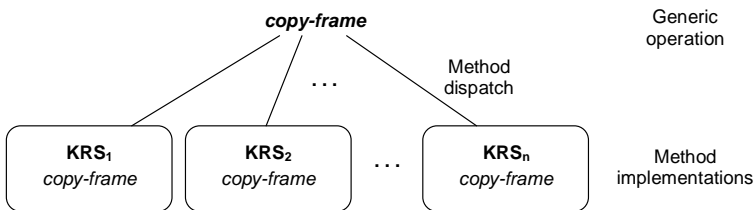


Fig. 1-5. OKBC as an application programming interface

Note that OKBC is a knowledge communication protocol, not a knowledge representation language. In fact, OKBC complements knowledge representation languages in that it specifies generic operations to *access* and *manipulate* knowledge, whereas representation languages *describe* the knowledge declaratively. The OKBC operations are those typically supported by knowledge representation systems, in terms of reasoning and knowledge manipulation (e.g., checking slot values, assessing predicates for truth values, and inheritance).

Since OKBC evolved from an earlier protocol (called the Generic Frame Protocol) that focused only on frame representation systems, many of its operations reflect frame-based representation and reasoning (e.g., *create-class*, *create-individual*, *enumerate-frame-slots*, *get-slot-facets*, and *get-kb-roots*). Other operations serve the pragmatics of using the protocol (e.g., *open-kb*, and *establish-connection*).

The underlying *OKBC Knowledge Model* is object-oriented and is precisely specified in terms of constants, frames, slots, facets, classes, individuals, and knowledge bases. This implies that the knowledge obtained from or provided to a knowledge representation system using OKBC is expressed in a way specified by the OKBC operations and the OKBC Knowledge Model. In other words, the OKBC Knowledge Model

serves as an implicit interlingua that knowledge representation systems and external applications that use OKBC translate knowledge into and out of. Implementing the translation from the knowledge representation system side may raise some minor initial problems of ambiguity and confusion due to terminological inconsistencies – in practice, different systems often use the same term to mean different things, or use different terms to mean the same thing. The translation from the application side is much easier to implement, since the majority of applications in which one may want to use OKBC to communicate with knowledge bases are nowadays object-oriented anyway, just like the OKBC Knowledge Model.

OKBC assumes a client–server architecture for application development and supports standard network protocols such as TCP/IP. A client application may access multiple OKBC servers, which in turn can access multiple knowledge representation systems. Each communication with an OKBC server requires establishing a connection with it first, finding out about the available knowledge representation systems available on the server, and opening a specific knowledge base. All of these operations are directly supported by the appropriate OKBC operations.

1.8 The Knowledge Level

The notion of the *knowledge level* was introduced by Allen Newell [Newell, 1982] to better distinguish between different important views (levels of abstraction) associated with intelligent systems, their organization, the knowledge represented in their knowledge bases, and their operation. It was also used by Newell to discuss the nature of knowledge and the important distinction between knowledge and its representation. As Newell points out, there are three levels/views from the most concrete to the most abstract: the implementation level (originally, Newell called this level the logic level), the logical level (originally, the symbol level), and the knowledge level (epistemological level). The new names for the three levels and the adapted view of the levels presented here have been taken from [Russell & Norvig, 2002] and are more appropriate for the coverage in this book.

- *Implementation level.* This is the level of the knowledge base of an operational intelligent system. At this level, knowledge representation means all of the *data structures* deployed to hold the domain knowledge and other knowledge necessary to solve the current problem, as well as those deployed to hold the current problem description. When it is necessary to make a clear distinction between the former and the latter

(e.g., in expert systems), then the term *working memory* is used to denote the latter. In any case, at the implementation level, both the knowledge and the problem description are represented as data structures. Taking the process view, the implementation level consists of the read/write operations and various internal computations of the system that access and update the knowledge at that level.

- *Logical level.* All data structures in the knowledge base represent something, and hence an interpreter is needed to process them in a way consistent with the representation. In other words, the logical level is the level of knowledge representation. It includes the underlying knowledge representation paradigms, formalisms, techniques, and languages that are used to represent the system's knowledge, implemented using the corresponding data structures at the implementation level, as well as the processes (interpretations) used by the system to reason and make inferences and derivations with that knowledge. At this level, the knowledge is encoded into sentences.
- *Knowledge level.* This is a distinct computer systems level, above the logical level, characterized by knowledge as the medium and the principle of rationality as the law of behavior [Newell, 1982]. It is tightly coupled to the notions of competence and rationality. At the knowledge level, an intelligent system is described in terms of what it knows (the *content*), what its goals are, and what actions it is capable of taking in order to pursue those goals. The system's body of knowledge, described at the knowledge level, is largely unstructured, in terms of both capacity and structural constraints. The system is intelligent if it is capable of rational behavior, i.e., if it knows one that of its actions will lead to one of its goals, it will select and perform that action.

As an illustration of the three different levels/views and the kinds of information associated with them, consider again the example of the king and the princess in Sect. 1.5.1. At the knowledge level, an intelligent system may be said to know that the king cannot see the princess from the garden, because she is in the palace. The system can also be said to use abstract procedures and functions that add new knowledge to the knowledge base and query the knowledge base. For example, an abstract procedure may add the knowledge that the prince is also in the palace; an abstract function enquiring who is in the palace will then return the princess, the prince, and possibly some other people. This knowledge and the abstract update/query processes assume nothing about the actual knowledge representation within the system. At the logical level, predicates such as *in(princess, palace)* and *in(king, garden)* may be the sentences that represent/encode the knowledge contained in the knowledge

body. These predicates are now processed by logical inference procedures to assess their truth values and derive new knowledge (facts), such as $\text{--see}(\text{king}, \text{princess})$. Still, it is up to the implementation to encode the sentences into suitable data structures. The logical sentence $\text{in}(\text{princess}, \text{palace})$ may be encoded as the string “ $\text{in}(\text{princess}, \text{palace})$ ”. Or it may be an entry in a table defined by a relation $\text{in}(\text{Who}, \text{Where})$ and indexed by persons and locations. Or it may be implemented as something else.

As another illustration, consider an OKBC operation such as *copy-frame* (Fig. 1-5). At the knowledge level, an external system may be said to know how to use OKBC to replicate frames. At the logical level, it has to invoke the OKBC *copy-frame* operation and pass it appropriate arguments as required by its specification. However, the external system is not concerned with how the *copy-frame* operation is implemented by the knowledge representation system, nor how the frames are implemented by that system.

Newell originally included computer hardware in his discussion of the implementation (logic) level. True, it is the hardware – the logic circuits and the registers – that performs all the low-level operations upon data structures. It is not necessary, though, for the purpose of this brief overview, to go to that level of detail concerning the implementation of knowledge representation inside a computer. However, it must be stressed that the way the data structures and the related interpreter procedures are designed and implemented is certainly very relevant for the performance of the system (and hardware does matter here). Nevertheless, it is irrelevant to the logical (symbol) level and the knowledge level.

2. Ontologies

The word “*ontology*” comes from the Greek *ontos*, for “being”, and *logos*, for “word”. In philosophy, it refers to the subject of existence, i.e., the study of being as such. More precisely, it is the study of the *categories* of things that exist or may exist in some domain [Sowa, 2000]. A domain ontology explains the types of things in that domain.

Informally, the ontology of a certain domain is about its terminology (domain vocabulary), all essential concepts in the domain, their classification, their taxonomy, their relations (including all important hierarchies and constraints), and domain axioms. More formally, to someone who wants to discuss topics in a domain D using a language L , an ontology provides a catalog of the types of things assumed to exist in D ; the types in the ontology are represented in terms of the concepts, relations, and predicates of L .

Both formally and informally, the ontology is an extremely important part of the knowledge about any domain. Moreover, the ontology is the fundamental part of the knowledge, and all other knowledge should rely on it and refer to it.

This chapter covers ontologies from the perspectives of computing and knowledge representation. It takes a pragmatic approach, explaining the benefits of developing and applying ontologies, the tools and languages used for the development and implementation of ontologies, and the methodologies that bring discipline and rigor to the process of building ontologies. It also shows some examples of ontologies and briefly discusses some applications.

Ontologies have become a very popular topic, not only in AI but also in other disciplines of computing. There are also efforts focused on developing ontologies in many other branches of science and technology. Hence ontologies are growing fast into a distinct scientific field with its own theories, formalisms, and approaches. For a more comprehensive coverage of the field, see [Staab & Studer, 2004].

2.1 Basic Concepts

In AI, the term “ontology” has largely come to mean one of two related things [Chandrasekaran et al., 1999]:

- a representation vocabulary, often specialized to some domain or subject matter;
- a body of knowledge describing some particular domain, using a representation vocabulary.

In both cases, there is always an associated underlying data structure that represents the ontology.

A number of fields of AI and computing use ontologies: knowledge representation, knowledge engineering, qualitative modeling, language engineering, database design, information retrieval and extraction, and knowledge management and organization [McGuinness, 2002].

2.1.1 Definitions

There are many definitions of the concept of ontology in AI and in computing in general. The most widely cited one is

Ontology is a specification of a conceptualization. [Gruber, 1993]

This definition is certainly the most concise one, and requires some further clarification. *Conceptualization* means an abstract, simplified view of the world. If the knowledge base of an intelligent system is to represent the world for some purpose, then it must be committed to some conceptualization, explicitly or implicitly. That is, every body of formally represented knowledge is based on a conceptualization. Every conceptualization is based on the concepts, objects, and other entities that are assumed to exist in an area of interest, and the relationships that exist among them. This also clarifies the meaning of the term “world” – in practice, “world” actually refers to some phenomenon in the world, or to some topic (or topics), or to some subject area.

The other part of the above definition – *specification* – means a formal and declarative representation. In the data structure representing the ontology, the type of concepts used and the constraints on their use are stated declaratively, explicitly, and using a formal language. The formal representation implies that an ontology should be *machine-readable*. However, an ontology is not “active”; it cannot be run as a program. It represents declaratively some knowledge to be used by programs.

Ontology ... can be seen as the study of the organization and the nature of the world independently of the form of our knowledge about it. [Guarino, 1995]

Guarino augments the above definition with the notion of a *formal ontology*, the theory of a priori distinctions between the entities of the world (physical objects, events, regions, quantities of matter, ...), as well as between the meta-level categories used to model the world (concepts, properties, qualities, states, roles, parts, ...). Fundamental roles are played in formal ontology by the theory of part-whole relations and topology (the theory of the connection relation).

Ontology is a set of knowledge terms, including the vocabulary, the semantic interconnections, and some simple rules of inference and logic for some particular topic. [Hendler, 2001]

The important parts in Hendler's definition are the *semantic interconnections* and *inference and logic*. The former says that an ontology specifies the meaning of relations between the concepts used. Also, it may be interpreted as a suggestion that ontologies themselves are interconnected as well; for example, the ontologies of "hand" and "arm" may be built so as to be logically, semantically, and formally interconnected. The latter part means that ontologies enable some forms of reasoning. For example, the ontology of "musician" may include instruments and how to play them, as well as albums and how to record them.

Swartout and Tate offer an informal and metaphorical but extremely useful definition for understanding of the essentials of an ontology:

Ontology is the basic structure or armature around which a knowledge base can be built. [Swartout & Tate, 1999]

Figure 2-1 illustrates this idea. Like an armature in concrete, an ontology should provide a firm and stable "knowledge skeleton" to which all other knowledge should stick. Another important issue here is the distinction between ontological knowledge and all other types of knowledge, illustrated in Table 1-1. An ontology represents the fundamental knowledge about a topic of interest; it is possible for much of the other knowledge about the same topic to grow around the ontology, referring to it, but representing a whole in itself.

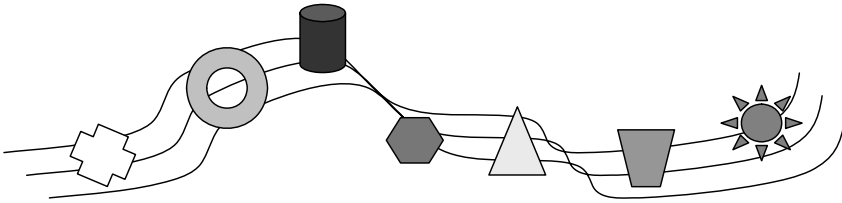


Fig. 2-1. Illustration of Swartout and Tate's definition of an ontology

Kalfoglou stresses yet another important issue related to ontologies:

An ontology is an explicit representation of a shared understanding of the important concepts in some domain of interest. [Kalfoglou, 2001]

The word *shared* here indicates that an ontology captures some *consensual* knowledge. It is not supposed to represent the subjective knowledge of some individual, but the knowledge accepted by a group or a community. All individual knowledge is subjective; an ontology implements an explicit cognitive structure that helps to present objectivity as an agreement about subjectivity. Hence an ontology conveys a shared understanding of a domain that is agreed between a number of individuals or agents. Such an agreement facilitates accurate and effective communication of meaning. This, in turn, opens up the possibility for knowledge sharing and reuse, which enables semantic interoperability between intelligent agents and applications.

2.1.2 What Do Ontologies Look Like?

The answer to the above question depends on the level of abstraction. When implemented in a computer, they typically look like XML-based files. Alternatively, they can be represented in a computer using a logic language, such as KIF [Genesereth & Fikes, 1992]. Since ontologies are always about concepts and their relations, they can be represented graphically using a visual language. Graphical tools for building ontologies always support conversion from a graphical format to XML and other text-based formats (see Sect. 2.2.1).

Humans can express ontologies as sets of declarative statements in a natural language. However, natural-language statements are difficult to process in a computer. Recall also from the definitions that representing ontologies in a computer requires a formal language.

As an example of the representation of ontologies at different levels of abstraction, consider again the concept of a musician. For the sake of

simplicity, assume that the concepts used to describe the essential knowledge about the notion of a musician are *musician*, *instrument*, some products of his/her work, namely *albums* the musician has recorded and music *events* (e.g., concerts) in which he/she has participated, and devoted *admirers* (fans), who keep his/her fame as an artist. Also, assume that the variety and multitude of relations among these concepts that can be considered may be reduced to just a few of the most essential ones, such as the facts that each musician *plays* some instrument, that when giving concerts he/she *plays at* that concert, that the admirers come to *attend* such events, and that the musician also *records* music albums. We deliberately avoid in this simple example the numerous kind-of and part-of relations to other concepts associated with musicians and their work.

These natural-language statements represent the conceptualization of the Musician ontology. At a high level of abstraction, this ontology can be informally diagrammed as the semantic network shown in Fig. 2-2.

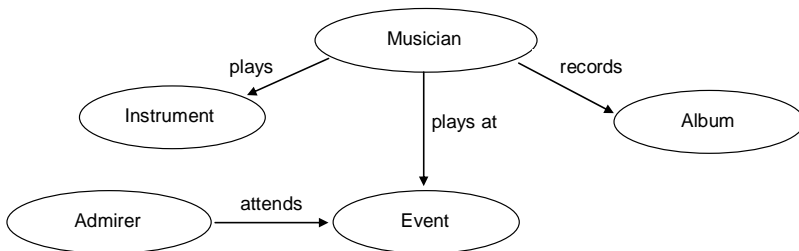


Fig. 2-2. Musician ontology visualized as a semantic network

Obviously, the representation in Fig. 2-2 suffers from many deficiencies. It is not a formal specification, i.e., it is not expressed in any formal language. It does not show any details, such as the properties of the concepts shown or the characteristics of the relations between them. For example, musicians have names, and albums have titles, durations, and years when they were recorded. Likewise, nothing in this semantic network shows explicitly that the musician is the *author* of an album that he/she records (note that recording engineers in music studios can also be said to record albums, but they are usually not the authors). Nevertheless, the semantic network in Fig. 2-2 does show some initial ideas about the Musician ontology.

For more detail and for a formal graphical representation, consider the UML model in Fig. 2-3. This represents the same world as does the semantic network in Fig. 2-2, but allows the properties of all the concepts used to be specified unambiguously, as well as the roles of concepts in

their relations. Another important detail in this representation is an explicit specification of the cardinalities of all concepts.

Figure 2-4 shows part of the Musician ontology in an equivalent XML-based format. The OWL language [Smith et al., 2004] used in this representation is described in more detail in Chap. 3. It is not necessary to go into all the details of this representation, since in practice the representation is always generated automatically, starting from a graphical ontology editor (see Sect. 2.2.1). However, note that it is exactly *this* representation of ontologies that is nowadays most widely used at the implementation level.

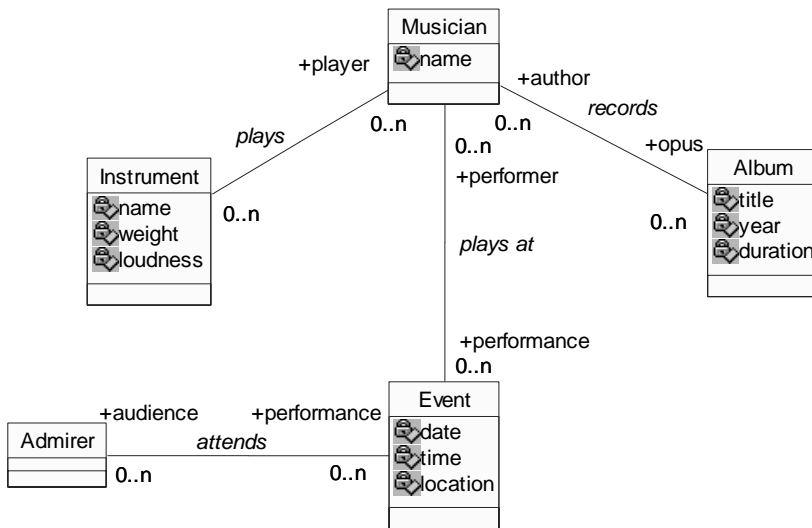


Fig. 2-3. UML model of the Musician ontology

2.1.3 Why Ontologies?

Ontologies provide a number of useful features for intelligent systems, as well as for knowledge representation in general and for the knowledge engineering process. This subsection summarizes the most important of these features, and is based on the treatments in [Chandrasekaran et al., 1999; Gruber, 1993; Guarino, 1995; McGuinness, 2002; Schreiber et al., 1994].


```

<owl:Class rdf:ID="Event"/>
<owl:Class rdf:ID="Album"/>
<owl:Class rdf:ID="Instrument"/>
<owl:Class rdf:ID="Musician"/>
<owl:Class rdf:ID="Admirer"/>
<owl:ObjectProperty rdf:ID="author">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="opus"/>
  </owl:inverseOf>
  <rdfs:domain rdf:resource="#Album"/>
  <rdfs:range rdf:resource="#Musician"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="player">
  <rdfs:range rdf:resource="#Musician"/>
  <rdfs:domain rdf:resource="#Instrument"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="loudness">
  <rdf:type
rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
  <rdfs:domain rdf:resource="#Instrument"/>
</owl:ObjectProperty>
...

```

Fig. 2-4. The Musician ontology represented in OWL (excerpt)

Vocabulary

An ontology provides a *vocabulary* (or the names) for referring to the terms in a subject area.

In real life, there is a whole spectrum of different kinds of vocabularies. A *controlled vocabulary*, such as a catalog, provides a finite list of terms together with an unambiguous interpretation of those terms. Every use of a term from a controlled vocabulary (e.g., “musician”) will denote exactly the same identifier (say 1). A *glossary* provides a list of terms and their meanings, but the meanings are specified in natural language and may often be interpreted differently by different people. Hence they are not unambiguous and thus are not suitable for machine processing. A *thesaurus* provides some additional semantics in the form of synonym relationships between terms, which drastically reduces ambiguity. However, thesauri do not provide explicit term hierarchies [McGuinness, 2002].

Ontologies are different from such human-oriented vocabularies in that they provide *logical statements* that describe what the terms are, how they are related to each other, and how they can or cannot be related to each other. They also specify *rules* for combining the terms and their relations

to define extensions to the vocabulary. As Chandrasekaran et al. [Chandrasekaran et al., 1999] note carefully, it is not the vocabulary as such that qualifies as an ontology, but the conceptualizations that the terms in the vocabulary are intended to capture. An ontology specifies terms with *unambiguous meanings*, with semantics independent of reader and context. Translating the terms in an ontology from one language to another does not change the ontology conceptually. Thus an ontology provides a vocabulary, *and* a machine-processable *common understanding* of the topics that the terms denote. The *meanings* of the terms in an ontology can be communicated between users and applications.

Taxonomy

A *taxonomy* (or *concept hierarchy*) is a hierarchical categorization or classification of entities within a domain. It is also a clustering of entities based on common ontological characteristics. The classification/clustering is organized according to a predetermined system. A good taxonomy should separate its corresponding entities into mutually exclusive, unambiguous groups and subgroups that, taken together, include all possibilities. It should also be simple, easy to remember, and easy to use. A good example of a taxonomy is a Web site's taxonomy; it is the way the site organizes its data into categories and subcategories, sometimes displayed in a site map.

Every ontology provides a taxonomy in a machine-readable and machine-processable form. However, an ontology is more than its corresponding taxonomy – it is a full specification of a domain. The vocabulary and the taxonomy of an ontology together provide a *conceptual framework* for discussion, analysis, or information retrieval in a domain. Note that simple taxonomies such as those of Web sites may not necessarily include complete generalization/specialization hierarchies (i.e., subclassing and is-a relations) for the concepts. With ontologies, the subclassing is strict, is formally specified, includes formal instance relationships, and ensures consistency in deductive uses of the ontology.

Content Theory

Since ontologies identify classes of objects, their relations, and concept hierarchies that exist in some domain, they are quintessentially *content theories* [Chandrasekaran et al., 1999]. Ontologies not only identify those classes, relations, and taxonomies, but also specify them in an elaborate way, using specific *ontology representation languages* (see Section 2.2.1 and Chap. 3). Classes are specified using frame-based representation

principles, i.e., their properties, property values, and possible value restrictions (restrictions on what can fill a property) are specified as well. In some ontology representation languages, the value of one property may be expressed as a mathematical equation using values of other properties. Also, some languages allow developers to specify first-order-logic constraints between terms and more detailed relationships such as disjoint classes, disjoint coverings, inverse relationships, and part-whole relationships, etc. [McGuinness, 2002]. Thus ontologies represent knowledge in a very structured way.

Well-structured and well-developed ontologies enable various kinds of *consistency checking* from applications (e.g., type and value checking for ontologies that include class properties and restrictions). They also enable and/or enhance *interoperability* between different applications. For example, we may want to expand the Musician ontology illustrated in Figs. 2-2–2-4 to include the concept of a *street musician*, to denote a musician who entertains people in the street. One way to do this is to define a *performsIn* property in the Musician class and include in the ontology a definition stating that a street musician is a Musician whose *performsIn* property has the value “Street”. This definition may be used to expand the term “StreetMusician” in an application that does not understand that term, but does understand the terms “Musician”, “performsIn”, and “Street”. If that application is asked by another application if a certain person is a street musician, it will “understand” the question and may be able to answer by querying a database of musicians to see if it contains an entry with the appropriate values of the *name* and *performsIn* fields.

Being content theories, ontologies clarify the structure of domain knowledge. Developing an ontology requires an effective *ontological analysis* of the domain whose content the ontology is intended to represent. Ontological analysis reveals the concepts of the domain knowledge, their taxonomies, and the underlying organization. Without such analysis, no knowledge representation for the domain can be well founded. Through ontological analysis, the entire process of knowledge engineering acquires a strong flavor of modeling. The resulting knowledge base does not merely transfer the knowledge extracted from a human expert, but also models the problem domain in the form of the observed behavior of an intelligent agent embedded in its environment [Gaines, 1991; Gruber, 1993; Guarino, 1995; Schreiber et al., 1994].

Knowledge Sharing and Reuse

The major purpose of ontologies is not to serve as vocabularies and taxonomies; it is *knowledge sharing* and *knowledge reuse* by applications.

The point is that every ontology provides a description of the concepts and relationships that can exist in a domain and that can be shared and reused among intelligent agents and applications (recall that the description looks like a formal specification of a program; see Figs. 2-3 and 2-4). Moreover, working agents and applications should be able to communicate such ontological knowledge. Shared ontologies let us build specific knowledge bases that describe specific situations but clearly rely on the same underlying knowledge structure and organization.

As Neches et al. note, there are many senses in which the work that went into creating a knowledge-based system can be shared and reused [Neches et al., 1991]; for example,

- through the inclusion of source specifications – the content of one module is copied into another one at design time, is then possibly extended and revised, and is finally compiled into a new component;
- through the runtime invocation of external modules or services – one module invokes another, either as a method from a class library or through a Web service, and the like;
- through communication between agents – the messages that intelligent agents send to and receive from each other can have various kinds of knowledge as their content;
- through the exchange of techniques – sharing and reusing not the content, but the approach behind it (in a manner that facilitates reimplementing of the content itself).

These modes of sharing and reuse require shared understanding of the intended interpretations of domain terms, compatibility of the domain models used by different agents and applications, and compliance with the kinds of requests that the external modules/services are prepared to accept. All of it is facilitated by shared ontologies. Shared ontologies ameliorate the problems of heterogeneous representations in the knowledge bases of different systems (even those developed using the same representational paradigm), dialects within language families, varieties in communication conventions, and model mismatches at the knowledge level (often caused by a lack of a shared vocabulary and domain terminology).

As an example of facilitating knowledge sharing and reuse by means of ontologies, suppose that someone has conducted a thorough ontological analysis of the topic of musicians, and has developed a much more elaborated Musician ontology than that shown in Figs. 2-2–2-4. This ontology would include domain-specific terms such as *musician* and *musical event*, some general terms such as *profession*, *location*, and *attendance*, and terms that describe behavior, such as *playing* and

recording. An ontology captures the intrinsic conceptual structure of the domain [Chandrasekaran et al., 1999], and can be used as a basis for developing a rich domain-specific knowledge representation language for building knowledge bases in that domain. In the present case, the language would provide a syntax for encoding knowledge about musicians in terms of the vocabulary, concepts, and relations in the ontology. Anyone who wanted to build a knowledge base related to musicians could use that content-rich knowledge representation language and thus eliminate the need to perform the time-consuming knowledge analysis task again – the language would already have a large number of terms that embodied the complex content theory of the domain. In this way, the Musician ontology could be shared among different developers, and reused as the “armature” knowledge in a number of knowledge bases and applications.

Make no mistake, though – in practice, knowledge sharing and reuse is still not easy, even if an ontology is readily available for a given purpose. To name but a few reasons, note that there are several different languages for representing ontologies, and knowledge base development tools may not support the language used to develop the ontology. There are also competing approaches and working groups, creating different technologies, traditions, and cultures. There may be several different ontologies that have been developed to describe the same topic or domain. Selecting any one of them may not satisfy all the requirements that the knowledge engineer must fulfill. Combining them may be anything but easy, because subtle differences between them may require a lot of manual adjustment, and the resulting ontology may still be inadequate. On top of all that, there is the problem of knowledge maintenance, since all parts of knowledge (including ontological knowledge) evolve over time.

2.1.4 Key Application Areas

There are many potential applications of ontologies, but Fikes [Fikes, 1998] has offered a high-level list of key application areas: collaboration, interoperation, education, and modeling.

Collaboration. Different people may have different views of the same problem area when working on a team project. This is particularly true for interdisciplinary teams, with specialists from different branches of science, technology, and development having different foci of interests and expertise. For such specialists, ontologies provide a unifying knowledge skeleton that can be used as a common, shared reference for further development and participation – these people can simply talk more easily

to each other when they have such a stable, consensual knowledge armature to rely on.

Perhaps even more importantly, ontologies play the same role in collaboration between intelligent agents in terms of agent-to-agent communication. When an agent sends a message to another agent that it is communicating with, the other agent must have the same world model (i.e., the same ontology) in order to interpret the message correctly. Knowledge exchange between agents is much more feasible when the agents are aware of the ontologies that the other agents are using as world models.

Interoperation. Ontologies enable the integration of information from different and disparate sources. End users typically do not show much interest in how they get their information; they are much more interested in getting the information they need, and getting all of it. Distributed applications may need to access several different knowledge sources in order to obtain all the information available, and those different sources may supply information in different formats and in different levels of detail. However, if all the sources recognize the same ontology, data conversion and information integration are easier to do automatically and in a more natural way.

Education. Ontologies are also a good publication medium and source of reference. Since they presumably always result from a wide consensus about the underlying structure of the domain they represent, they can provide reliable and objective information to those who want to learn more about the domain. Simultaneously, domain experts can use ontologies to share their understanding of the conceptualization and structure of the domain.

Modeling. In modeling intelligent, knowledge-based applications, ontologies represent important reusable building blocks, which many specific applications should include as predeveloped knowledge modules. For example, the Musician ontology defines knowledge that can be used as is in both a recommender system that suggests to users what new music CDs they should buy, and a Web-based intelligent educational system that learners may use to find out more about famous instrumentalists of the twentieth century.

Fikes' classification can be seen from a more pragmatic perspective as well. In fact, many consider e-commerce to be *the* application domain for ontologies. Ontologies can enable machine-based communication between buyers and sellers, can help in customer-profiling tasks, can support vertical integration of markets, and can describe reuse between different marketplaces. In e-commerce, ontologies can be applied in terms of all of the four categories (roles) that Fikes suggested.

Another extremely demanding general application area provides very fertile soil for applying ontologies – search engines. Ontologies can support structured, comparative, and customized searches [McGuinness, 2002]. Concepts and taxonomies from ontologies can be used to find pages with syntactically different but semantically similar content, simultaneously eliminating many irrelevant hits. For example, if a user searches the Web for special offers on music CDs of individual performers and gets too many hits, an intelligent search engine can consult an ontology such as the Musician ontology to obtain relevant properties that can help to refine the search. On the basis of the properties retrieved, the user may be presented with a special-purpose form to fill in, so that he/she can provide a detailed set of specifications for the CDs of interest (such as the preferred instrument, and live or studio recordings). The ontology may also offer refinements using its generalization/specialization relations, so, in the case of individual performers, the user may also be automatically asked to select between instrumentalists and singers, and, further on, to select between guitarists, pianists, and so on. While waiting for user input, the search engine could proactively perform a less refined search in the background and possibly cache information for further (more refined) searches.

2.1.5 Examples

The following brief descriptions of some existing ontologies are intended to show the variety of ontologies and their potential uses. Most of the examples have been selected by starting from ontology libraries available on the Web [DAML Ontology Library, 2005; OWL Ontology Library, 2005; Protégé Ontologies Library, 2005].

Example 1: The Gene Ontology project ***(<http://www.geneontology.org/>)***

This project provides a controlled vocabulary to describe gene and gene product attributes for any organism. The ontology has three organizing principles: molecular function, biological process, and cellular component. A gene product has one or more molecular functions. Also, a gene product is used in one or more biological processes, and might be associated with one or more cellular components. This ontology is frequently updated and is available for download in various formats.

Example 2: The Learner Ontology

The goal of the Learner Ontology Project (<http://www.l3s.de/~dolog/learnerrdfbindings/>) is to research user modeling in open P2P environments, where learner profiles are distributed and, also, fragments of learner profiles are distributed. The ontology covers various features of learners, several types of security access, various levels of the learner profile, and so on.

Example 3: The Object-Oriented Software Design Ontology (ODOL)

The objectives of the Web of Patterns project (<http://www-ist.massey.ac.nz/wop/>) are to create an ontology to describe the design of object-oriented software, to represent software design patterns [Gamma et al., 1995] and related concepts using the concepts developed in that ontology, and to provide a flexible framework that can be used by the software engineering community to share knowledge about software design. The ODOL-based descriptions of patterns are OWL documents that can be imported by popular ontology development editors such as Protégé (see Sect. 2.2.1) for a more detailed description of Protégé).

2.2 Ontological Engineering

To develop a really useful ontology requires a lot of engineering effort, discipline, and rigor. *Ontological engineering* denotes a set of design principles, development processes and activities, supporting technologies, and systematic methodologies that facilitate ontology development and use throughout its life cycle – design, implementation, evaluation, validation, maintenance, deployment, mapping, integration, sharing, and reuse. Ontological engineering provides a design rationale for the development of knowledge bases and enables systematization of knowledge about the world of interest and accumulation of knowledge [Mizoguchi & Kitamura, 2001].

Knowledge engineering for an intelligent system should always include ontological engineering, which implies using specific development tools and methodologies.

2.2.1 Ontology Development Tools

The standard tool set of an ontology engineer includes ontology representation languages and graphical ontology development environments.

More recently, ontology-learning tools have also started to appear, in order to partially automate the development process and help in the evolution, updating, and maintenance of ontologies. Other tools are also required in the context of developing ontologies for deployment on the Semantic Web (see Chap. 3).

Ontology representation languages

There are a number of ontology representation languages around. Some of them were developed at the beginning of the 1990s within the AI community. Others appeared in the late 1990s and later, resulting from the efforts of AI specialists and the World Wide Web Consortium (W3C). Roughly speaking, the early ontology representation languages belong to the pre-XML era, whereas the later ones are XML-based. Also, most of the later ones were developed to support ontology representation on the Semantic Web, and hence they are also called “*Semantic Web languages*”. Other common names for them are “*Web-based ontology languages*” and “*ontology markup languages*” [Gómez-Pérez & Corcho, 2002].

Some of the best-known examples of the early ontology representation languages are:

- KIF [Genesereth & Fikes, 1992], which is based on first-order logic;
- Ontolingua [Gruber, 1992], which is built on top of KIF but includes frame-based representation;
- Loom [MacGregor, 1991], based on description logics.

Among the widely used Web-based ontology languages are:

- SHOE [Luke & Heflin, 2000], built as an extension of HTML;
- XOL [Karp et al., 1999], developed by the AI center of SRI International as an XML-ization of a small subset of primitives from the OKBC protocol called OKBC-Lite;
- RDF [Manola & Miller, 2004], developed by the W3C as a semantic-network-based language to describe Web resources;
- RDF Schema [Brickley & Guha, 2004], also developed by the W3C, is an extension of RDF with frame-based primitives; the combination of both RDF and RDF Schema is known as RDF(S);
- OIL [Fensel et al., 2001], which is based on description logics and includes frame-based representation primitives;
- DAML+OIL [Horrocks & van Harmelen, 2002] is the latest release of the earlier DAML (DARPA Agent Markup Language), created as the result of a joint effort of DAML and OIL developers to combine the expressiveness of the two languages;

- OWL, or Web Ontology Language [Smith et al., 2004], developed under the auspices of the W3C and evolved from DAML+OIL; OWL is currently the most popular ontology representation language.

Some of these Web-based ontology languages are described in detail in Chap. 3. For more comprehensive information and comparative studies of all of them, see the sources cited above and [Corcho et al., 2002; Gómez-Pérez & Corcho, 2002].

The ontological-engineering reality of having a number of ontology representation languages is best expressed in the following quotation:

Ideally, we would like a universal shared knowledge-representation language to support the Semantic Web, but for a variety of pragmatic and technological reasons, this is unachievable in practice. Instead, we will have to live with a multitude of metadata representations. [Decker et al., 2000]

There are a number of implications of that multitude. For example, an application may already use another ontology, developed in a language other than that which the developers have chosen for building a new ontology. Experience shows that combining ontologies developed using different languages can require a lot of effort – two languages coming from different branches of the ontology-language research communities may not be compatible, and may require painful manual adaptation in order to provide interoperability at a satisfactory level. However, sometimes the developers may be constrained by the fact that the development tools support only a few languages. The situation is even worse if there are other applications that already use an ontology that the new application needs to consult as well, but design considerations show that another language would be much more appropriate for the new applications. On the other hand, an appropriate ontology representation language may facilitate the integration of an ontology into a new application. Moreover, a newer ontology representation language may simply be more expressive than the other languages of choice, and translators to and from those other languages may already exist as well. Note that the ontology community has already developed a number of such translators, although many of them suffer from partial loss of knowledge in the translation process.

Ontology Development Environments

No matter what ontology representation language is used, there is usually a graphical *ontology editor* to help the developer organize the overall

conceptual structure of the ontology; add concepts, properties, relations, and constraints; and, possibly, reconcile syntactic, logical, and semantic inconsistencies among the elements of the ontology. In addition to ontology editors, there are also other tools that help to manage different versions of ontologies, convert them into other formats and languages, map and link between ontologies from heterogeneous sources, compare them, reconcile and validate them, and merge them. Yet other tools can help acquire, organize, and visualize the domain knowledge before and during the building of a formal ontology [Denny, 2002].

Graphical *ontology development environments* integrate an ontology editor with other tools and usually support multiple ontology representation languages. They are aimed at providing support for the entire ontology development process and for the subsequent use of the ontology [Corcho et al., 2002].

Protégé. Currently the leading ontology development editor and environment, Protégé was developed at Stanford University and has already been through a number of versions and modifications [Protégé, 2005]. It facilitates the defining of concepts (classes) in an ontology, properties, taxonomies, and various restrictions, as well as class instances (the actual data in the knowledge base). Furthermore, its uniform GUI (Fig. 2-5) has a tab for the creation of a knowledge acquisition tool for collecting knowledge into a knowledge base conforming to the ontology. Customizable forms determine how instance information is presented and entered. The knowledge base can then be used with a problem-solving method to perform various inference tasks.

Protégé conforms to the OKBC protocol for accessing knowledge bases stored in knowledge representation systems. Applications built on top of such systems can be executed within the integrated Protégé environment.

Protégé supports several ontology representation languages, including OWL and RDF(S). Some forms of reasoning over ontologies developed with Protégé are also facilitated; for example, since OWL is based on description logics, inferences such as satisfiability and subsumption tests are automatically enabled.

Protégé's plug-in-based extensible architecture allows integration with a number of other tools, applications, knowledge bases, and storage formats. For example, there are plug-ins for the ezOWL (Visual OWL) editor, for the OWL-S editor (enabling loading, creating, management, and visualization of OWL-S services), and for Jess (which allows the use of Jess and Protégé knowledge bases together), as well as storage back ends for UML (for storing Protégé knowledge bases in UML), XML Metadata Interchange (XMI) (for storing Protégé knowledge bases as XMI files),

DAML+OIL (for creating and editing DAML+OIL ontologies), and many more formats.

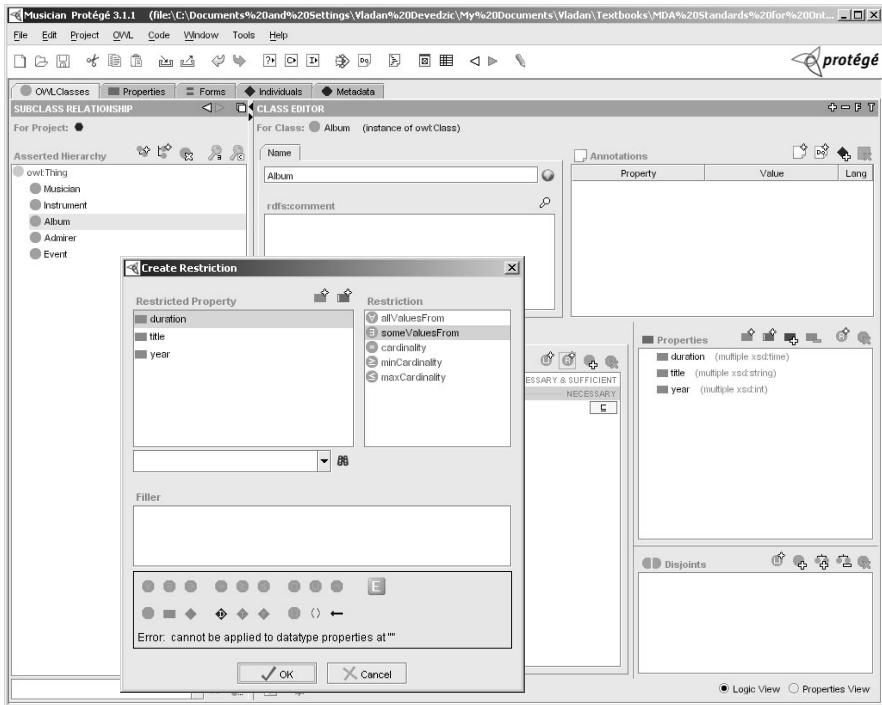


Fig. 2-5. A screen from Protégé

Initially, Protégé was used to develop ontologies in application domains such as clinical medicine and the biomedical sciences. Over the years, its user community has grown rapidly and it is now used in a number of other application domains. More information can be found in [Noy et al., 2001].

Other environments. Although Protégé is currently the most widely used ontology development environment, there are literally dozens of other tools and environments. A relatively recent comparative survey [Denny, 2004] discovered the fact that there is a lot of room for improvement in all such environments. For example, a number of users suggested an enhancement in the form of a higher-level abstraction of ontology language constructs to allow more intuitive and more powerful knowledge-modeling expressions. Many users also would like friendlier visual/spatial navigation among concept trees/graphs and linking relations, more options for using reasoning facilities to help explore, compose, and check ontologies, more features for aligning ontologies with one another, and

tools that would help integrate ontologies with other data resources such as enterprise databases. The desirable improvements also include support for natural-language processing and collaborative development.

The overall sentiment expressed by users of the various ontology development environments clearly reflected the need for facilitating the use of such tools by domain experts rather than by ontologists. Likewise, there is a strong need for the integration of ontology development environments with existing domain and core ontologies and libraries, as well as with standard vocabularies. Another, more contemporary focus is emerging as well – ontology development in concert with trends in enterprise application integration and development trends.

Ontology-Learning Tools

Ontology development is hard work. Even with the most advanced ontology development languages, environments, and methodologies, a major problem in ontological engineering still remains in the area of knowledge acquisition and maintenance – the collection of concepts and relations in a domain, achieving consensus on them among the domain experts and other interested parties, and frequent updates due to the dynamics of the knowledge structure of the domain and its unpredictable changes over time.

This fact has created the idea of *ontology learning*, with the objective of partially automating the processes of ontology development and maintenance by developing tools and frameworks to help extract, annotate, and integrate new information with old information in an ontology. The typical sources of information are Web documents, which reflect sufficiently well the dynamics of change in most domains. A typical prerequisite for enabling (semi)automated information extraction from Web documents is the use of natural-language-processing and text-processing technologies.

To this end, Maedche and Staab [Maedche & Staab, 2001] have proposed an ontology-learning framework and developed an ontology-learning environment called the Text-To-Onto workbench. The framework and tool are based on an architecture that combines knowledge acquisition with machine learning from Web documents. The framework recognizes the fact that traditional machine learning techniques rely on data from structured knowledge bases or databases, and hence are not applicable to documents and other sources of information on the Web, which are at best partially structured or semistructured. Instead, the framework relies on natural-language-processing, data-mining, and text-mining technologies. The specific techniques that it applies are ontology learning from free text,

dictionaries, and legacy ontologies; reverse engineering of ontologies from database schemata; and learning from XML documents.

The framework suggests several steps in the ontology-learning process:

- importing and reusing existing ontologies by merging their structures or defining mapping rules between those structures;
- modeling major parts of the target ontology using ontology extraction from Web documents with support from machine learning;
- outlining the target ontology to fit it to its primary purpose;
- refining the ontology at a fine granularity (in contrast to extraction);
- validating the resulting ontology by use of the target application;
- repeating the above steps to include new domains in the constructed ontology, to fine-tune it, or to maintain and update its scope.

Note that much of this process is semiautomated, i.e., the ontology engineer is supposed to interactively select and run a number of techniques from a graphical environment. He/she first has to select the potential sources of ontological information from the Web, such as HTML and XML documents, databases, or existing ontologies, and then drive the discovery process to exploit them further. This may include indexing various documents, so as to transform semistructured documents such as online dictionaries into a predefined relational structure. The process also assumes access to a natural-language-processing system that enables lexical analysis, tokenization, named-entity recognition, and other operations over a free-form natural-language text. The Text-To-Onto ontology-learning workbench integrates an ontology editor and all of the above other tools and techniques. A modified, generalized association-rule learning algorithm known from the field of data mining is used to discover relations between concepts in a given class hierarchy. The relations discovered are collected in a database, which can be also visualized to help analyze the relations more effectively.

A similar idea has been exploited by Navigli et al. in their OntoLearn architecture, which uses the Ariosto language processor [Navigli et al., 2003]. OntoLearn extracts terminology from appropriate Web sites, warehouses, and documents exchanged among members of a virtual community, filters it using natural-language processing and statistical techniques, and interprets it semantically using the WordNet lexical knowledge base [WordNet, 2005]. The extracted terminology and concepts are then related semantically according to taxonomic (kind-of) and other semantic relations, obtained from WordNet and a rule-based inductive-learning method. The initial ontology is then edited, validated, enriched, and updated interactively, again with the aid of WordNet, which ensures

correct use of various synonyms, hypernyms, homonyms, and the like for semantic disambiguation between terms in the ontology.

All approaches to ontology learning such as the above two are promising, but are still very much in the research phase and have not been integrated into common ontology development environments such as Protégé.

2.2.2 Ontology Development Methodologies

An ontology development methodology comprises a set of established principles, processes, practices, methods, and activities used to design, construct, evaluate, and deploy ontologies. Several such methodologies have been reported in the literature. From surveys such as those in [Corcho et al., 2002] and [Staab & Studer, 2004], we can conclude that:

- most ontology development methodologies that have been proposed focus on *building* ontologies;
- some other methodologies also include methods for *merging*, *reengineering*, *maintaining*, and *evolving* ontologies;
- yet other methodologies build on general software development processes and practices and apply them to ontology development.

There is no one best methodology, because there is no one “correct” way to model a domain. Also, ontology development is necessarily an iterative process.

Of the methodologies in the first of the above three categories, some are fairly general and merely suggest steps to be followed in the ontology development process. An example is the simple methodology proposed by Noy and McGuinness [Noy & McGuinness, 2001]. Others advise specific ontology development processes (such as the one proposed by van der Vet and Mars [van der Vet & Mars, 1998] for bottom-up construction of ontologies). As an illustration, consider the steps in the ontology-building process suggested in [Noy & McGuinness, 2001]:

- Determine the domain and scope of the ontology – this should help create a clear vision of the ontology’s coverage, its intended use, the types of questions the information in the ontology should provide answers to, and maintenance guidelines.
- Consider reusing existing ontologies – since ontology development is hard work, it is always a good idea to check if someone else has already done the work and made the result publicly available, so that it can be refined and extended to suit a particular domain or task. Language

conversion, interoperability, and tool support issues are important here as well.

- Enumerate important terms in the ontology – this is where building the terminology starts.
- Define the classes and the class hierarchy – this step, closely intertwined with the next one, can be performed top-down (identifying the most general concepts and classes first), bottom-up (identifying the most specific ones first), middle-out (starting from some important middle-layer concepts and expanding the hierarchy in both directions), or using a combination of these approaches.
- Define the properties (slots) of classes – describe the internal structure of concepts by explicating their extrinsic properties (e.g., *name*, *duration*, and *use*), intrinsic properties (e.g., *weight*), parts, and relations to other classes and individuals in those classes.
- Define the facets of the slots – these are things such as the slot value type, the allowed values (domain and range), the number of values (cardinality), and other features of the values that the slot can take.
- Create instances – this includes filling in the slot values for each instance created.

Again, the process outlined above is much more complicated in practice. It requires one to consider a lot of conflicting issues and a number of fine-grained details. After multiple, usually time-consuming iterations, at least a minimum consensus about the final ontology should be achieved.

An example of a more comprehensive methodology is the *Methontology framework* [Fernández-López et al., 1999]. Methontology's starting point is that ontological engineering requires the definition and standardization of the entire ontology life cycle – from the specification of requirements to maintenance – as well as methodologies and techniques that drive ontology development through the life cycle. So, the Methontology framework includes:

- identification of the ontology development process;
- a life cycle based on evolving prototypes;
- the methodology itself, which specifies the steps for performing each activity, the techniques used, the products of each activity, and an ontology evaluation procedure.

The ontology development process in Methontology comprises the following phases:

- *specification* – identification of the ontology's terminology, primary objective, purpose, granularity level, and scope;

- *conceptualization* – organizing and structuring in a semiformal way the knowledge acquired during the specification phase, using a set of intermediate representations that both domain experts and ontologists can understand (thus bridging the gap between their mindsets);
- *implementation* – using an ontology development environment to formally represent and implement the products of the above two phases, namely concepts, hierarchies, relations, and models.

In addition to the above major stages of ontology development, Methontology covers processes that run in parallel throughout the ontology life cycle, i.e., along with the processes taking place in the above three major stages: quality assurance, integration, evaluation, maintenance, documentation, and configuration management. It also identifies interdependencies between the life cycle of the ontology being developed and the life cycles of other, related ontologies. Furthermore, Methontology specifies in detail the techniques used in each activity, the products that each activity outputs, and how they have to be evaluated.

Methontology also recognizes the importance of knowledge acquisition. In this framework, knowledge acquisition is the long process of working with domain experts, and its activities are intertwined with activities from the specification and conceptualization phases. It comprises the use of various knowledge acquisition techniques (see Sect. 1.6), to create a preliminary version of the ontology specification document, as well as all of the intermediate representations resulting from the conceptualization phase.

Note that Methontology is suitable for building ontologies either from scratch or by reusing other ontologies, as they are, or by a process of reengineering them.

An example of the third category of ontology development methodologies and processes – those that rely on general software engineering principles – can be found in [Devedžić, 2002]. It proposes an ontology development methodology analogous to that of object-oriented software analysis and design. The rationale is as follows. Ontologies represent concepts, their properties, the values of those properties, events and their causes and effects, processes, and time [Chandrasekaran et al., 1999]. Also, ontologies always include some kind of hierarchy, and most ontologies represent and support generalization, inheritance, aggregation (part-of), and instantiation relationships among their concepts. Nearly all of these issues are relevant to an object-oriented analysis and design for a problem domain. Moreover, the processes that ontological engineers use in ontology development (see above) almost coincide with established processes of object-oriented analysis and design (see, e.g., [Larman,

2001)). In both cases, it is important to assemble the domain vocabulary at the beginning, often starting from the domain's generic nouns, verbs, and adjectives. The result of an object-oriented analysis is actually a draft of the domain ontology relevant to the application. True, software analysts do not call this result an ontology. Object-oriented analysis stresses different aspects from those that ontological analysis does, but the parallels are obvious. In the next phase, object-oriented designers define classes, objects, hierarchies, interface functions, and the system's behavior, while ontological engineers use various intermediate representations to design ontologies in detail. Both kinds of specialists commonly use various templates for specifying details of their products. Some classes become merged and some refined, and so do some ontologies. Class libraries and previous design specifications often provide the possibility of reuse in object-oriented design, and so do previously encoded and publicly available ontologies in ontological engineering.

A similar methodology described in [Devedžić, 1999] is built on the idea of using *design patterns* as simple and elegant solutions to specific problems in object-oriented software design [Gamma et al., 1995]. Design patterns provide a common vocabulary for designers to communicate, document, and explore software design alternatives. They contain the knowledge and experience that underlie many redesign and recoding efforts of developers who have struggled to achieve greater reuse and flexibility in their software. Although design patterns and ontologies are not the same, they overlap to some extent. First, both of them are about vocabularies, about knowledge, and about “architectural armatures”. Both concepts also describe things at the knowledge level. Ontologies are more commonsense-oriented; design patterns are more concrete. Next, it is possible to draw an analogy between libraries of ontologies and catalogs of design patterns. Although catalogs of design patterns do not provide ready-to-use building blocks like ontologies from libraries, some attempts are being made to make ready-to-use blocks available from them (see, e.g., [Staab & Studer, 2004]). Also, it does not take a hard mental shift to view ontologies as abstract patterns or as knowledge skeletons of domains. Likewise, it is not too hard to understand design pattern templates as knowledge of what the ontologies of design patterns may look like. All of these similarities support the possibility of developing ontologies in much the same way as software systems are developed using design patterns.

Two important observations come out of this brief survey of ontology development methodologies. First, there are many common points in the various methodologies. Steps in different processes may be named differently, may also be of different granularity, or may only partially overlap, but the processes are still very much alike. Second, many of the

principles and practices of ontology development are analogous to those of software engineering. The major topic of this book – applying the principles and standards of the MDA modeling and development approach, taken from software engineering, to ontology development – is completely in line with these observations.

2.3 Applications

Ontologies have become a major conceptual backbone for a broad spectrum of applications [Staab & Studer, 2004]. There is an increasing awareness among researchers and developers that ontologies are not just for knowledge-based systems, but for all software systems – all software needs models of the world, and hence can make use of ontologies at design time [Chandrasekaran et al., 1999]. The major application fields for ontologies nowadays include knowledge management, e-learning, e-commerce, and integration of Web resources, intranet documents, and databases. They also include cooperation of Web services with enterprise applications, natural-language processing, intelligent information retrieval (especially from the Internet), virtual organizations, and simulation and modeling. The following examples from a broad spectrum of ontology application scenarios are but a few typical illustrations.

2.3.1 Magpie

Magpie is a tool that supports semantic interpretation of Web pages, thus enabling intelligent Web browsing [Domingue et al., 2004]. It automatically associates an ontology-based semantic layer with Web resources, which enables the invocation of relevant services within a standard Web browser. In other words, ontologies make it possible to associate meaning to information on a Web page and then, on the basis of the identified meaning, to offer the user appropriate functionalities. In fact, Magpie offers complementary knowledge sources relevant to a Web resource, thus facilitating quick access to the underlying background knowledge and helping the user to make sense of content and contextual information on Web pages the user may be unfamiliar with.

Magpie works as a plug-in to standard Web browsers and appears as an additional toolbar in the browser. It relies on the availability of ontologies that represent various domains of discourse. The user can select an ontology for Magpie to work with, and the buttons that will appear in the Magpie toolbar will correspond to the concepts in the ontology. He/she can

then use the Magpie toolbar to toggle highlighting of specific concepts of interest to him/her for the browsing session. The underlying ontology selected must be populated by instances, possibly automatically mined from relevant Web pages. The browser, when showing a Web page, will then highlight information related to the types of entities in the ontology that the user has selected from the Magpie toolbar. For example, if the ontology selected is the Musician ontology, and the highlighted concepts in the Magpie toolbar are Instrument and Album, in the home page of, say, a rock and roll star shown in the browser, words such as “guitar” and “keyboards” will be highlighted, and so will the titles of the star’s albums.

Magpie detects patterns in the browsing session by tracking interesting items in the browsing log with the aid of an ontology-based filter. When a pattern is detected, Magpie activates an appropriate context-dependent trigger service. In the case of the Musician ontology and Web pages related to musicians and their activities, Magpie may use one panel to show the names of the musicians that it has semantically recognized from the Web pages in the browsing log, and another panel to show the titles of all of the albums related to those names in one way or another. In yet another panel, it may show musicians and albums not explicitly mentioned on the pages accessed in that session, but obtained from the populated ontology. This reveals explicitly to the user the information semantically related to the context of the browsing session. By right-clicking on any of the highlighted concepts on the page shown, the user can access from a popup menu any of the context-dependent (i.e., ontology-dependent) semantic services. In the case of musicians, these might be related to their concerts and tours, colleagues, managers, and so forth.

2.3.2 Briefing Associate

Knowledge sharing and reuse through automatic exchange of Web documents among applications and agents is possible only if the documents contain ontologically encoded information, often called semantic markup or semantic annotation, that software agents and tools can interpret accurately and reliably. Current annotation technology is covered in more detail in Chap. 3, but it suffices for this overview of Briefing Associate to note that annotation is usually performed manually (using annotation tools), which is a tedious and error-prone process. Briefing Associate deploys ontological knowledge to encode document annotation automatically as authors produce documents [Tallis et al., 2002].

The approach used in Briefing Associate can be described in simple terms as extending a commercial, frequently used document editor with ontology-based additional tools that the targeted category of authors will be highly motivated to use. Whenever such an author applies any of the additional tools, an appropriate annotation is automatically created and inserted into the document. Thus annotation comes at virtually no extra cost, as a by-product of activities that the author would perform anyway. The prerequisites include the existence of domain ontologies that authors creating documents can rely on, and easy creation of specific widgets to represent the ontology-based additional editing tools.

To this end, Briefing Associate has been implemented as an extension of Microsoft's PowerPoint in much the same way as Magpie extends standard Web browsers (see above) – it appears in PowerPoint's native GUI as a toolbar for adding graphics that represent a particular ontology's classes and properties. The graphical symbols on the toolbar are obtained from a special-purpose tool that lets graphic designers create such symbols to visually annotate ontologies. In a hypothetical example, the Musician ontology might be visualized with that tool and various graphical symbols might be created to represent musical events, instruments, and the other concepts in the ontology. These symbols would then be inserted into the toolbar to represent the domain ontology (Musician) in PowerPoint.

To the domain author, the native PowerPoint GUI is still there, the editing process continues normally, and the resulting slide show looks as if the ontology was not used in the editing of the presentation. However, using any graphical symbol from the additional toolbar in the presentation document results in inserting a transparent annotation into the slides, which is saved with the document. PowerPoint installations not extended with Briefing Associate ignore such transparent annotations. However, the point is that the annotation can be used by Briefing Associate internally to produce various metadata (such as the document title, the author, and a reference to the ontology used), and additional XML-based documents that can be published on the Web so that other agents and applications can locate the main document more easily and automatically interpret its content in terms of the concepts in the ontology.

2.3.3 Quickstep and Foxtrot

Quickstep and Foxtrot are ontology-based recommender systems that recommend online academic research papers [Middleton et al., 2004]. Although they both focus on a relatively small target group of Web users,

the principles built into these two systems can be translated to other target groups as well.

In general, recommender systems unobtrusively watch user behavior and recommend new items that correlate with a user's profile. A typical example of such recommendation can be found on Amazon.com, where users obtain suggestions about what books, CDs, and other items to buy according to their observed shopping behavior and a previously created set of user profiles. Recommender systems usually create user profiles on the basis of user ratings of specific items and the content of items; however, in many cases, this may be insufficient and can lead to inconsistent recommendations.

Quickstep and Foxtrot rely on ontology-based user profiling. They use a research-paper topic ontology to represent user interests in ontological terms. True, some of the fine-grained information held in the raw examples of interest is lost in this way. However, the ontology allows inference to assist user profiling through is-a relationships in the topic classification. Moreover, communication with other, external ontologies is enabled, and so is visualization of user profiles in terms of the topic ontology. Both systems provide a set of labeled example papers for each concept in the ontology to assist the creation of initial user profiles. The users themselves can add papers of interest to such sets in order to fine-tune their profiles and to reflect their changing needs. Through profile visualization, the users can better understand what the recommenders "think" about their interests and adjust their profiles interactively.

2.4 Advanced Topics

Being shared world models, content theories, representational artifacts of essential knowledge about topics and domains, and reusable building blocks of knowledge-based systems, ontologies are also tightly coupled to other concepts related to domain/world modeling, such as metadata and metamodeling. Being simultaneously concept hierarchies, ontologies also raise the question of how far the generalization/specialization in the hierarchies can extend.

2.4.1 Metadata, Metamodeling, and Ontologies

The prefix "*meta*" means one level of description higher. *Metadata* are data about data, and a *metamodel* is a model used to describe other models.

Metadata are descriptions of data, and metamodels are descriptions used to characterize models.

From a more practical point of view, metadata most often represent some description of digital resources, especially on the Web. Such a description is a mechanism for expressing the semantics of the resource, as a means to facilitate seeking, retrieval, understanding, and use of information. The nature and purpose of metadata are referential, that is, metadata express something about some resource. For example, Web documents, considered as a class of resources, are described in terms of their *authors*, *titles*, *URIs*, etc. A specific Web document is described by a metadata record containing metadata (characterizing all Web documents) and their specific values (characterizing that specific document only), i.e., by a set of attribute–value pairs (metadata–value pairs) such as “*author*–Denny, M.”, and “*title*–Ontology building: a survey of editing tools”.

Metadata can be expressed in various forms. For example, the *title* element of an HTML page represents embedded metadata. A *metadata language* is a shared description system that can be used to encode metadata.

One view of ontology representation languages is that they can be used as metadata languages – ontologies engineered with such languages can provide vocabularies for metadata to facilitate the management, discovery, and retrieval of resources on the Web. However, some consistency must be ensured in using terms defined in these ontologies as metadata. To an extent, this has been provided by standardization initiatives such as the Dublin Core Metadata Initiative [DC Metadata Schema, 2005], which promotes the widespread adoption of interoperable metadata standards and is developing specialized metadata vocabularies for describing various electronic resources. The purpose of such standards is to support both machine interoperability (information exchange) and targeted resource discovery by human users of the Web.

A metamodel is an explicit model of the constructs and rules needed to build specific models within a domain of interest. This characterizes a valid metamodel as an ontology, since such constructs and rules represent entities in a domain and their relationships, i.e., a set of building blocks used to build domain models. In other words, a metamodel is an ontology used by modelers. For example, when software developers use UML to construct models of software systems, they actually use an ontology implemented in it. This ontology defines concepts such as objects, classes, and relations. However, not all ontologies are modeled explicitly as metamodels.

2.4.2 Standard Upper Ontology

Imagine that someone wants to extend the simple Musician ontology shown in Figs. 2-2–2-4 and has included concepts such as *street musician*, *professional musician*, and *occasional musician* using a direct *is-a* link (relation) to the higher-level concept of *musician*. In the case of the Musician ontology, *musician* is also one of the top-level concepts. However, an obvious question might be: Is there an even higher-level, more general concept than *musician* that we can use as a generalization of the concept of *musician* through another direct *is-a* link? In other words, what concept extends the Musician ontology upwards, starting from the concept of *musician*?

Apart from the obvious dilemma of selecting the right concept to suit one’s purpose (is that concept *Performer*, *Profession*, or something else?), another problem quickly becomes apparent when one starts thinking about such extensions – how far up is it possible to go in this way? The problem can be reformulated as a series of questions such as:

- Is there a highest, root concept, and if so, what is it?
- What does the hierarchy of higher, i.e., upper-level, concepts look like?
- Who is responsible for those upper-level concepts?

The first question is easy to answer, but only technically – just as Java defines the *Object* class as the topmost class, it is possible to define such a root concept to model all of the upper-level concepts in a single upper-level ontology. In fact, there are examples of such technical approaches and even implementations of the concept of “topmost”; for instance, Protégé internally implements the concept of *Thing* as the root concept of all hierarchies and ontologies.

However, the problem is far more complex than a simple technical implementation, because it concerns a number of scientific and technical disciplines. Today, researchers in the fields of computer science, artificial intelligence, philosophy, library science, and linguistics are making attempts to formulate a comprehensive, formal, upper-level ontology to provide definitions for general-purpose terms and act as a foundation for more specific domain ontologies. However, workers in none of these fields have been able, on their own, to construct a *standard* upper-level ontology [Niles & Pease, 2001].

It is intuitively clear that such an endeavor must be a collaborative effort of specialists from different fields, but that fact creates a consensus problem – philosophers themselves have not yet reached a consensus on what the upper-level concept hierarchy should look like, let alone the question of a consensus among experts from other disciplines. Fig. 2-6

shows some of the proposed hierarchies (root concepts shown in bold face), but the second of the questions listed above is still open. Although the hierarchies represented may appear to be trees, in many approaches they are actually more complex structures, i.e., lattices. In Fig. 2-6c, for example, a lattice is implicitly present in the fact that the Collection concept appears in two different branches, which, in terms of frame-based languages, implies multiple inheritance. For the sake of simplicity, the representation in Fig. 2-6b is not shown here in its expanded version, which is actually also a complex lattice [Sowa, 2000]. Its root concept, τ , was originally defined as *universal type* that subsumes all other concepts.

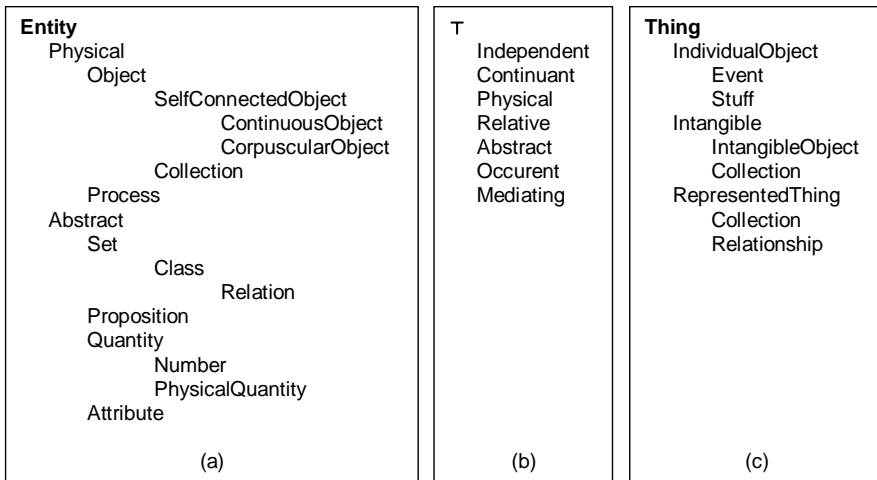


Fig. 2-6. Top levels of various concept hierarchies for the upper-level ontology: (a) according to [Niles & Pease, 2001]; (b) according to [Sowa, 2000]; (c) according to [Lenat & Guha, 1990; Lenat, 1995].

It is clear from Fig. 2-6 that a consensus is so much lacking that even the term used for the root concept is different in different representations. The lack of a consensus arises for an obvious reason: it is extremely difficult to conceive such a very large, wide-coverage ontology, in spite of the fact that there is a need for one and that there already exist the necessary formal ontology representation languages.

Going back to practice, the need for a standard upper-level ontology was the driving force that caused the IEEE to organize the Standard Upper Ontology Working Group (SUO WG) to develop such a standard ontology under its project P1600.1 [SUO WG, 2005]. This is, at least at the moment, an answer to the third of the questions listed above. The *standard upper ontology* (SUO) that will result from this project is limited to metaconcepts

and generic, abstract, and philosophical concepts, which therefore are general enough to address (at a high level) a broad range of domain areas. It is intended to provide a structure and a set of general concepts upon which domain ontologies (e.g. medical, financial, or engineering) could be constructed. The collaborators in the SUO WG come from various fields of engineering, philosophy, and information science. The group has started from several upper-level ontologies suggested in the literature so far (e.g., SUMO (Suggested Upper Merged Ontology) [Niles & Pease, 2001], the CYC ontology [Lenat & Guha, 1990; Lenat, 1995], the upper-level ontology described by Russell and Norvig [Russell & Norvig, 2002], and John Sowa's upper-level ontology [Sowa, 2000]) and is trying to accommodate most of their concepts and relations under the same umbrella. It is estimated that the SUO will eventually contain between 1000 and 2500 terms and roughly ten definitional statements for each term. Currently, the representation language used for the SUO is a somewhat simplified version of KIF, called SUO-KIF.

When completed, the SUO is envisioned to have a variety of purposes, such as the design of new knowledge bases and databases, reuse/integration of legacy databases (by mapping their data elements to a common ontology), and integration of domain-specific ontologies. Since the SUO is an IEEE-sponsored open-source standards effort, it is hoped that it will eventually be embraced by a large class of users. Also, the SUO is based on very pragmatic principles and any distinctions of strictly philosophical interest have been removed from it [Niles & Pease, 2001]; because of this, it should be simpler to use than some of the other upper-level ontologies that the working group started from.

The point is that the SUO will contribute to the goal of enhancing application interoperability by providing a common framework for the integration and sharing of different databases, knowledge bases, and domain ontologies, by virtue of shared terms and definitions. An important implication is that all other SUO-compliant ontologies that may exist under such a common ontological framework will be interconnected in some way.

2.4.3 Ontological Level

Recalling Allen Newell's idea of describing knowledge representation in terms of the implementation, logical, and knowledge levels (Sect. 1-8), one may ask a simple question: Where do ontologies come in this three-layer scheme? They do have a formal representation and structure, and thus might be thought of as parts of the logical level. On the other hand, they

also represent the essential concepts and relations in a domain, and all of the other domain knowledge relies on them, which may qualify them as parts of the knowledge level.

The dilemma was resolved by Guarino [Guarino, 1995], who proposed the notion of the *ontological level* and placed it as an intermediate level in between the logical and knowledge levels. The purpose of the ontological level is to constrain and to make explicit the intended model of the knowledge representation language, i.e., the intended meaning of the primitives of the knowledge representation formalisms used.

As an example, consider the following expression at the logical level: $\exists x. \textit{Musician}(x) \wedge \textit{Beginner}(x)$. At the knowledge level, this expression may be intended to mean that *Musician* is a concept and that *Beginner* is the value (filler) of the property *Level*. However, it is also perfectly valid in the real world to think of *Beginner* as a concept as well. If the modeler wants to make sure that the system interprets *Beginner* as the value of the property *Level*, and not as another concept, then it is actually the ontology that must explicitly state so, thus explicitly constraining the possible interpretations.

3. The Semantic Web

The Semantic Web is the new-generation Web that tries to represent information such that it can be used by machines not just for display purposes, but for automation, integration, and reuse across applications [Boley et al., 2001]. It has been one of the hottest R&D topics in recent years in the AI community, as well as in the Internet community – the Semantic Web is an important W3C activity [SW Activity, 2005].

Semantic Web is about making the Web more understandable by machines [Heflin & Hendler, 2001]. It is also about building an appropriate infrastructure for intelligent agents to run around the Web performing complex actions for their users [Hendler, 2001]. In order to do that, agents must retrieve and manipulate pertinent information, which requires seamless agent integration with the Web and taking full advantage of the existing infrastructure (such as message sending, security, authentication, directory services, and application service frameworks) [Scott Cost et al., 2002]. Furthermore, Semantic Web is about explicitly declaring the knowledge embedded in many Web-based applications, integrating information in an intelligent way, providing semantic-based access to the Internet, and extracting information from texts [Gómez-Pérez & Corcho, 2002]. Ultimately, Semantic Web is about how to implement reliable, large-scale interoperation of Web services, to make such services computer interpretable – to create a Web of machine-understandable and interoperable services that intelligent agents can discover, execute, and compose automatically [McIlraith et al., 2001]. [Devedžić, 2004a]¹

¹ Paragraph reprinted (with minor adjustments to citation formatting) with permission from IOS Press.

3.1 Rationale²

Why do we need all that? Isn't the Web an immense, practically unlimited source of information and knowledge that everyone can use?

The problem is that the Web is huge, but not smart enough to easily integrate all of those numerous pieces of information from the Web that a user really needs. Such integration at a high, user-oriented level is desirable in nearly all uses of the Web. Today, most of the Web information is represented in natural-language; however, our computers cannot understand and interpret its meaning. Humans themselves can process only a tiny fraction of information available on the Web, and would benefit enormously if they could turn to machines for help in processing and analyzing the Web contents [Noy et al., 2001]. Unfortunately, the Web was built for human consumption, not for machine consumption – although everything on the Web is *machine-readable*, it is not *machine-understandable* [Lassila, 1998]. We need the Semantic Web to express information in a precise, machine-interpretable form, ready for software agents to process, share, and reuse it, as well as to understand what the terms describing the data mean. That would enable Web-based applications to interoperate both on the syntactic and the semantic level.

Note that it is Tim Berners-Lee himself who pushes the idea of the Semantic Web forward. The father of the Web first envisioned a Semantic Web that provides automated information access based on machine-processable semantics of data and heuristics that use these metadata [Berners-Lee et al., 1999; Berners-Lee et al., 2001]. The explicit representation of the semantics of data, accompanied with domain theories (that is, ontologies), will enable a Web that provides a qualitatively new level of service – for example, intelligent search engines, information brokers, and information filters [Decker et al., 2000; Fensel & Musen, 2001].

People from W3C already develop new technologies for Web-friendly data description. Moreover, AI people have already developed some useful applications and tools for the Semantic Web [Noy et al., 2001; Scott Cost et al., 2002].

There is a number of important issues related to the Semantic Web. Roughly speaking, they belong to four categories: languages for the Semantic Web, ontologies, semantic markup of pages on the Semantic Web, and services that the Semantic Web is supposed to provide.

² Reprinted (with minor adjustments to citation formatting) from [Devedžić, 2004a] with permission from IOS Press.

3.2 Semantic Web Languages

In the literature, the Web-based ontology languages listed in Chap. 2 are usually called Semantic Web languages as well. However, W3C is more specific about this

The Semantic Web Activity develops specifications for technologies that are ready for large scale deployment, and identifies infrastructure components through open source advanced development. The principal technologies of the Semantic Web fit into a set of layered specifications. The current components are the Resource Description Framework (RDF) Core Model, the RDF Schema language and the Web Ontology language (OWL). Building on these core components is a standardized query language, SPARQL, enabling the “joining” of decentralized collections of RDF data. These languages all build on the foundation of URIs, XML, and XML namespaces. [W3C SW Activity, 2005]

The above statement is a rough textual equivalent of Tim Berners-Lee's vision of Web development, aptly nicknamed the “Semantic Web layer-cake”, Fig. 3-1 [Berners-Lee et al., 1999; Berners-Lee et al., 2001]. Note that in the Semantic Web layer-cake, higher-level languages use the syntax and semantics of the lower levels. All Semantic Web languages use XML syntax; in fact, XML is a metalanguage for representing other Semantic Web languages. For example, XML Schema defines a class of XML documents using the XML syntax. RDF provides a framework for representing metadata about Web resources, and is XML-based as well. RDF Schema, OWL, and other ontology languages also use the XML syntax.

The following subsections discuss the Semantic Web layer-cake in more detail and depict the genesis of the current state of the art in the domain of Semantic Web languages.

3.2.1 XML and XML Schema

It is important for Semantic Web developers to agree on the data's syntax and semantics before hard-coding them into their applications, since changes to syntax and semantics necessitate expensive modifications of applications [Wuwongse et al., 2002].

Current Semantic Web languages rely on an XML-based syntax [Klein, 2001; XML, 2005]. Generally, XML (eXtensible Markup Language) enables the specification and markup of computer-readable documents. It

looks very much like HTML in that special sequences of characters – tags – are used to mark up the document content, and that XML data is stored as ordinary text. Unlike HTML, however, XML can be used to annotate documents of arbitrary structure, and there is no fixed tag vocabulary. Typically, XML tags contain information indicating the human interpretation of pieces of the document's content, such as `<name>`, `<musician>`, and `<record>`. Thus XML lets people *meaningfully* annotate documents by adding context to and indicating the meaning of the data. People can define their own custom tags to represent data logically, making XML documents self-describing (because the tags describe the information the documents contain).

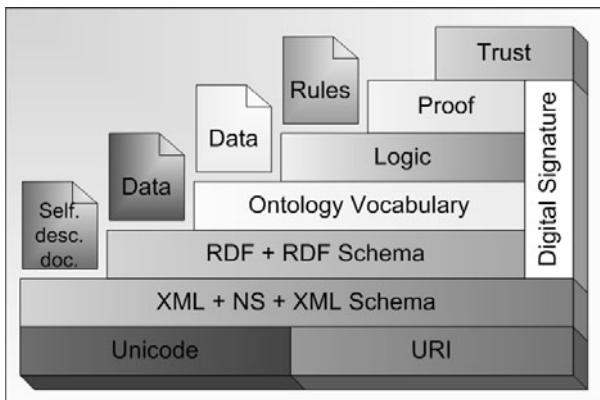


Fig. 3-1. Tim Berners-Lee's Semantic Web layer-cake

Fig. 3-2 shows an example of how the same piece of information can be represented in HTML and in XML. Obviously, HTML is layout-oriented, whereas XML is more structure-oriented.

<pre> Eric Clapton, Unplugged, Reprise/WEA, 1992. </pre> <p style="text-align: right;">(a)</p>
<pre><ALBUM> <AUTHOR> Eric Clapton </AUTHOR> <TITLE> Unplugged </TITLE> <LABEL> Reprise/WEA </LABEL> <YEAR> 1992 </YEAR> </ALBUM></pre> <p style="text-align: right;">(b)</p>

Fig. 3-2. (a) A piece of HTML code (b) The same information in XML code

However, XML does not itself imply a specific machine interpretation of the data. In the piece of XML in Fig. 3-2b, the meaning is not formally specified. The information is only encoded in an unambiguous syntax, but its use and the semantics are not specified. In other words, XML is aimed only at the structure of a document, not at a common machine interpretation of it. It provides only a data format for structured documents, without specifying a vocabulary.

On the other hand, owing to the standardized data format and structure of XML documents, programs and scripts can dynamically access and update the content, structure, and style of such documents [DOM, 2005]. Appropriate parsers and other processing tools for XML documents are readily available. Moreover, XML is extensible in a standardized way, and hence enables customized markup languages to be defined for unlimited types of documents.

Using XML for document and data exchange among applications requires prior agreement on the vocabulary, its use, and the meaning of its terms. Such agreement can be partly achieved by using XML schemas, which provide a mechanism to specify the structure of XML documents. Every XML schema provides the necessary framework for creating a category of XML documents [XML Schema, 2005]. The schema describes the various tags, elements, and attributes of an XML document of that specific category, the valid document structure, the constraints, and the custom data types (these are based on built-in types, such as integer and string). The XML Schema language also provides some limited support for specifying the number of occurrences of child elements, default values, choice groups, etc. The encoding syntax of the XML Schema language is XML, and just like XML itself XML Schema documents use *namespaces* that are declared using the *xmlns* attribute. The example schema in Fig. 3-3 declares the *xsd* namespace and uses it throughout its element definitions. Namespaces define contexts within which the corresponding tags and names apply. The schema shown corresponds to a category of XML documents such as that shown in Fig. 3-2b.

```
<xsd:schema xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<xsd:element name="ALBUM" type="ALBUMTYPE"/>
<xsd:complexType name="ALBUMTYPE" >
  <xsd:element name="AUTHOR" type="xsd:string"
    minOccurs="1" maxOccurs="unbounded"/>
  <xsd:element name="TITLE" type="xsd:string"/>
  ...
  <xsd:element name="YEAR" type="xsd:integer"/>
</xsd:complexType>
</xsd:schema>
```

Fig. 3-3. An example of an XML schema

3.2.2 RDF and RDF Schema

XML provides an easy-to-use syntax for encoding all of the kinds of data that are exchanged between computers, by using XML schemas to prescribe the data structure. However, since it does not provide any interpretation of the data beforehand, it does not contribute much to the semantic aspect of the Semantic Web. To provide machine interpretation of Web data, a standard model is needed to describe facts about Web resources. Such a standard model can be specified by use of RDF and RDF Schema.

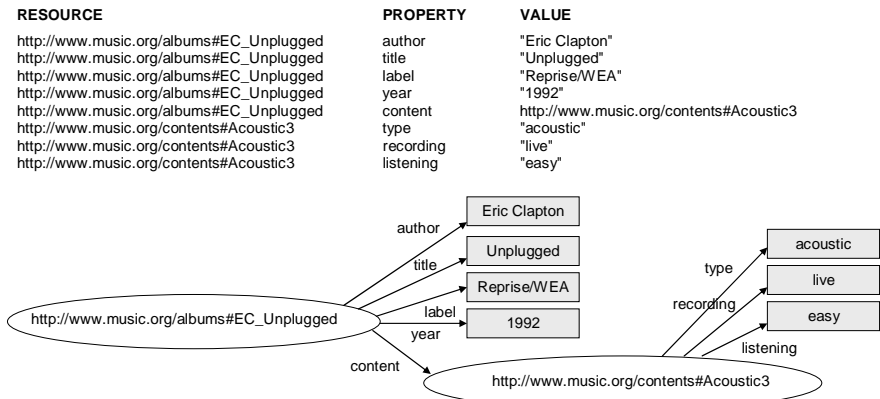


Fig. 3-4. Examples of RDF resources, properties, and values, and the corresponding graph

RDF’s model for representing data about “things on the Web” (resources) is that of O–A–V triplets and semantic networks. A resource description in RDF is a list of *statements* (triplets), each expressed in terms of a Web *resource* (an object), one of its *properties* (attributes), and the *value* of the property [Manola & Miller, 2004]. The value can be a literal (text), or another resource. Every RDF description can be also represented as a directed labeled graph (a semantic network), parts of which are equivalent to RDF statements. Figure 3-4 shows several such triplets and the corresponding graph. These can be represented in an RDF encoding (which also uses XML syntax) as in Fig. 3-5.

The fact that the value of a property in an RDF encoding can be either a literal value or another resource creates a *stripping pattern* in the RDF encoding format (Fig. 3-6). This makes it easy to check the overall consistency of an RDF-encoded document. It also makes RDF a suitable

format for visual knowledge representation languages based on semantic networks.

```

<Album rdf:ID="EC_Unplugged"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns="http://www.music.org/albums#"
  xml:base="http://www.music.org/albums">
  <author>Eric Clapton</author>
  <title>Unplugged</title>
  <label>Reprise/WEA</label>
  <year>1992</year>
  <content>
    <Content rdf:ID="Acoustic3"
      xmlns="http://www.music.org/contents#">
      <type>acoustic</type>
      <listening>easy</listening>
      <recording>live</recording>
    </Content>
  </content>
</Album>

```

Fig. 3-5. RDF encoding of the resources shown in Fig. 3-4

An RDF model itself provides only a domain-neutral mechanism to describe individual resources. It neither defines (a priori) the semantics of any application domain, nor makes assumptions about a particular domain. Defining domain-specific features and their semantics, i.e., ontologies, requires additional facilities. RDF itself is used to describe instances of ontologies, whereas RDF Schema encodes ontologies.

RDF Schema (RDFS) provides an XML-based vocabulary to specify classes and their relationships, to define properties and associate them with classes, and to enable the creation of taxonomies [Brickley & Guha, 2004]. To do all this, RDFS uses frame-based modeling primitives such as *Class*, *subClassOf*, *Property*, and *subPropertyOf*. The *Resource* concept occurs in the root of all hierarchies and taxonomies. Figure 3-7 shows an example of an RDFS encoding.

There is an important departure in RDFS from the classical frame-based paradigm: properties are defined separately from classes. An implication is that anyone, anywhere, anytime can create a property and state that it is usable with a class, or with multiple classes. Each property is typically described by *rdfs:domain* and *rdfs:range*, which restrict the possible combinations of properties and classes. For example, in Fig. 3-7 the domain of the property *year* is restricted to the class *Album*, which means that this property is used only with that class. On the other hand, a property may be defined so as to feature multiple classes. As in the classical case, in class hierarchies classes inherit properties from their ancestors.

<?xml version="1.0"?>
<Resource-A>
<property-A>
<Resource-B>
<property-B>
<Resource-C>
<property-C>
Value-C
</property-C>
</Resource-C>
</property-B>
</Resource-B>
</property-A>
</Resource-A>

Fig. 3-6. The stripping pattern of an RDF model

```

<rdf:RDF      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
              xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
              xml:base="http://www.music.org/albums">

  <rdfs:Class rdf:ID="Album">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
  </rdfs:Class>

  <rdfs:Class rdf:ID="Content">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource"/>
  </rdfs:Class>

  ...

  <rdf:Property rdf:ID="author">
    <rdfs:domain rdf:resource="#Album"/>
    <rdfs:range rdf:resource="#Musician"/>
  </rdf:Property>

  ...

  <rdf:Property rdf:ID="year">
    <rdfs:domain rdf:resource="#Album"/>
    <rdfs:range rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
  </rdf:Property>

  ...

</rdf:RDF>

```

Fig. 3-7. RDFS encoding of albums (excerpt)

RDF(S) provides a standard model to describe facts about Web resources, but modelers often need even richer and more expressive primitives to specify the formal semantics of Web resources. RDFS is

quite simple compared with full-fledged knowledge representation languages. For example, one cannot state in RDFS that “this class is equivalent to this other class”, and cannot specify cardinality constraints.

3.2.3 DAML+OIL

DAML+OIL has evolved from two other languages, DAML-ONT [Hendler & McGuinness, 2000] and OIL [Fensel et al., 2001], both of which were heavily influenced by RDF(S). In the context of the Semantic Web layer-cake (Fig. 3-1), RDF(S) defines a simple model for representing semantics and basic ontological-modeling primitives, whereas DAML-ONT, OIL, and DAML+OIL all aim at providing a more expressive ontology development vocabulary. They offer richer ways to define concepts and attributes, and a more intuitive choice of some of the modeling primitives (widely used modeling primitives from logic-based and frame-based languages).

DAML-ONT [Hendler & McGuinness, 2000] was part of the DARPA Agent Markup Language (DAML) initiative, aimed at supporting the development of the Semantic Web. As an ontology language, it covered the capturing of definitions of terms – classes, subclasses, and their properties, as well as their restrictions and individual object descriptions. Another part of the DAML language (called DAML-LOGIC) addressed the issue of encoding inference and general logical implications. DAML-ONT stressed the role of ontological markup of Web resources to facilitate intercommunication between agents.

OIL (Ontology Inference Layer) originated from a European initiative; its formal semantics was based on description logics and it had a set of customized editors and inference engines to support working with it [Fensel et al., 2001]. One of OIL’s important emphases was on a layered approach to ontology language specification. *Core OIL* largely coincides with RDFS. This means that even simple RDFS agents can process Core OIL ontologies and understand their semantics, and that an OIL processor can understand RDFS. *Standard OIL* captures semantics and makes complete inference viable by means of the necessary mainstream modeling primitives, such as *Class-def* (equivalent to *rdfs:Class*), *Subclass-of* (*rdfs:subClassOf*), *Slot constraint* (*oil:hasSlotConstraint* and *oil:SlotConstraint*), *AND* (*oil:and*), *NOT* (*oil:not*), and *Has-value* (*oil:hasValue*). Furthermore, *Instance OIL* provides thorough integration of individuals through a full-fledged database capability, and *Heavy OIL* opens the language to the possibility of including even higher-level representational and reasoning needs, such as various rule languages and metaclasses.

```

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns   ="http://www.daml.org/2001/03/daml+oil-ex#"
>

<daml:Ontology rdf:about="">
  <rdfs:comment>
    An example simple DAML+OIL ontology.
  </rdfs:comment>
  <daml:imports rdf:resource="http://www.daml.org/2001/03/daml+oil"/>
</daml:Ontology>

<daml:Class rdf:ID="Musician">
  <rdfs:label>Musician</rdfs:label>
  <rdfs:comment>
    This class assumes just two disjoint kinds of musicians - instrumentalists and singers.
  </rdfs:comment>
</daml:Class>

<daml:Class rdf:ID="Album">
  <rdfs:label>Album</rdfs:label>
  <rdfs:comment>
    The class describing recorded albums.
  </rdfs:comment>
</daml:Class>

<daml:Class rdf:ID="Instrumentalist">
  <rdfs:subClassOf rdf:resource="#Musician"/>
</daml:Class>

<daml:Class rdf:ID="Singer">
  <rdfs:subClassOf rdf:resource="#Musician"/>
  <daml:disjointWith rdf:resource="#Instrumentalist"/>
</daml:Class>

...

</rdf:RDF>

```

Fig. 3-8. An example DAML+OIL ontology – header and classes (excerpt)

DAML+OIL [Horrocks & van Harmelen, 2002; Scott Cost et al., 2002] has merged the principles and advantages of DAML and OIL, in an effort to develop a universal Semantic Web language that can enable machines to read data and interpret and draw inferences from it. As a result, and in addition to providing rules and definitions similar to those of RDF(S), DAML+OIL also enables further constraints and relationships among resources to be specified, including cardinality, domain and range restrictions, and union, disjunction, inverse, and transitive rules.

```

<rdf:RDF
...
>

...

<daml:ObjectProperty rdf:ID="isAuthorOf">
  <rdfs:domain rdf:resource="#Musician"/>
  <rdfs:range rdf:resource="#Album"/>
</daml:ObjectProperty>

...

<daml:DatatypeProperty rdf:ID="year">
  <rdfs:comment>
    year is a DatatypeProperty whose range is xsd: nonNegativeInteger.
    year is also a UniqueProperty (there is only one year when an album is recorded)
  </rdfs:comment>
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:range rdf:resource="http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger"/>
</daml:DatatypeProperty>

...

<Musician rdf:ID="Eric Clapton">
  <rdfs:label>Eric Clapton</rdfs:label>
  <rdfs:comment> Eric Clapton is a musician.</rdfs:comment>
  <name><xsd:string rdf:value="Eric Clapton"/></name>
  ...
</Musician>

...

</rdf:RDF>

```

Fig. 3-9. An example DAML+OIL ontology – properties and individuals (excerpt)

DAML+OIL divides the universe into two disjoint parts – the *datatype domain* (the values that belong to XML Schema datatypes), and the *object domain* (individual objects, considered to be members of classes described within DAML+OIL or RDF). Likewise, there are generally two sorts of DAML+OIL properties – those that relate objects to other objects (specified by *daml:ObjectProperty*), and those that relate objects to datatype values (specified by *daml:DatatypeProperty*). The syntax for classes and properties is similar to that of DAML and OIL (Fig. 3-8 and 3-9); instances of classes and properties are written in RDF(S) syntax (Fig. 3-9).

3.2.4 OWL

OWL [Smith et al., 2004] is a successor to DAML+OIL (see Fig. 3-10). Like its predecessors, the OWL vocabulary includes a set of XML elements and attributes, with well-defined meanings. These are used to describe domain terms and their relationships in an ontology. In fact, the OWL vocabulary is built on top of the RDF(S) vocabulary. Things such as *Class* and *subClassOf* exist in OWL as well, and so do *ObjectProperty*, *DatatypeProperty*, and many more, as a heritage from DAML+OIL (see Fig. 3-11).

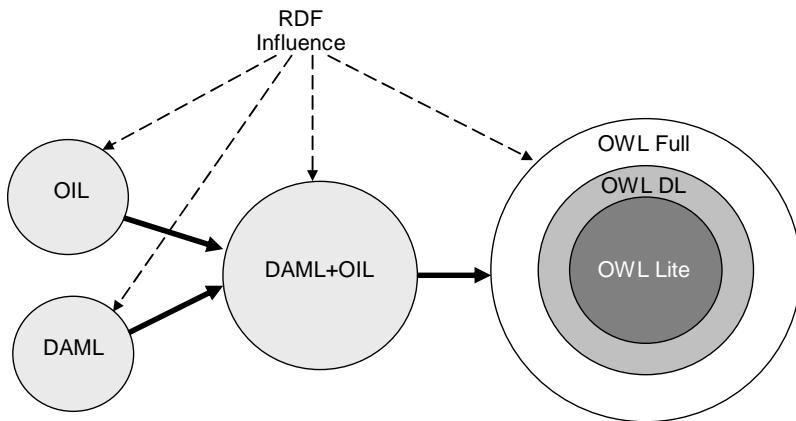


Fig. 3-10. The genesis of OWL

An important feature of the OWL vocabulary is its extreme richness for describing relations among classes, properties, and individuals. For example, we can specify in OWL that a property is, *Symmetric*, the *InverseOf* another one, an *equivalentProperty* of another one, and *Transitive*; that a certain property has some specific *cardinality*, or *minCardinality*, or *maxCardinality*; and that a class is defined to be an *intersectionOf* or a *unionOf* some other classes, and that it is a *complementOf* another class. Similarly, a class instance can be the *sameIndividualAs* another instance, or it can be required to be *differentFrom* some other instance. For example, using the *equivalentProperty* relation, the object properties *author* and *artist* are specified to be equivalent in Fig. 3-11. Thus, if an instance of *Album* specifies its *artist*, and an application “knowing” that an album must have an *author* consults the ontology to “understand” what the instance is about, it will infer that the *artist* specified in the instance is actually the *author* of the album. A nice consequence is that reasoning can be performed in spite of such terminological differences.

```

<?xml version="1.0"?>
<rdf:RDF
  xmlns="http://www.music.org/musicians.owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xml:base="http://www.music.org/musicians.owl">
  <owl:Ontology rdf:about="Musician"/>
  <owl:Class rdf:ID="Musician"/>
  <owl:Class rdf:ID="musician_Class_13">
    <rdfs:subClassOf rdf:resource="#Musician"/>
  </owl:Class>
  <owl:Class rdf:ID="Instrument"/>
  <owl:Class rdf:ID="Album">
    <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#title"/>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#year"/>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int">1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:ObjectProperty rdf:ID="author">
    <owl:equivalentProperty rdf:resource="#artist"/>
    <rdfs:domain rdf:resource="#Album"/>
    <rdfs:range rdf:resource="#Musician"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:about="#artist">
    <rdfs:range rdf:resource="#Musician"/>
    <rdfs:domain rdf:resource="#Album"/>
  </owl:ObjectProperty>
  ...

```

Fig. 3-11. Excerpt from the Musician ontology developed in Protégé

Another important DAML+OIL heritage is OWL's layered structure, also indicated in Fig. 3-10. In fact, OWL is not a closed language; it is, rather, a combination of three increasingly expressive sublanguages building on top of each other, designed to suit different communities of implementers and users. *OWL Lite* is intended to support the building of simple classification hierarchies and simple constraints. To this end, the ability to specify constraints in OWL Lite is rather restricted; for example, the only cardinality values permitted in OWL Lite are 0 and 1. *OWL DL* reflects the description-logic foundation of its predecessor, DAML+OIL. OWL DL provides the maximum expressiveness, but also guarantees that all conclusions are computable and will finish in a finite time. It includes all OWL language constructs, although it imposes certain restrictions on

using them. *OWL Full* supports users who want maximum expressiveness and the syntactic freedom of RDF, but does not guarantee computational completeness and decidability. OWL Full can be viewed as an extension of RDF, whereas OWL Lite and OWL DL can be viewed as extensions of a restricted view of RDF (see [Smith et al., 2004] for details).

3.2.5 SPARQL

Unlike OWL and RDF(S), SPARQL is not intended for ontology and resource representation, but for querying Web data; precisely, it is a query language for RDF [SPARQL, 2005].

To understand SPARQL, the view of RDF resources as semantic networks (set of triplets; see Fig. 3-4) helps. SPARQL can be used to:

- extract information from RDF graphs in the form of URIs, bNodes, and plain and typed literals;
- extract RDF subgraphs;
- construct new RDF graphs based on information in queried graphs.

SPARQL queries match graph patterns against the target graph of the query. The patterns are like RDF graphs, but may contain named variables in place of some of the nodes (resources) or links/predicates (i.e., properties). The simplest graph pattern is like a single RDF triplet (resource–property–value triplet, or O–A–V triplet). For example, consider the two RDF triplets in Fig. 3-12. Clearly, they both match the simple triplet pattern shown in Fig. 3-13. A *binding* is a mapping from a variable in a query to terms. Each triplet in Fig. 3-12 is a *pattern solution* (a set of correct bindings) for the pattern in Fig. 3-13. *Query results* in SPARQL are sets of pattern solutions. The results of the query represented by the pattern in Fig. 3-13 are the following pattern solutions:

album

http://www.music.org/albums#EC_Unplugged

http://www.music.org/albums#PG_UP

author

Eric Clapton

Peter Gabriel

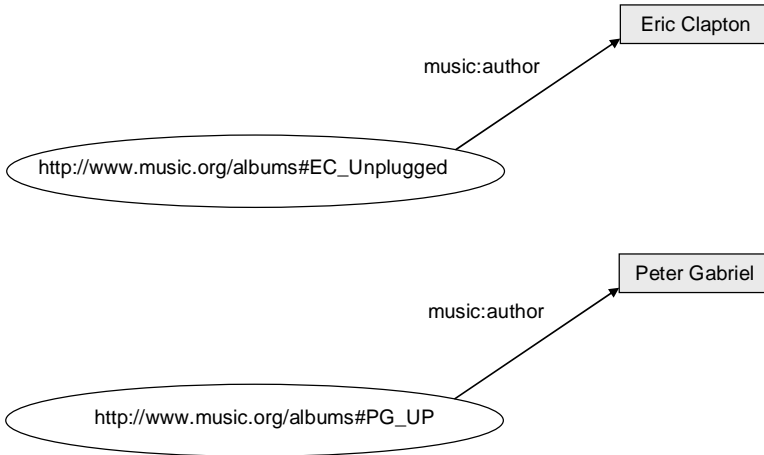


Fig. 3-12. Simple RDF triplets

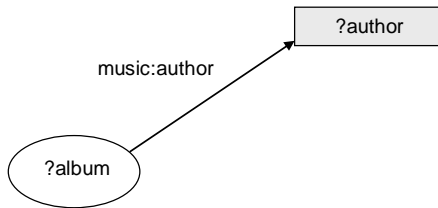


Fig. 3-13. A simple RDF triplet pattern

Simple graph patterns can be combined using various operators into more complicated graph patterns. For example, the graph in Fig. 3-14 matches the more complex pattern shown in Fig. 3-15, and the pattern solution is

album	<code>http://www.music.org/albums#EC_Unplugged</code>
ccontent	<code>http://www.music.org/contents#Acoustic3</code>
recording	live

Syntactically, SPARQL queries are of the form presented in Fig. 3-16. Obviously, the syntax closely resembles that of database query languages such as SQL. The SELECT clause contains variables, beginning with “?” or “\$”. The WHERE clause contains a pattern. Prefixes are used as an abbreviation mechanism for URIs and apply to the whole query.

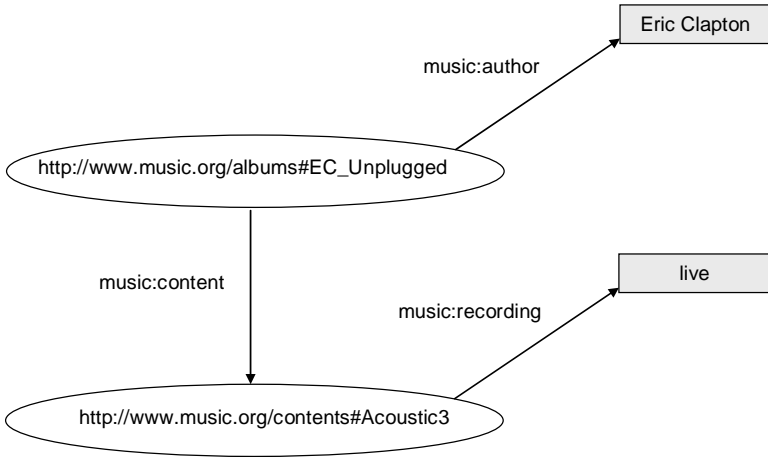


Fig. 3-14. A more complex RDF graph

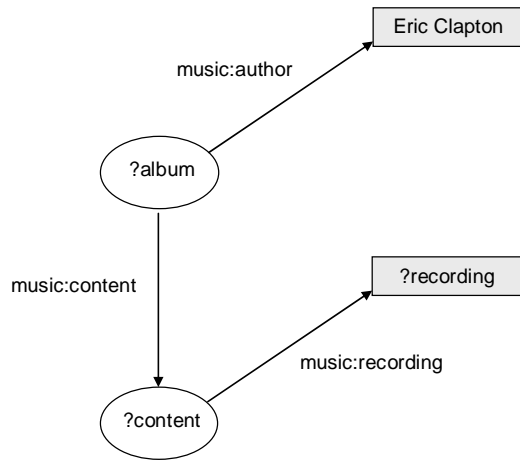


Fig. 3-15. A more complex SPARQL pattern

```

SELECT ?author
WHERE { <http://www.music.org/albums#EC_Unplugged> <http://www.music.org/elements/author> ?author }

PREFIX music: <http://www.music.org/elements/>
SELECT ?author
WHERE { <http://www.music.org/albums#EC_Unplugged> music:author ?author }

PREFIX music: <http://www.music.org/elements/>
PREFIX : <http://www.music.org/albums/>
SELECT $author
WHERE { :EC_Unplugged music:author $author }
    
```

Fig. 3-16. Examples of SPARQL queries

3.3 The Role of Ontologies

Another direct answer to the question “Why ontologies?” put in Chap. 2 is “Because they are essential building blocks in the infrastructure of the Semantic Web”. Semantic-level interoperation among Web applications is possible only if the semantics of Web data is explicitly represented on the Web as well, in the form of machine-understandable domain and content theories – ontologies. Through automatic use and machine interpretation of ontologies, computers themselves can offer enhanced support and automation in accessing and processing Web information. This is qualitatively different from the established practices of using the Web, in terms of extracting and interpreting information – instead of putting the main burden on the user, the Semantic Web should do much of that job itself [Fensel & Musen, 2001].

Ontologies enable access to a huge network of machine-understandable and machine-processable human knowledge (Fig. 3-17), encoded in XML-based formats. Once the essential knowledge of a certain domain has been put on the Web in the form of interconnecting ontologies, it creates a solid basis for further development of intelligent applications in that domain because it alleviates the problem of knowledge acquisition.

More specifically, ontologies play multiple roles in the architecture of the Semantic Web (see Fig. 3-1):

- they enable Web-based knowledge processing, sharing, and reuse between applications, by the sharing of common concepts and the specialization of the concepts and vocabulary for reuse across multiple applications;
- they establish further levels of interoperability (semantic interoperability) on the Web in terms of mappings between terms within the data, which requires content analysis;
- they add a further representation and inference layer on top of the Web's current layers;
- they enable intelligent services (information brokers, search agents, information filters, intelligent information integration, knowledge management, ...; see Sect. 3-5 for details).

Note, however, that the prerequisites for all of the above roles include not only an initial effort by the interested communities to create ontologies, but also considerable discipline in annotating the relevant applications and Web resources to get them interconnected with the ontologies (see Sect. 3-4). Also, supporting tools are needed for those millions of developers of Web pages and applications who weave their

domain knowledge into the Web daily. Using knowledge representation techniques in such tools becomes increasingly important. Last but not least, an all-encompassing framework for developing the network of ontologies and interconnecting them across domains is also highly desirable. Efforts are under way to provide such a framework in the form of a standard upper ontology, as discussed in Sect. 2.4.2.

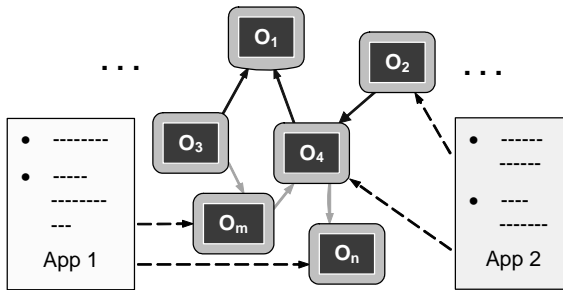


Fig. 3-17. Interconnecting ontologies and applications on the Semantic Web

With ontologies, the Semantic Web provides a qualitatively new level of service, as it becomes an extremely large system with various specialized reasoning services. Ontologies provide an infrastructure for transforming the Web of information and data into the Web of knowledge – the Semantic Web.

3.4 Semantic Markup

Ontologies serve merely to standardize and provide interpretations for Web content. To make content machine-understandable, Web resources must contain *semantic markup*, or *semantic annotation* – descriptions which use the terminology that one or more ontologies define [Heflin & Hendler, 2001]. Such ontologically annotated Web resources enable reasoning about their content and advanced query-answering services. They also support ontology creation and maintenance, and help map between different ontologies.

Through the process of semantic markup of Web resources, information is added to the resources without changing the originals. The added information may serve human users in much the same way as highlighted text does in paper-based documents, or, more importantly, may enhance the semantic analysis and processing of Web resources by computers. For

example, semantic annotation may help intelligent agents discover Web resources more easily, or it may indicate that the content of different resources is semantically similar. Also, adequate metadata about the semantic markup added to a resource might help search engines locate the right information.

There are several levels of sophistication in annotating Web resources and making applications use the markup. The simplest approach is to use annotation tools to mark up downloaded Web pages manually and save the annotation together with the pages locally. Typically, such annotation is in the form of highlighted text, new elements inserted into the document, and hyperlinks. However, Web servers do not normally allow annotated Web resources to be uploaded back. A more sophisticated approach is to save the markup in a separate document (locally or on a remote server) and load it in a browser along with the document. The next step up in sophistication is provided by collaborative Wiki sites that let their users insert and share their comments and other annotations along with Web pages.

None of the above cases has to be supported by explicitly represented ontologies. As a consequence, such approaches have been aimed mainly at supporting human users. On the other hand, if ontologies are used to drive the markup creation, then machine consumers can make use of annotations as well. Ontology-based annotation tools enable unstructured and semistructured information sources to be linked with ontologies.

There are numerous approaches to ontology-based markup [Handschuh & Staab, 2003a]. As an illustration, consider how semantic markup is done according to the CREAM framework [Handschuh & Staab, 2002; Handschuh & Staab, 2003b]. This framework is suitable for both annotation of existing Web pages and annotation of content while authoring a Web page. The key concept in the CREAM framework is *relational metadata*, i.e., metadata that instantiate interrelated definitions of classes in a domain ontology. More precisely, for various instantiations of classes and properties in an ontology, there may exist several semantic relationships; relational metadata are annotations that contain relationship instances. The annotations are represented as an XML serialization of RDF facts and are attached to HTML pages as in the hypothetical, simplified example shown in Fig. 3-18. Assume that a Web page at the hypothetical URL <http://www.guitar.org/legendaryrecordings> contains information about the album *Unplugged* by Eric Clapton, whose homepage is at <http://www.ericclapton.com/>. Assume also that the Musician ontology sketched in Fig. 2-2 is implemented in OWL and used to annotate these two Web pages. Furthermore, let two namespaces be defined as follows:

load and display an existing one to be annotated. In either case, while editing the Web page the author can also load an ontology to be used for markup, and display its term hierarchies and concept/attribute descriptions graphically. By selecting parts of the content of the Web page being edited and connecting them with the terms in the ontology by simple mouse clicks and drag-and-drop operations, the author can produce the markup almost for free – not much additional effort is needed to insert the annotations with such a graphical tool. When the author saves the Web page at the end of the process, the markup is saved as well.

The CREAM framework has evolved over the years [Handschuh et al., 2003a; Handschuh et al., 2003b] to enable querying a Web site semantically about the resources it publishes. The resources are typically kept in a database on a server, and the site developer may annotate the database model (i.e., the entities and their relationships) and publish the annotation in RDF. The annotation will describe the structure of all the tables involved in a query of the site, thus acting as an API that hides the intricacies of access to the database on the server side. A suitable place for publishing the annotation is the header part of the site's main Web page. A client may wish to use a specific ontology to put semantic queries to the server. To do this, the client must first create rules for mapping between the terms in his/her ontology and the terms that are used in the database model on the server side and published there as a set of RDF annotations. The tool called OntoEdit can be used to create and publish the mapping rules. The user (or a third party) can then load both the ontology and the mapping rules to query the database, for example through a Web-service API.

We can conclude from the above example and from [Heflin & Hendler, 2001; Hendler, 2001] that in order for annotation of Web pages with ontological information to be effective:

- nonexperts in ontological engineering must be able to perform this task, starting from existing ontologies, transparently, through normal computer use;
- most Web page developers should not need even know that ontologies exist, but should still do (almost) free markup;
- ontology-driven authoring tools should support both authoring and annotation processes by enabling class hierarchies based on a number of underlying ontologies to drive the creation of Web pages;
- the content of the pages being designed and developed should be presented, modified, and mixed consistently, using ontologies linked to libraries of terms, and interlinked in order to reuse or change terms;

- tool developers should enable libraries of ontologies to be accessed from the tools that they develop, in order to support appropriate markup of pages in a wide range of domains.

3.5 Semantic Web Services

Roughly speaking, *Web services* are activities that allow both end users and, under appropriate circumstances, software agents to invoke them directly [Preece & Decker, 2002]. In the traditional Web model, users follow hypertext links manually. In the Web-services model, they invoke tasks that facilitate some useful activity (e.g., meaningful content-based discovery of resources, fusion of similar content from multiple sites, or commercial activities such as course advertising and registration for distance learning) [Devedžić, 2004b].

Technically, Web services are autonomous, platform-independent computational elements that can be described, published, discovered, orchestrated, and programmed using XML artifacts for the purpose of developing massively distributed interoperable applications. The platform-neutral and self-describing nature of Web services and, particularly, their ability to automate collaboration between Web applications make them more than just software components. In the *service-oriented architecture* [Vinoski, 2002] (Fig. 3-19), Web services advertise themselves in a registry, allowing client applications to query the registry for details of the service and interact with the service using those details.

The service-oriented architecture shown in Fig. 3-19 can greatly enhance the traditional development process in the case of the development of Web applications, since the client-side system can be built on the basis of Web services even if those services are not yet available or are not known by the developers. This is due to the fact that every Web service is described through a service description language, dynamically discovered by applications that need to use it, and invoked through the communication protocol defined in its interface [Vinoski, 2002]. The central component in Fig. 3-19 – the service directory – is a dynamically organized, but highly structured (e.g., as a tree, or as a table/database) information pool pertaining to various services. The underlying assumption is that at every point in time the directory lists those services that are ready to be invoked by the user; these services are assumed to advertise their readiness and availability to the directory. Hence an agent can find out about the available services by looking up the directory. Then it can decide whether to automatically invoke a suitable service on the

user's behalf, or merely suggest that the user interacts with the service directly.

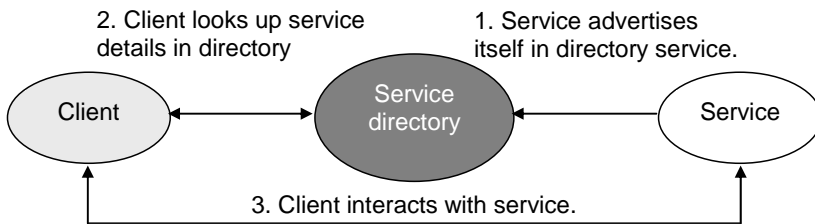


Fig. 3-19. Service-oriented architecture

There is a lot of supporting technology for developing, publishing, and using Web services, such as WSDL (Web Services Description Language), WSFL (Web Services Flow Language), UDDI (Universal Description, Discovery, and Integration), and SOAP (Simple Object Access Protocol). See [Preece & Decker, 2002] for starting points on the use of these technologies.

Note, however, that in the Semantic Web the idea is to employ *intelligent* Web services to go beyond the XML/RDF infrastructure of Web pages, i.e., to explore Web services that intelligent-systems technology can make possible. Intelligent Web services may turn the Web into a collection of different resources, each with a well-defined interface for invoking its services [Vinoski, 2002]. In other words, intelligent Web services deploy intelligent-systems techniques to perform useful, reusable tasks for Web users. This view of Web services implies that the properties, capabilities, interfaces, and effects of Web services must be encoded in an unambiguous, machine-understandable form, and properly marked up to make the services computer-interpretable, with a use that is apparent, and agent-ready [McIlraith et al., 2001]. Such requirements, in turn, imply the need for ontologies of Web services, to be used as machine-readable descriptions of services (as to how they run), including the consequences of using those services. Each such ontology should explicitly represent the logic of the services and the terms needed to describe the invocation of those services. Web service ontologies bring intelligence to Web services, as they enable integration of agents and ontologies in some exciting ways. For example, an agent performing a keyword-based Web search may invoke services such as controlled vocabularies that enable fuzzy-terms-based searches and inexact matches; if the requested keyword is not in the dictionary, the service can come up immediately with a more general concept suggested in the ontology.

The difference between conventional and intelligent Web services is best understood through the pragmatics of their use. In the conventional case, the user has to discover the desired service first (using a search engine). In most cases, this involves a lot of reading of the Web pages discovered. Alternatively, the user may execute a service to see whether it satisfies his/her request; this, in turn, means filling in the forms of the service manually and composing manually the sequence of services required to complete a complex task.

On the other hand, intelligent Web services enable automatic service discovery, using preprovided semantic markup of Web pages and ontology-enhanced search engines. Intelligent agents can execute such services on behalf of their users automatically, since the semantic markup of a service provides a declarative API that tells the agent what input is necessary for automatic invocation, what information will be returned, and how to execute and, potentially, interact with the service automatically. Automatic service composition and interoperation are also provided, since the semantic markup of services provides all the necessary information to select, compose, and respond to services. The markup is encoded and stored at the sites of the services, and appropriate software tools manipulate the markup together with specifications of the service task's objectives.

Obviously, the real power of intelligent Web services results not from their individual use, but from combining them in a variety of ways [Preece & Decker, 2002]. This creates the need for standard models of interaction between services [McIlraith et al., 2001]. Such models should be implemented as declarative, machine-processable descriptions of how to combine intelligent Web services to achieve more sophisticated tasks. The descriptions can be encoded in Web service composition languages such as WSFL or OWL-S (see below), and contain the knowledge about how to perform the sophisticated real-life tasks that the services perform [Preece & Decker, 2002]. The point is that implementing these composition descriptions on the Web makes them downloadable, understandable, and executable for everyone, not only humans but also automated agents.

Recently, the idea of intelligent Web services has evolved slightly into the concept of *Semantic Web services* [Payne & Lassila, 2004] in the form of an augmentation of Web Service descriptions through Semantic Web annotation to facilitate higher-level automation of service discovery, composition, invocation, and monitoring in an open, unregulated, and often chaotic environment (that is, the Web). The objective of Semantic Web services is to provide a ubiquitous infrastructure for deploying intelligent multiagent systems on the Web.

The Semantic Web community has already developed *OWL-S*, an OWL-based ontology of Web services and a core set of markup language constructs for describing the properties and capabilities of Web services in unambiguous, computer-intepretable form [Martin et al., 2004]. OWL-S comes with supporting tools and agent technology to enable automation of services on the Semantic Web, including automated Web service discovery, execution, interoperation, composition, and execution monitoring.

Conceptually, the top level of the OWL-S ontology looks as in Fig. 3-20 [Martin et al., 2004]. The *ServiceProfile* describes what a service does. It specifies the service's input and output types, preconditions, and effects. The *ProcessModel* describes how the service works; each service is either an *AtomicProcess* that executes directly, or a *CompositeProcess*, i.e., a composition that combines subprocesses (Fig. 3-21). The *ServiceGrounding* contains the details of how an agent can access the service. This grounding specifies a communications protocol, the parameters to be used in the protocol, and the serialization techniques to be employed for communication. Such a rich description of services greatly supports automation of their discovery and composition.

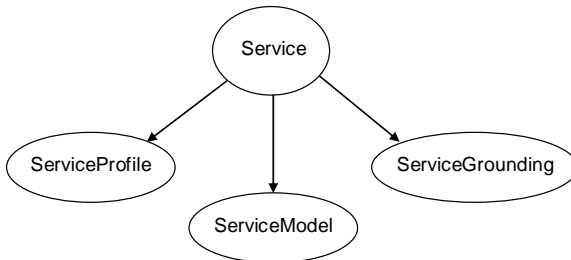


Fig. 3-20. Top level of the OWL-S ontology (after [Martin et al., 2004])

```

<owl:Class rdf:ID="CompositeProcess">
  <rdfs:subClassOf rdf:resource="#Process"/>
  <owl:disjointWith rdf:resource="#AtomicProcess"/>
  <owl:disjointWith rdf:resource="#SimpleProcess"/>
  <rdfs:comment>
    A CompositeProcess must have exactly 1 composedOf property.
  </rdfs:comment>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Process"/>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#composedOf"/>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
        1 </owl:cardinality>
      </owl:Restriction>
    </owl:intersectionOf>
  </owl:Class>
  
```

Fig. 3-21. An excerpt from the OWL-S ontology defined in [Martin et al., 2004]

OWL-S service descriptions are structured as OWL documents, so developers can build them using all of OWL's domain-modeling features, as well as concepts from other ontologies [Sirin et al., 2004]. Also, some aspects of the derivation of OWL-S descriptions directly from WSDL descriptions can be partially automated.

The tricky part in building Semantic Web service descriptions using OWL-S is combining different services from different providers – these services might assume different ontologies [Payne & Lassila, 2004]. Thus mapping of concepts between different ontologies and composition of new concepts from multiple ontologies is necessary. One way around this problem is to have agents and service requesters translate service descriptions into a familiar ontology to formulate valid requests. This translation can be done by employing a set of “bridging axioms”. Sirin et al. have proposed an alternative approach – generating service compositions that satisfy users' requirements through an interactive metaphor [Sirin et al., 2004]. This approach assumes knowledge-based indexing and retrieval of services by both agent brokers and humans, as well as automated reasoning about the services, but is essentially semiautomated, since the user is involved in the composition of the services as well. Nevertheless, the supporting tool that Sirin et al. have developed uses contextual information to locate semantically interoperable services that it can present to the user at each stage of the composition framework. Built on top of OWL-S, the tool enables semantic discovery and filtering to determine a meaningful set of candidate services on the basis of advertised Semantic Web service descriptions. The user's involvement here is reduced to a necessary minimum – it is the user who has the final word about selecting a particular service (for the next step in composition) after the automatic filtering of candidate services. The next step in automation might be creating an intelligent agent to do this selection on behalf of the user. However, owing to the huge variety of possible services, and of the respective domain ontologies and their representations, as well as variations in the composition, such an agent (or even a multiagent system) may not be easy to build.

3.6 Open Issues

While the Semantic Web is certainly gaining momentum, it is important to realize that some still unresolved problems and factors hold back its more rapid development. Critics of the Semantic Web frequently complain that

there is no “killer app” for the Semantic Web yet, which they interpret as a sign of a poorly grounded field and poorly envisioned future development.

In spite of the fast-growing representational and technological support, development of ontologies is still hard work. Tools such as Protégé are easy to use, but, nevertheless, someone always has to transfer human knowledge about a topic or subject to the ontological representation. Given that a domain ontology is supposed to represent the essential concepts in a domain and their relationships, the process always requires knowledge acquisition in which human experts are involved to a large extent. Building and representing ontologies in computers is not so much a technical matter as a matter of obtaining and organizing high-quality human knowledge about the domain of interest. True, partial automation of this process is possible by applying machine learning techniques, but such approaches are largely still under development.

Moreover, domain knowledge is seldom static – it evolves over time, much information that was once relevant may easily become obsolete, and new, important information may be discovered after the ontology has been built and represented on the Web explicitly. That raises the important issue of knowledge maintenance.

As already mentioned in Sect. 2.1, the term “ontology” is sometimes used to refer to a body of knowledge describing a domain, typically a domain of commonsense knowledge, using a representation vocabulary [Chandrasekaran et al., 1999]. However, commonsense ontologies are still not widely available, and efforts to create a standard upper ontology are still under way (see Sect. 2.4.2). The Austin-based company Cycorp has devoted many years to creating CYC, a huge ontology of commonsense knowledge that has been used in a wide range of applications [Lenat & Guha, 1990; Lenat, 1995]. The CYC ontology is extremely impressive, but Cycorp retains proprietary rights to the vast bulk of it and has released only a small part of it (called OpenCYC) to the public [Niles & Pease, 2001; SUO WG, 2005]. The contents of CYC have not been subject to extensive peer review, and thus using it as a standard would be problematic.

On the other hand, Cycorp’s philosophy is grounded in the fact that spontaneous development of domain-specific ontologies for the Semantic Web may lead to a Web rich in semantics, but poor in ontological consistency. Interrelating many heterogeneous ontologies in a useful way is anything but easy if they are richly specified (which takes a lot of effort in itself). However, if developers were to create a number of lightweight ontologies just to get started, they might want to use CYC to elaborate their ontologies, by using it to describe more precisely the intended meanings of the terms they have used. In this case, CYC could serve as a

global point of integration for many domain-specific ontologies through a large corpus of encoded commonsense knowledge.

Automation is a key issue for many aspects of the Semantic Web, including annotation/markup. In practice, much of the semantic markup of Web resources has been done more or less manually. This is not only time-consuming and error-prone – it is tedious. Moreover, any markup is good only as long as the resource remains unchanged; what if the resource is modified and the markup is no longer valid? Note that creating semantic markup will be one of the key factors for the success of the Semantic Web, but it is certainly not the ultimate goal. It is therefore necessary to enable easy annotation and automate the process as much as possible, as well as to achieve effortless updating of markup in the ever-changing environment of the Web. Automated annotation is a hot research topic, and there are several approaches.

Some also argue that the success of the Semantic Web will depend largely on the integration of Semantic Web technology with commercial software products. A good example of technology designed for this purpose is Briefing Associate [Tallis et al., 2002], discussed in Sect. 2.3. Its integration with MS PowerPoint, a tool used very widely for creating presentations, indicates a way to achieve the mass annotation of certain categories of documents without the authors having to care about it. Efforts are underway to enable such “semantic markup as a side effect” for different categories of Web resources, as well as to provide multiple annotations of the same resource to facilitate its reuse by multiple applications.

Semantic Web services represent an important step toward the full-blown vision of the Semantic Web, in terms of utilizing, managing, and creating semantic markup [Payne & Lassila, 2004]. Note that Semantic Web services nicely complement ontologies – services tackle behavioral issues of the Semantic Web (e.g., interactions between intelligent agents), whereas ontologies implement the Semantic Web’s original objective of creating representational and logical frameworks to allow increasing automation in processing Web-based information and in improving the interoperability of Web-based applications.

Still, as Preece and Decker carefully note [Preece & Decker, 2002], there is a trade-off between the functionality of Semantic Web services and the cost of developing the underlying markup and computational processes. The greater the functionality, the greater the cost. A more detailed study of the real needs of users may indicate a way to reduce this trade-off. This issue is closely related to another one – the trust that users will put in automated Semantic Web services. As Semantic Web services become more common, users will want to know about their quality before

they delegate their hands-on browsing to hands-off “black box” services. This creates a need for a set of objective and subjective metrics for the quality of Semantic Web services, such as how well the service has “understood” the user’s needs, how good it was in fulfilling those needs, and how accurate and complete the result was.

3.7 Quotations

The Semantic Web is a universal space for anything which can be expressed in classical logic. In the world of knowledge representation there are many different systems, and the following is an attempt to generalize. ... Exposing rules as classic logic facts strips the (pragmatically useful) hint information which controls the actual sequence of operation of a local inference engine. When the facts corresponding to all the rules of all the inference engines are put onto the Web, then the great thing is that all the knowledge is represented in the same space. The drawback is that there is no one inference engine which can answer arbitrary queries. But that is not a design goal of the Semantic Web. The goal is to unify everything which can be expressed in classical logic (including more mathematics when we get to it) without further constraint. We must be able to describe the world, and our hopes and needs and terms and conditions. A system which tries to constrain the expressive power cannot be universal. ... The choice of classical logic for the Semantic Web is not an arbitrary choice among equals. Classical logic is the only way that inference can scale across the Web. [Berners-Lee, 1997–2004]

Instead of trying to rebuild some aspects of a human brain, we are going to build a brain of and for humankind. [Fensel & Musen, 2001]

4. The Model Driven Architecture (MDA)

A relevant initiative from the software engineering community called Model Driven Development (MDD) is being developed in parallel with the Semantic Web [Mellor et al., 2003a]. The MDD approach to software development suggests that one should first develop a model of the system under study, which is then transformed into the real thing (i.e., an executable software entity). The most important research initiative in this area is the Model Driven Architecture (MDA), which is being developed under the umbrella of the Object Management Group (OMG). This chapter describes the basic concepts of this software engineering effort.

4.1 Models and Metamodels

Models play a major role in the MDA. The most general definition says that a model is a simplified view of reality [Selic, 2003], or, more formally, a model is a set of statements a system under study [Seidewitz, 2003]. In fact, one can say that a model is a clear set of formal elements that describes something being developed for a specific purpose and can be analyzed using various methods [Mellor et al., 2003a]. In addition to what is specified by the definition of a model, an engineering model must possess, to a sufficient degree, the following five key characteristics [Selic, 2003]:

- *Abstraction.* A model is always a reduced rendering of the system that it represents.
- *Understandability.* It is not sufficient just to abstract away detail; we must also present what remains in a form (e.g., a notation) that most directly appeals to our intuition.
- *Accuracy.* A model must provide a true-to-life representation of the modeled system's features of interest.
- *Predictiveness.* We should be able to use a model to correctly predict the modeled system's interesting but nonobvious properties, either

through experimentation (such as by executing a model on a computer) or through some type of formal analysis.

- *Inexpensiveness*. A model must be significantly cheaper to construct and analyze than the modeled system.

Metamodels are another key concept used in the MDA. A metamodel is a specification model for a class of systems under study, where each system under study in the class is itself a valid model expressed in a certain modeling language. A metamodel makes statements about what can be expressed (i.e., asserted) in the valid models of a certain modeling language. In fact, a metamodel is a model of a modeling language [Seidewitz, 2003]. The UML diagram shown in Fig. 4-1 represents the relations between a system under study and a model expressed in a specific modeling language. Since a metamodel itself is a model, it can be represented in a modeling language. In some modeling architectures, such as the MDA, there is a specialized modeling language for defining metamodels, and that language is defined in the metametamodeling layer of a specific modeling architecture. In the case of the MDA, this modeling language is called the Meta-Object Facility (MOF); this is described in more detail later in this chapter.

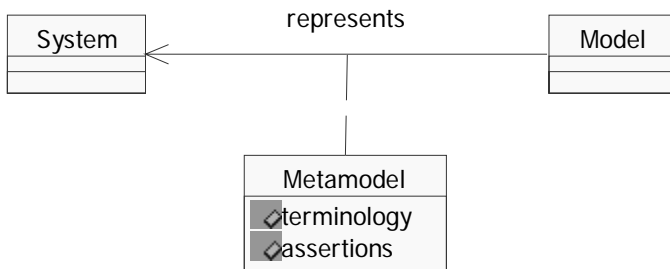


Fig. 4-1. The correspondence between a model and a system

4.2 Platform-Independent Models

Ideally, developers should see an information system as an *independent* world during its development, without any artificial constraints such as the operating system, hardware, network performance, or application incompatibility.

One possible solution that can enable such an approach is to execute every information system on the same hardware platform, operating system, architecture, etc. Even though this may not sound very realistic, one may notice that this approach is being already applied to the Windows operating system and other related software technologies by the company that produces them. However, even if all of the components were to be under the same development control, the field of information systems is so heterogeneous and prone to so many revisions that the constant need for improved versions of software would cause this approach to collapse.

Since there have already been many different platforms and too many conflicting requirements, it is very hard for developers to agree to one common solution. Accordingly, a more realistic scenario is to enable coexistence of the present systems by representing them by models and by transforming models into other models. The MDA indeed attempts to provide a solution to the problem of such an integration. Although the OMG's architectural framework is constantly being changed, the primary goals of achieving both interoperability and portability are still unchanged.

The MDA defines three viewpoints (levels of abstraction) for analyzing systems. Given a viewpoint, we can define a representation of a system under study, that is to say, a model of the system seen from that viewpoint. In fact, for each viewpoint there is a corresponding model, namely

- the computational-independent model (CIM);
- the platform-independent model (PIM);
- the platform-specific model (PSM).

The CIM does not show details of the structure of the system. In software engineering, it is well-known as a domain model specified by domain experts. This is very similar to the concept of an ontology.

The PIM is a computation-dependent model, but it does not consider the characteristics of specific computer platforms. In other words, the PIM is a model assumed to be executed on a technologically independent virtual machine.

The PSM finalizes the specification of a whole computer system. The main goal is to shift developers' focus from the PSM to both the PIM and the CIM. In this way, platform-specific details should be generated using various tools for automatic generation of those details (e.g., code). These tools should transform the PIM into the PSM (see Fig. 4-2). More details of these abstraction levels and their corresponding models can be found in [Frankel, 2003; Kleppe et al., 2003; Mellor et al., 2003b].

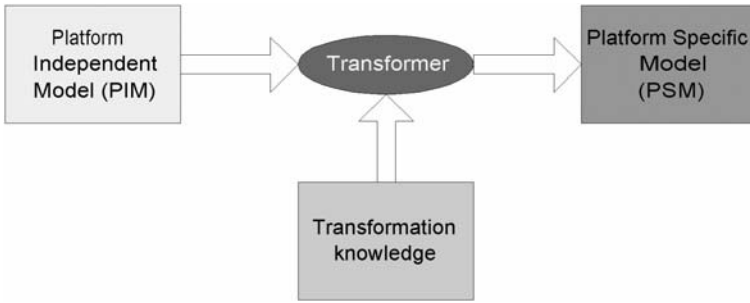


Fig. 4-2. Transforming a platform-independent model (PIM) into platform a specific-model (PSM) using transformation tools

4.3 Four-Layer Architecture

The MDA is based on a four-layer metamodeling architecture and a few complementary standards (see Fig. 4-3):

- the Meta-Object Facility (MOF);
- the Unified Modeling Language (UML);
- the XML Metadata Interchange (XMI).

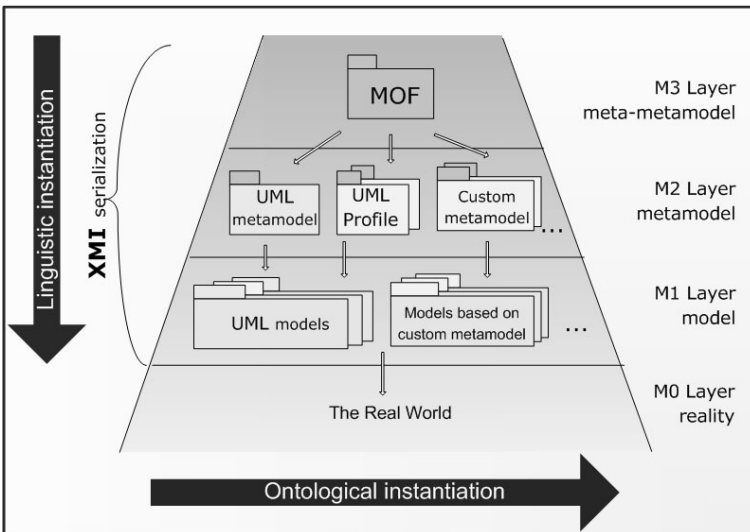


Fig. 4-3. The four-layer Model Driven Architecture and its orthogonal instance-of relations: linguistic and ontological

The topmost level of this architecture is the metametamodel, that is to say, the MOF. It is an abstract, self-defined language and framework for specifying, constructing, and managing technologically independent metamodels. It is a basis for defining any modeling language, such as UML or the MOF itself. The MOF also defines a backbone for the implementation of a metadata (i.e., model) repository described by metamodels [OMG MOF, 2002; OMG MOF2, 2003]. The rationale for having these four levels with one common metametamodel is to enable both the use and the generic managing of many models and metamodels, and to support their extensibility and integration.

All metamodels, standard and custom (user-defined), that are defined in the MOF are placed in the M2 MDA level. One of these metamodels is UML, which stands as a language for specifying, visualizing, and documenting software systems. The basic concepts of UML (e.g., Class and Association) can be extended in UML profiles in order to adapt UML to specific needs. Models of the real world, which are represented by concepts from a metamodel belonging to the M2 level, are in the M1 level of the MDA four-level architecture. In the M0 level, are things from the real world that are modeled in the M1 level. For example, the MOF Class concept (from the M3 level) can be used for defining the UML Class concept (M2), which, in turn, defines the Person concept (M1). The Person concept is an abstraction of the real thing *person*. We provide a more detailed discussion of modeling architectures and their mutual relations in the next chapter.

The bottommost layer is the instance layer (M0). There are two different approaches to describing this layer:

1. The instance layer contains instances of the concepts defined in the model layer (M1), for example, objects in programming languages.
2. The instance layer contains things from our reality, both concrete (e.g., Mark is an instance of the class Person, and Lassie is an instance of the class Dog) and abstract (e.g., UML classes – Dog, Person, etc.) [Atkinson & Kühne, 2003].

In this book, we advocate the second approach, but we need to give more detail about its impact on UML. In UML, both classes and objects are in the same layer (the model layer) in the MDA four-layer architecture. The MDA layers are called linguistic layers. On the other hand, concepts from the same linguistic layer can be in different ontological layers. Hence, UML classes and objects are in different ontological layers, but in the same linguistic layer.

In addition to the aforementioned standards, the MDA is based on XML, a standard that defines mappings of MDA-based metametamodels, metamodels, and models onto XML documents and XML schemas. Since XML is widely supported by many software tools, it empowers XMI to enable better exchange of metametamodels, models, and models.

4.4 The Meta-Object Facility

The Meta-Object Facility (MOF) [OMG MOF2, 2003] originated as an adaptation of the UML core, which had already gained popularity among software modelers, to the needs of the MDA. The MOF is, essentially, a minimal set of concepts which can be used to define other modeling languages. It is similar (but not identical) to the part of UML which is used in the modeling of structure. In the latest version (2.0), the concepts of the MOF and of the UML superstructure, are derived from the concepts of the UML infrastructure.

Essentially, there is an OMG standard called the UML infrastructure [OMG UML, 2003a], which contains basic concepts that are intended to be used in other metamodels. Figure 4-4 shows the dependency of some widely used metamodels based on the UML core package.

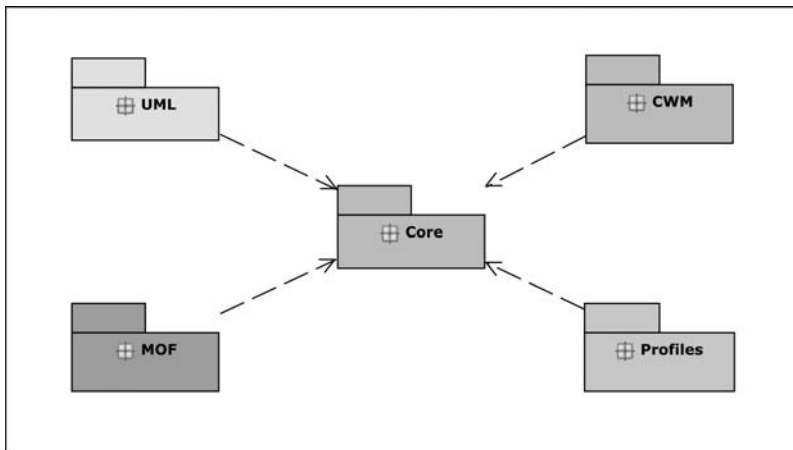


Fig. 4-4. The UML core package considered as a mutual kernel

The UML core package precisely defines the basic concepts that are frequently used in modeling. A notable difference with respect to the older version is that every concept in the new version is narrowly focused on

some small aspect. This enables these concepts to be easily combined in various metamodels, avoiding usage of those aspects that are not required.

In version 2.0 of the MOF standard, there are two metametamodels to choose from:

- Essential MOF (EMOF);
- Complete MOF (CMOF).

EMOF is a compromise that favors simplicity of implementation over expressiveness, while CMOF is more expressive, but more complicated and harder to implement. Figure 4-5 shows EMOF and CMOF and their dependencies.

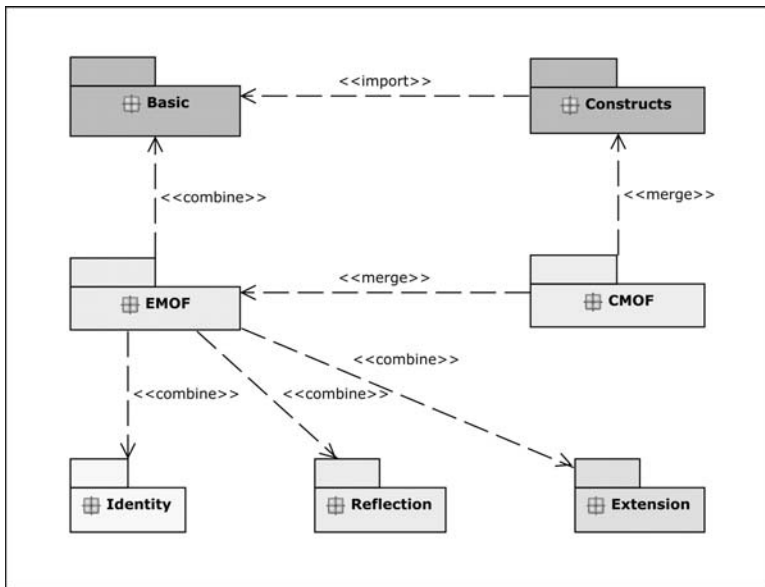


Fig. 4-5. EMOF and CMOF and their dependencies

From Fig. 4-5, we can see that EMOF is derived mostly from the Basic package of the UML infrastructure, whereas CMOF expands EMOF using concepts from the Constructs package (which is a part of the UML infrastructure).

Basically, the four main modeling concepts in the MOF are:

- Class – this models MOF metaobjects, concepts which are entities in metamodels (i.e. UML Class, Attribute and Association, ODM Class, Property, etc., and even MOF concepts such as Class or Property);

- Association – this models binary relationships (UML or MOF superclasses and types or ODM superproperties, for example);
- DataType – this models primitive types (String, Integer, etc.);
- Package – this modularizes other concepts; for example, it groups similar concepts.

Of course, there are many more such concepts, but these are the most important and most frequently used. For details, see the MOF specification [OMG MOF2, 2003]. Figure 4-6 should give a feeling of what some metamodels look like. It shows part of the MOF definition of a metamodel.

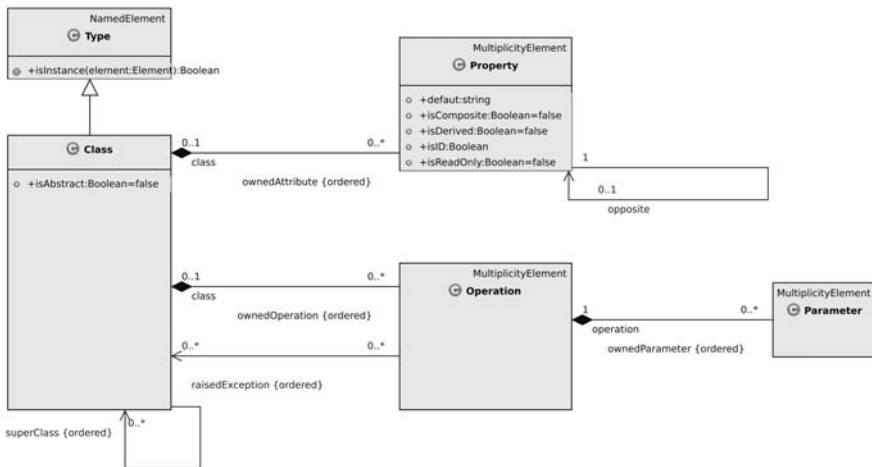


Fig. 4-6. The definition of part of a metamodel (in this case, EMOF is defined within itself)

Do not be confused by the fact that MOF Class is defined using MOF Class. This is like defining words in a dictionary: one word is defined by some other words, but all words are defined by words from the same dictionary. For example, the definition of “or” in a dictionary may include the following statement: “restating or clarifying the first item mentioned.” So, the definition of “or” includes the word that it defines! Regarding other metamodels defined in EMOF, there is a part of the UML infrastructure that defines the UML Class concept, and that diagram is very similar to the diagram shown in Fig. 4-6. In this book, you will find a large part of the definition of the Ontology Definition Metamodel – it is defined in EMOF.

4.5 Specific MDA Metamodels

In order to illustrate the use of the MOF language, this section describes three well-known MOF-defined metamodels defined in the OMG specifications.

4.5.1 Unified Modeling Language

The Unified Modeling Language (UML) is a language for specifying, visualizing, and documenting software systems, as well as for modeling business and other nonsoftware systems. UML is a result of the best practice in engineering modeling and has been successfully proven in modeling many big and complex systems.

As we have already mentioned, the UML core is the same as that of the MOF. Accordingly, we shall not discuss its elements; a comprehensive overview can be found in the UML language specification [OMG UML, 2003a]. UML is often identified as a graphical notation, which was true for its initial versions. Recently, UML has been recognized more as a language independent of any graphical notation rather than as a graphical notation itself.

However, UML is also very important as a language for the graphical representation of models of software systems. The point of view from which a problem under study determines crucially which elements of that problem will be stressed in the final model. UML has features for emphasizing specific views of a model by using graphical representations of models, namely UML diagrams. In this way, we can abstract models; we may otherwise not be able to analyze and solve complex systems. UML defines the following diagrams for various views of models:

- Use Case Diagram
- Class Diagram
- Behavior Diagrams
 - Statechart Diagram
 - Activity Diagram
- Interaction Diagrams:
 - Sequence Diagram
 - Collaboration Diagram
- Implementation Diagrams:
 - Component Diagram
 - Deployment Diagram

These diagrams provide developers with various perspectives of the system under study or development. A model which captures a whole system and is graphically represented by diagrams just integrates all these perspectives into one common entity, comprising the union of all modeled details of that system.

Probably, the best overview of the use of the UML language is given in [Fowler, 2003]; more details about the language itself can be found in [Booch et al., 1998; Rumbaugh et al., 2004].

4.5.2 Common Warehouse Metamodel (CWM)

The great complexity and diversity of the fields described by metamodels means that many different metamodels are constantly evolving. Even when those metamodels are based on the same metametamodel, their interoperation requires a number of bridges (e.g. transformations). This problem is emphasized in the field of data warehouse, where one aims to manipulate and integrate data based on a large number of different metamodels.

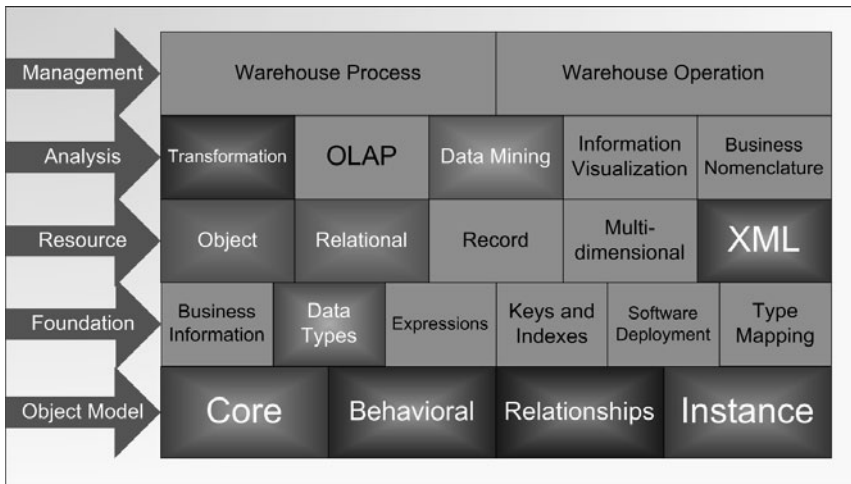


Fig. 4-7. The structure of Common Warehouse Metamodel (CWM)

Common Warehouse Metamodel (CWM) is an OMG open industrial standard for integrating tools for data warehouse and business analysis. It is based on using common metadata [Poole, 2002]. CWM is an example of an approach to model-based metadata integration. It is not one monolithic metamodel, but actually comprises many different but related models.

Each model is an information subdomain in a business chain. Figure 4-7 shows a block diagram of the organization of CWM, namely the metamodels contained in CWM.

CWM comprises several metamodels organized into the following layers: Object Model, Foundation, Resource, Analysis, and Management. Metamodels located in the higher layers rely on metamodels from the lower layers. The basic layer (Object Model) is based on UML, which is actually a superset of it. In Fig. 4-7, we have emphasized metamodels that are more relevant for designing metamodels of intelligent information systems, even though it is not very sensible to exclude any metamodel a priori. For example, the Data Mining metamodel belongs to the group of metamodels of intelligent information systems.

CWM may have a big influence on the integration of metamodels for intelligent information systems. It can be used as either an extensible basis or a template for building new metamodels.

4.5.3 Ontology Definition Metamodel

The MDA and its four-layer architecture provide a solid basis for defining the metamodels of any modeling language, and thus a language for modeling ontologies based on the MOF. Such a language can employ MDA tool support for modeling, model management, and interoperability with other MOF-based metamodels. The current software tools do not implement many of the fundamental MDA concepts. However, we can expect that most of these tools, which are presently oriented towards UML and the modeling layer (M1), will be improved and equipped with MDA support in the coming years.

Currently, there is an RFP (Request for Proposal) within the OMG that is aimed at defining a suitable language for modeling Semantic Web ontology languages in the context of the MDA [OMG ODM RFP, 2003]. The proposal is a result of an extensive previous research in the fields of the MDA and ontologies [Baclawski et al., 2002a; Baclawski et al., 2002b; Brockmans et al., 2004; Cranefield, 2001a; Djurić et al., 2005a; Falkovych et al., 2003; Kendall et al., 2002]. This proposal prescribes the most important components that should be included in the final OMG specification, namely:

- the Ontology Definition Metamodel (ODM);
- the Ontology UML Profile – a UML profile that supports UML notation for ontology definition;

- two-way mappings between OWL and ODM, between ODM and other metamodels, between ODM and the Ontology UML Profile, and between the Ontology UML Profile and other UML profiles [Gašević et al., 2004a].

At the time of writing this book, ODM and the Ontology UML Profile are still in the early stages of their specification. Initially, there were four submissions to the OMG ODM RFP, which have been joined into a common specification proposal. However, the final specification has not yet been adopted. One of the main reasons is that the UML 2.0 and MOF 2.0 specifications have not been adopted at the time of writing (late 2005), even though they have been completely defined. Since the main purpose of this book is to reflect the use of MDA technologies for developing ontologies, this OMG initiative is described in detail later in the book.

4.6 UML Profiles

The UML profile is a concept used for adapting the basic UML constructs to a specific purpose. Essentially, this means introducing new kinds of modeling elements by extending the basic ones, and adding them to the modeler's repertoire of tools. In addition, free-form information can be attached to the new modeling elements.

The basic UML constructs (model elements) can be customized and extended with new semantics by using four UML extension mechanisms defined in the UML specification [OMG UML, 2003b]: stereotypes, tag definitions, tagged values, and constraints.

Stereotypes enable one to define virtual subclasses of UML metaclasses, by assigning them additional semantics. For example, we may want to define the «OntClass» stereotype (see Fig. 4-8) by extending the UML Class «metaclass» to denote modeling element used to represent ontological classes (and not other kinds of concepts).

Tag definitions can be attached to model elements. They allow one to introduce new kinds of properties that the model elements may have, and are analogous to meta-attribute definitions. Each tag definition specifies the actual values of the properties of individual model elements; these values are called tagged values. Tag definitions can be attached to a stereotype to define its virtual meta-attributes. For example, the «OntClass» stereotype in Fig. 4-8 has a tag definition specifying four tagged values (for enumeration, intersection, etc.).

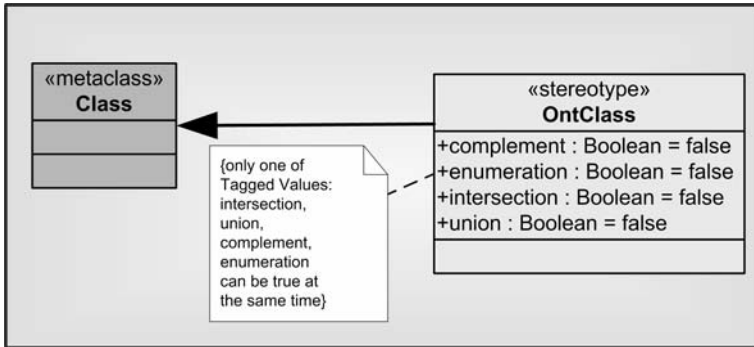


Fig. 4-8. A definition of a new stereotype

Constraints make it possible additionally to refine the semantics of the modeling element that they are attached to. They can be attached to a stereotype using OCL (Object Constraint Language) or the English language (i.e., spoken language) in order to precisely define the stereotype's semantics (see the example in Fig. 4-8). More details of UML extension mechanisms can be found in [OMG UML, 2003b; Rumbaugh et al., 1998].

A coherent set of extensions of the basic UML model elements, defined for specific purposes or for a specific modeling domain, constitutes a UML profile. A UML profile definition in the context of the MDA four-layer metamodeling architecture means extending UML at the level of the metamodel layer (M2). One can understand such extensions as a new language, but can also understand UML as a family of languages [Duddy, 2002]. Each of these languages uses the UML notation, with the four UML extension mechanisms. The recent UML specifications enable graphical notation to be used for specifying stereotypes and tagged definitions [Kobryn, 2001]. Thus, all of the stereotypes and tagged values used in this chapter have been defined in this way.

4.6.1 Examples of UML Profiles

Many important UML profiles have now been developed. Some UML profiles have been adopted by the OMG, such as the UML Profile for CORBA [OMG CORBA, 2002] and the UML Profile for Enterprise Application Integration [OMG EAI, 2004]. Besides these formal specifications, several well-known UML profiles have been widely accepted by software engineers. One of the most popular ones is the UML profile for building Web applications developed by Jim Conallen [Conallen, 2002]. Recognizing the lack of UML primitives for modeling

Web applications (e.g., links, Web forms, and Web pages), Conallen extended some UML model elements (e.g., Class and Attribute) with primitives relevant to Web applications (e.g., server page, client page, and form). Figure 4-9 shows a UML diagram of a Web application modeled using Conallen's UML profile. The diagram contains UML representations of server pages (e.g., spUnbounded), a form, and buttons on the form (e.g., Simulate). This example shows the feature of UML whereby one can develop a specific icon for each primitive developed in a UML profile. In Fig. 4-9 we have used the graphical features of the Rational Rose UML tool for modeling Web applications using this UML profile. Of course, the use and definition of graphical icons is optional. In fact, the same semantic meaning of an extended UML model element (e.g., Class) would be obtained if we used a graphical representation of standard model elements together with the standard way of representing stereotypes (e.g., the Simulate *button* in the "form" class).

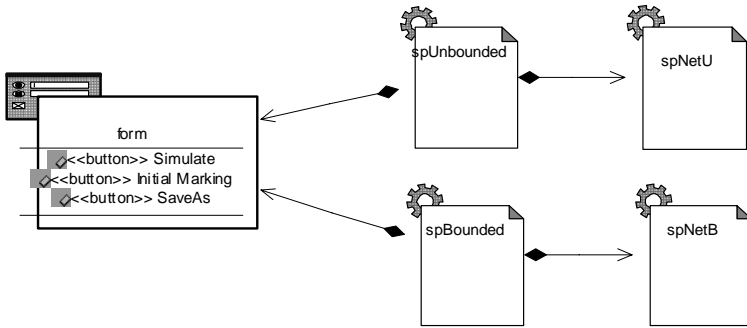


Fig. 4-9. UML diagram of a Web application modeled using a UML profile for modeling Web applications

Another important UML profile is that for modeling XML Schema developed by David Carlson [Carlson, 2001]. Understanding the needs of developers of business applications to use a rich set of XML Schema features, Carlson found that the resulting schemas were difficult to share with wider audiences of users and business partners. In fact, other forms of representation and presentation of models are more effective than XML Schema when specifying new vocabularies or sharing definitions with users. Carlson proposed the use of UML as a standard for system specification and design. Figure 4-10 shows the Musician ontology represented in Carlson's UML profile for modeling XML Schema. It is important to note that each class and attribute has a corresponding stereotype (i.e., XSDcomplexType and XSDattribute). In this way, one can map this model onto an XML Schema definition using tools (e.g., an

XSLT that transforms an XMI representation of a UML model into an XML Schema definition). Besides the stereotypes shown in Fig. 4-10, the UML Profile contains all other primitives required for defining XML Schemas.

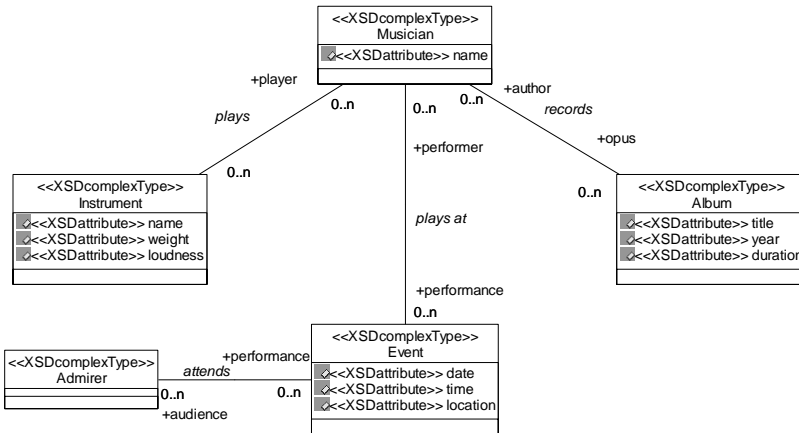


Fig. 4-10. A UML diagram of the Musician ontology, represented with the UML profile for modeling XML Schema

Note, finally, that there are many other important UML profiles apart from the ones that we have illustrated in this book. Some examples are the UML profile for database design [Naiburg & Maksimchuk, 2001] and the UML profile for framework architectures. Taking into account the fact that the main goal of this book is to illustrate the use of MDA standards for developing ontologies, we shall explain the Ontology UML Profile in detail.

4.7 An XML for Sharing MDA Artifacts

XML Metadata Interchange (XMI) is an XML-based standard for sharing MDA metadata [OMG XMI, 2002]. Although this standard sounds as if it ought to be very well-known, most practitioners are confused by this term. We shall try to explain XMI using Fig. 4-11. The confusion comes from the presence of several different XML schemas, and all of them are called XMI. We usually encounter two kinds of XMI documents, or, more precisely, the two XML schemas defining XMI:

- the XML schema for MOF metamodels;

- the XML schema for UML models.

The first of these defines the syntax for sharing both MOF-based metamodels and the definition of the MOF itself. So, we use one schema in two different MDA layers, M3 and M2, for sharing both metametamodels and metamodels. For instance, the MOF is defined by itself, so we use the MOF XML schema to describe the XMI document comprising the MOF specification, and this document is also a part of the MOF standard.

Similarly, there is the document that describes the XMI standard containing the UML metamodel. However, UML is a modeling language that developers use for various different models. It is obvious that here is a need for an XML schema for exchanging UML models. In fact, there the standardized one called the UML XMI Schema. UML tools such as IBM/Rational Rose, Poseidon for UML, and Together support this schema, but some researchers report that we always lose some information when sharing UML models between two UML tools [Uhl & Ambler, 2003].

Furthermore, it is well-known to developers that we never employ the UML XML Schema as an XML language in domain applications. We always define a domain XML language, i.e., a domain XML schema. Accordingly, there is a set of rules for mapping UML models into XML schemas [Grose et al., 2002]. Thus, one can generate an XML schema for any UML model, while instances of models (i.e., objects) are shared in accordance with those schemas. In view of the presumption that both UML objects and UML classes are in the same MDA layer (the M1 layer), we regard the generated XML schemas and their instance XML documents as being placed in the M1 layer.

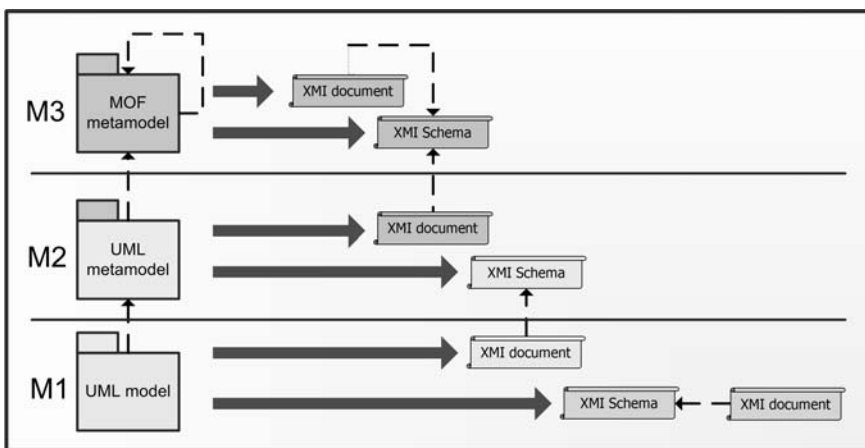


Fig. 4-11. Mapping MDA metametamodels, metamodels, and models to XMI

Since we have a set of rules for generating XML schemas from UML models, we can apply the same principle to the upper MDA layers (M2 and M3), and so we can produce an XML schema for each MOF-based metamodel. Using that principle, we have generated XML schemas for various metamodels (e.g., ODM [Djurić et al., 2005a]). For example, the developers of Protégé have produced an XML schema for a MOF-compatible metamodel of the Protégé ontology editor [Protégé, 2005].

Summarizing the story about XMI, we can say that XMI is:

- a set of rules for serialization of object;
- a set of rules for generation of schema.

Accordingly, the OMG has specified several documents related to XMI:

- the XML schema for MOF metamodels;
- the XMI document for the MOF metamodel;
- the XMI document for the UML metamodel; and
- the XMI schema for UML models.

In order to illustrate the aforementioned XMI documents and schemas, we show in Fig. 4-12 an excerpt from an XMI document representing the Musician UML class defined in the previous chapters. The document shown has been written in accordance with the standardized XMI schema for UML models. This document is at the bottommost level (M1) in Fig. 4-11, and is an instance of the XML schema defined at level M2 in Fig. 4-11.

```
<UML:Class xmi.id="Im19247250m106c9d71d66m24d0" name="Musician"
visibility="public" isSpecification="false" isRoot="true" isLeaf="true" isAbstract="false" isActive="false">
  <UML:Classifier.feature>
    <UML:Attribute xmi.id="Im19247250m106c9d71d66m24d1"
name="name" visibility="private" isSpecification="false"
ownerScope="instance" changeability="changeable"
targetScope="instance">
      <UML:StructuralFeature.multiplicity>
        <UML:Multiplicity xmi.id="Im19247250m106c9d71d66m250c">
          <UML:Multiplicity.range>
            <UML:MultiplicityRange xmi.id="Im19247250m106c9d71d66m250d" lower="1" upper="1"/>
          </UML:Multiplicity.range>
        </UML:Multiplicity>
      </UML:StructuralFeature.multiplicity>
    <!-- -->
    <UML:Attribute.initialValue>
      <UML:Expression xmi.id="Im19247250m106c9d71d66m24d2" language="java" body=""/>
    </UML:Attribute.initialValue>
    <UML:StructuralFeature.type>
      <UML:DataType xmi.idref="Im19247250m106c9d71d66m24d3"/>
    </UML:StructuralFeature.type>
  </UML:Attribute>
</UML:Classifier.feature>
</UML:Class>
```

Fig. 4-12. An excerpt from a UML XMI document representing the Musician UML class

4.8 The Need for Modeling Spaces

After reading the previous section, readers can be expected to be confused by the use of many XML documents to represent models from different MDA levels. The main reason for doing that is that we are trying to represent one modeling paradigm (i.e., the MDA) in terms of another one (i.e., XML). One of the critical issues is that a metamodel can be represented using an XML document, but at the same time we generate an XML schema based on the same metamodel. Furthermore, we use both XML documents and MDA models to represent things taken from reality, but at the same time we represent MDA models using XML. So, are MDA models things from the real world? An additional question is whether instantiation relations between XML schemas and XML documents are the same as instantiation relations between MDA metamodels and models. Similar problems can be expected in understanding the differences and similarities between MDA models and ontologies. Since an understanding of the relations between these two modeling spaces is very important for the purpose of this book, the next chapter defines the concept of modeling spaces [Djurić et al., 2006b].

5. Modeling Spaces

It is well known from software project management practice that developers' familiarity with the principles of several different modeling approaches can be beneficial for the development team. Changing the modeling and design aspects from time to time during the system development, e.g. from OO to relational, from relational to functional, and the like, often helps prevent schedule overruns [Zahniser, 1993].

In practice, developers commonly use one or more integrated, complex tools that involve multiple modeling approaches. Remember your last round-trip with UML [OMG UML, 2003b], Java/C# code, specialized technologies such as J2EE, database schemas, XML files etc.? You do not always need to think about the relationships between these; sometimes it is a matter of choosing the right option from the menu of your favorite integrated development environment. However, you might find that, even when the tool you use supports changes from one modeling aspect to another, getting the big picture is a rather complex task. Things get even more complex in situations when the support provided by tools is not straightforward and you need a lot of effort to figure out how to transform one model to another.

Using a number of modeling approaches greatly increases the developers' ability to better describe the problem using the right approach, but is also more complex to comprehend in its entirety. Moreover, it can be difficult for software developers to adopt the practice of using a wide spectrum of modeling approaches because such approaches may seem too different and too unrelated to each other. In order to save time and effort when working under tight development schedules, practitioners tend to focus on individual aspects of modeling [Seidewitz, 2003]. For all these reasons, a common view of various modeling approaches is highly desirable.

We have defined a formal encompassing framework for studying many modeling problems in a more comprehensive way. We call this framework “modeling spaces”. Its development was motivated by a desire to provide a better understanding of the vast diversity of things that can be modeled, and a clear rationale for using and combining different software modeling technologies. It has direct implications for software engineering processes

and activities, since it allows systematic change of modeling and design views throughout the system development cycle.

5.1 Modeling the Real World

If we search a dictionary for the word “model”, we find several definitions – most of them just referring to the special cases of models most often used; for instance: a replica of an item, a person wearing clothes at a fashion show, or a drawing of a building. Few of these definitions are generally applicable. Fortunately, there is a simple and general definition – a model is a simplified abstraction of reality [Hagget & Chorley, 1967]. Using that definition, we can explain why these special cases are models. A person wearing clothes at a fashion show does not represent herself/himself, but the appearance of any person wearing those clothes; thus that is a model. A drawing of a building is not just a sheet of paper with lines; it is also a simplified abstraction of a building containing only data that is important in a given context.

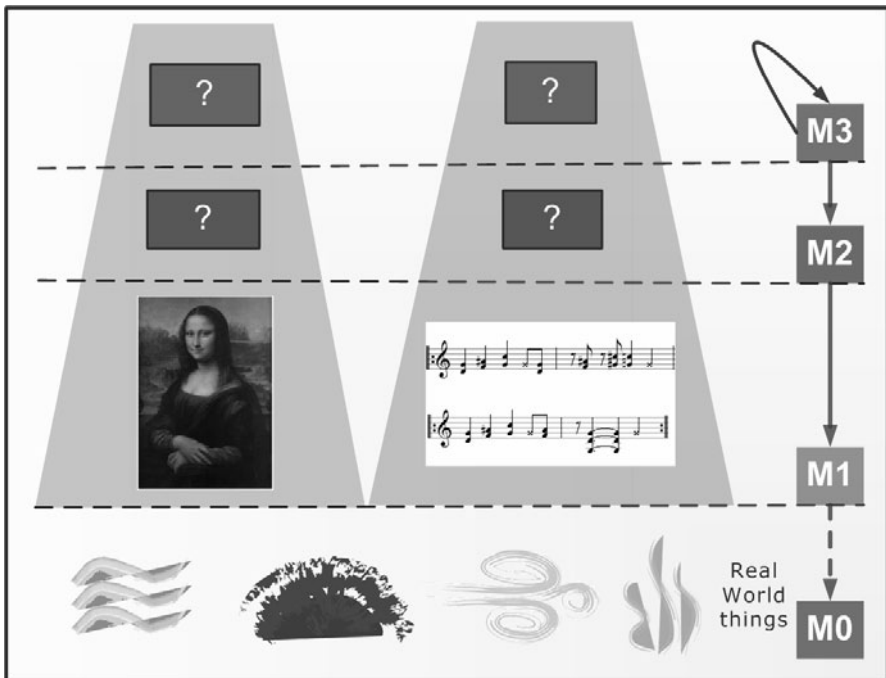


Fig. 5-1. A few common models put into a MDA-inspired layered architecture

Figure 5-1 shows some common examples of models put into a layered architecture conceptually inspired by the MDA— a famous painting (the Mona Lisa, also known as La Gioconda, painted by Leonardo da Vinci in the 16th century), and part of a music score (of the song “Smoke on the Water”). A noble Renaissance woman and a rock song are things from the real-world, in the M0 layer. A painting and a sheet of music are obviously abstractions of real world things. Therefore, they are models and can be put in the M1 (model) layer. Metamodels are used to define these models. In the case of a music score, which is a written interpretation of a song, the metamodel (M2) is a set of concepts – stave, note, etc. – and rules that define what each note means and how we can arrange them on a stave. In this context, the meta-metamodel (M3) includes self-defined concepts that also define a stave, note etc. Although this architecture is imprecise and definitely not perfect from the perspective of music theory, at least it captures a formal interpretation of music.

Things get harder in the case of the painting. Is it possible to specify a metamodel that can define such a complex and ambiguous model as a masterpiece painting? A simplistic view based on physical appearance only might lead to a definition of this metamodel in terms of concepts such as line, circle and color. The meta-metamodel would then be a set of concepts used to define line, circle, color and their meanings. However, this painting, like any other work of art, is much more than just lines and colors. It has much to do with human psychology and experience and can be interpreted in many ways. It is much more difficult, if not impossible, to define a formal metamodel or meta-metamodel in this case. We may anticipate that they exist, but they will be extremely complex and implicit.

Another important issue is that, although the Mona Lisa and written notes are models, they are also things in the real world. We can hold them in our hands (if the guards let us do this, in the case of the Mona Lisa) and they can be items entered in an information system that stores information about art.

5.2 The Real World, Models, and Metamodels

The previous analysis leads to two important conclusions. First, something can be taken as a model if it is an abstraction of things in the real world, but is simultaneously a thing in the real world. Whether we take it as a model or as a real-world thing depends on the context, i.e. on our point of view. Second, models can be defined using metamodeling concepts formally or implicitly. Since implicit metamodels, as in the case of works

of art, cannot be precisely defined using formalisms, we shall analyze only formal models in the rest of this discussion. Nevertheless, many of the conclusions can also be applied to implicit metamodels.

Figure 5-2B shows a general modeling architecture that was inspired by the MDA, and is in fact a generalization of it. In such a modeling architecture, the M0 layer is the real world as in [Atkinson & Kühne, 2003; Bézivin, 2004]. It includes all possible things that we try to represent using models (the M1 layer). This representation may be more or less abstract and simplified, depending on how rich our models are. Models are defined using concepts defined in metamodels (M2), and so each metamodel determines how expressive its models can be. The metamodels are also defined using concepts. These concepts belong to meta-metamodels (M3). We could continue this layering to infinity, but it makes much more sense to stop at some point. The topmost layer contains the super-metamodel (Mn), which is metacircular (defined by its own concepts). Between Mn and M1, there are n-2 layers. This architecture is said to be n-layered, where n is the actual number of layers above M0.

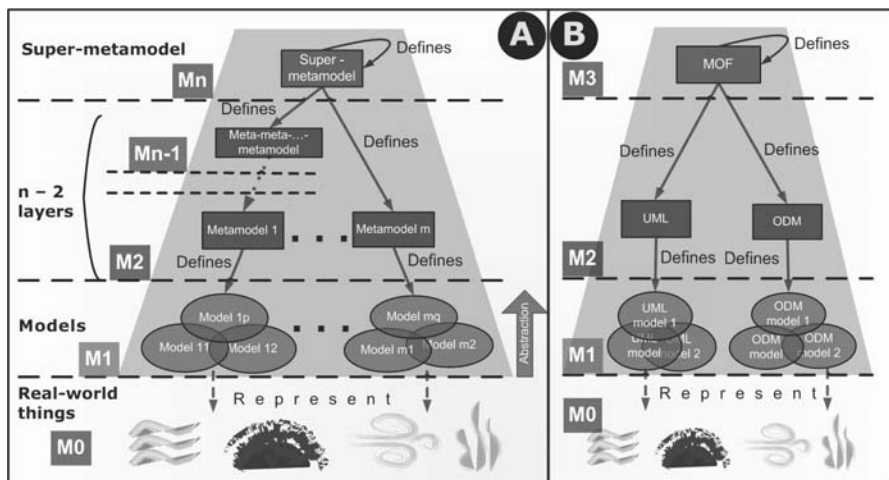


Fig. 5-2. (A) General four-layer modeling architecture (B) MDA

We use the term “represents” to denote that a model takes the place of a real-world thing, acting on its behalf in some specific context. Models’ concepts “conformTo” the metaconcepts that define them, in the sense that metaconcepts determine their nature, specify their precise meaning and form, and identify essential qualities. These terms can be defined more precisely in specific contexts and modeling systems. In the context of art, the Mona Lisa represents the ideal of beauty. In a different way, a

specification of an information system represents the actual programs in some other context. In yet another way, these programs represent the real business problem in a third context. Moreover, the abstraction is not absolute, but context-dependent; a real-world thing and its model could switch places when the context is changed.

The most widely known example of an n-layered architecture today is the Model Driven Architecture (MDA), based on three layers (Fig. 5-2B). The MDA's meta-metamodel, the Meta-Object Facility (MOF), is the super-metamodel (because M3 is the topmost layer here). The MOF standard defines an abstract language and a framework for specifying, constructing and managing technology-neutral metamodels. In essence, the MOF is an object-oriented metamodeling language extracted from the UML core. Well-known examples of MOF-defined metamodels include UML and CWM (Common Warehouse Metamodel). In the MDA, the XML Metadata Interchange standard (XMI) is used to define a mapping from MOF-defined metamodels to XML documents and schemas, thus enabling sharing of meta-metamodels, metamodels and models.

We shall not ask the question "How many layers are optimal?" here. In the original two-layered UML-based architecture, UML was the super-metamodel and the only metamodel. This solution was not able to support other metamodels, so another layer based on the MOF was added to support different kinds of models. We shall show later that most of today's popular modeling architectures can be considered as three-layered, and thus the following discussion will use three layers for the sake of simplicity.

5.3 The Essentials of Modeling Spaces

A modeling space (MS) is a modeling architecture based on a particular super-metamodel. Models in the M1 layer represent the real world in the context of a separate universe, organized in a hierarchy. Every layer above M0 in this hierarchy conforms to the higher layer, finally reaching the top layer containing the self-defined super-metamodel. If the super-metamodel was defined by some other concepts, it would not be a super-metamodel, it would exist at some lower layer in some other modeling space.

Figure 5-3 shows a few examples of well-known modeling spaces. The most straightforward example from this picture is the MOF MS. It is defined by the MOF meta-metamodel (MOF is the super-metamodel here), which in turn is defined by itself. It defines various metamodels, for instance UML [OMG UML, 2003b] or ODM (Ontology Definition

Metamodel - a metamodel for ontology modeling in MDA) [Djurić et al., 2005a], that are used to describe models that represent things in the real world. The same reality is described in the context of other modeling spaces, such as EBNF and RDFS spaces. Resource Description Framework Schema (RDFS) is a language that defines classes of defined resources and their properties in a machine-processable way. Many software engineers would associate terms such as “model” and “modeling” exclusively with the UML “aristocracy”, considering EBNF-based models (Java, C#, C++ code) as more technical, flattened artifacts and “ignoble citizens”. However, Java code (and also C++ and other code) is a model, since it represents an abstraction of reality. The same applies to XML code, databases, books, etc – they are all models, but modeled in terms of different modeling spaces, determined by different super-metamodels.

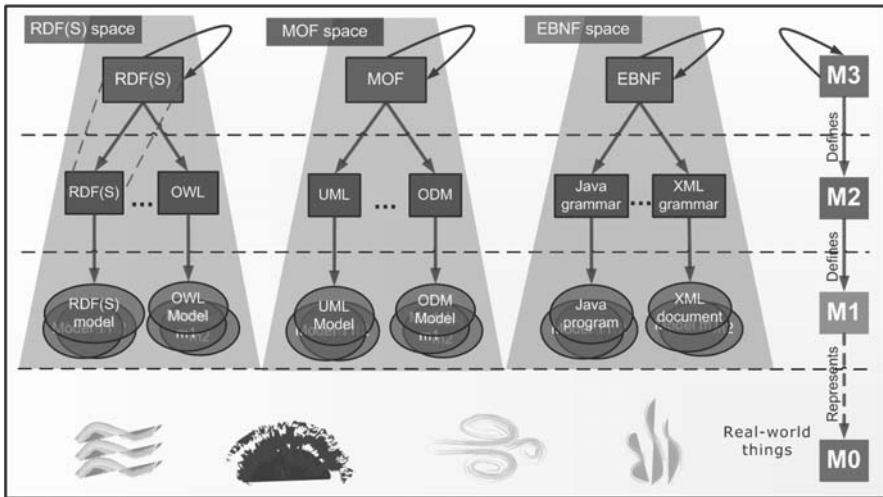


Fig. 5-3. The RDFS, MOF and EBNF modeling spaces

If we model the real world in a particular modeling space, we shall use certain models. If we model the same reality in another modeling space, we shall describe it with different kinds of models, highlighting other characteristics when abstracting from reality. The models in the first modeling space will be a part of reality that we can model using the models in the second modeling space.

Figure 5-4 clarifies this duality with an example of the same thing being simultaneously a model and a real-world thing. Along the vertical axis, the world is modeled in the MOF modeling space. Along the horizontal axis is the EBNF space hierarchy, which is a real-world thing in the MOF space. An interesting observation here is that any modeling space, such as the

EBNF space or even the MOF space itself, is a part of the real world from the MOF-space point of view. In general, the way we model a business system or other “real” domain is very much the same as the way we model meta-metamodels, metamodels, or models from another modeling space. Of course, these models involve a certain level of abstraction, so there is a possibility of losing some information.

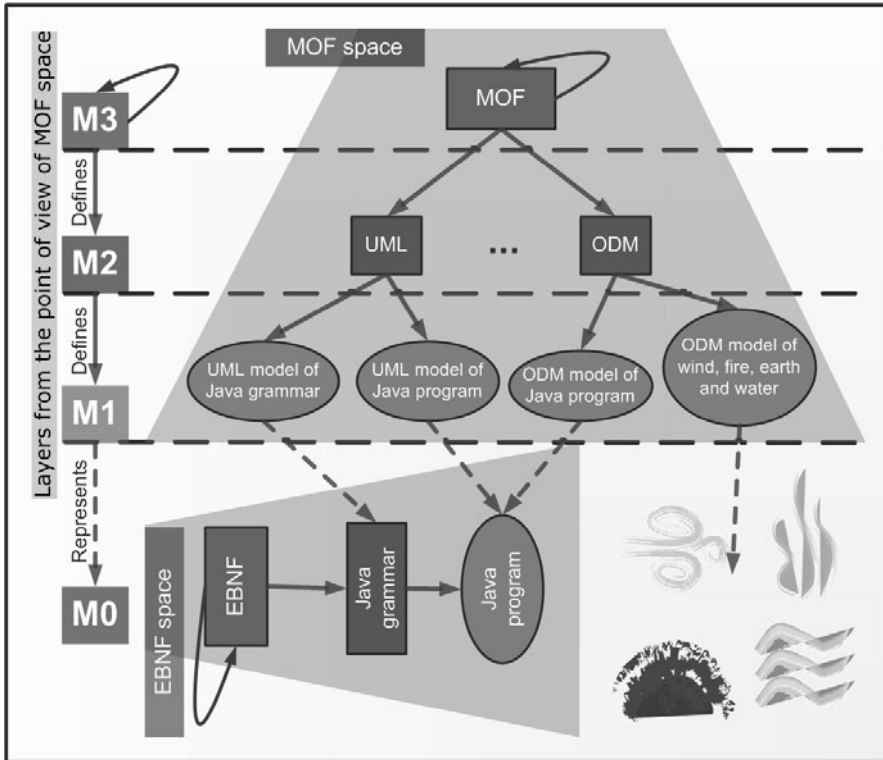


Fig. 5-4. The MOF MS sees EBNF MS as a set of things from the real world

For many software engineers, this duality is complicated to understand at first. Try a couple of analogies. Ghosts do not really exist (well, we hope so!), but they are things from the real world. Some people believe that there is life outside of the Solar system, and some claim it is pure fiction, but life outside the Solar system is a thing from the real world. Otherwise, we would not be able to model it using movies, literature, video games, etc. The fact that the M1-M3 layers are fiction and above the M0 layer does not mean that meta-metamodels, metamodels and models are things outside of reality. Everything is in the real world; we just use a convention to put some things into layers, depending on the context.

5.4 Modeling Spaces Illuminated

Modeling spaces can be defined in a more or less abstract manner. Some modeling spaces are focused on conceptual (abstract or semantic) things, such as models, ontologies and mathematical logics. These spaces are not aimed at in techniques for representation or for sharing their abstractions. We call such spaces conceptual modeling spaces. However, we must have some techniques to materialize (or serialize) those modeling spaces. We can do this using concrete modeling spaces, which are equipped with notation. Examples of such materializations are syntax and databases.

Take the MOF space as an example of a conceptual modeling space. The basic concepts of the MOF meta-metamodel – such as Class, Association, Attribute, Package and the relations among them – are expressed using those concepts themselves. We can draw them using UML diagrams, but a group of boxes and lines is not an MOF model – it is a drawing of an MOF model. We can serialize them into XMI, enabling computers to process them and programs to share them – but then we leave the MOF modeling space and enter the EBNF modeling space. One can argue that these drawings, i.e. UML diagrams, or models serialized into XMI, are inside the MOF modeling space because they represent MOF concepts. Indeed, they do represent concepts from the MOF space. Simultaneously, they represent other things from the real world. It means that the MOF concepts are modeled in another modeling space.

EBNF is an excellent example of a concrete modeling space. Theoretically well founded, it arguably has some “semantics” primarily for type checking, and has a syntax that is formally specified using a grammar. However, it in fact lacks semantics; when we parse the expression name = “Mona Lisa”, we obtain a syntax tree that does not know that it is dealing with the name of a painting. We always need some external interpretation of what its abstract syntax means. Actually, this interpretation is given in corresponding models from other technical spaces that have been serialized into the BNF form.

Being able to represent only bare syntax, concrete modeling spaces need some means to express the semantics, i.e. the meaning of the data they carry. Conceptual modeling spaces, on the other hand, are able to represent semantics, but need a means to represent their information physically. It is obvious that they should complement each other’s representation abilities to create models that have both a semantics and a syntax. One of the most interesting examples of this symbiosis of various modeling spaces can be found in the OMG Model Driven Architecture.

We can find similar “modeling patterns” in spoken languages, where people of different nationalities use different languages to express the same meaning. Regardless of the fact that they have the same things in mind (i.e. the same semantics), they need some rules to share the meanings. Spoken languages have their own syntax for sharing semantics. Just as we can model the same semantics in different modeling spaces, we can talk about the same thing using different spoken languages. Similarly, definitions of mathematical languages (i.e. logics) include two parts: syntax and semantics.

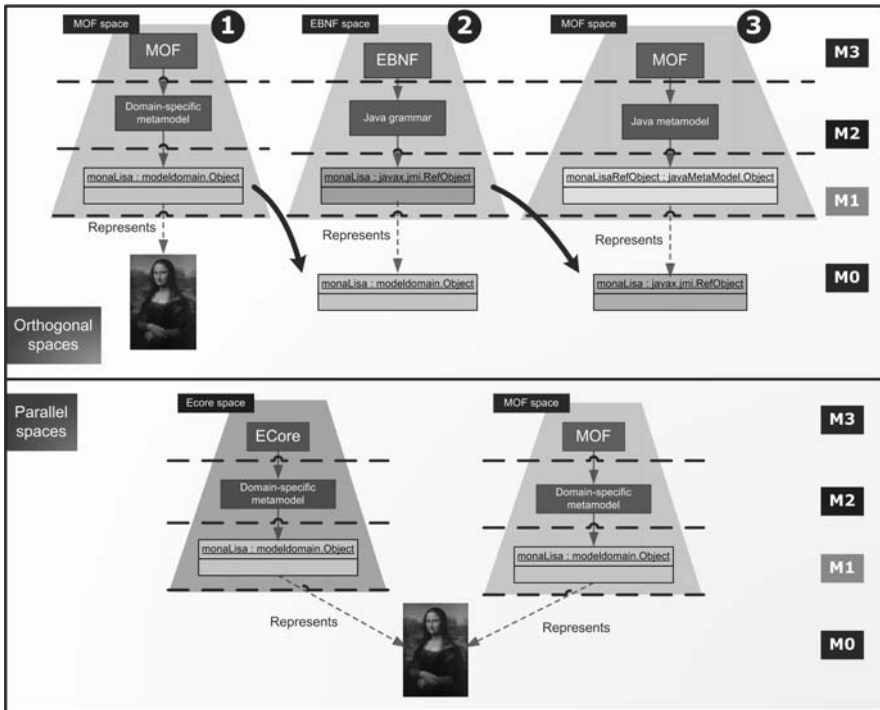


Fig. 5-5. Orthogonal and parallel spaces

There are two types of scenarios in which more than one modeling space is used, both shown in Fig. 5-5:

- **Parallel spaces** – one modeling space models the same set of real-world things as another modeling space, but in another way. In this case, the relation between these modeling spaces is oriented towards pure transformation, bridging from one space to another. Examples of such parallel modeling spaces are the MOF and ECore (ECore is a meta-model very similar to but also different from the MOF; although

ECore is a little simpler than the MOF, they are both based on similar object-oriented concepts). As another example, the RDF(S) MS and the MOF MS can also be used as parallel spaces.

- Orthogonal spaces – one modeling space models concepts from another modeling space, taking them as real-world things, i.e. one MS is represented in another MS. This relation is often used in round-trip engineering to facilitate various stages of modeling a system. For example, in order to make a Java program we could first use a Java UML profile to create classes and method bodies, then transform this UML model into Java code, and complete the Java program using a Java Integrated Development Environment. Orthogonal modeling spaces are also used when a conceptual modeling space is implemented using a concrete modeling space – for example, when one is developing an MOF-based repository to run in a Java virtual machine. If we use the MOF to define an RDFS or OWL metamodel, the MOF and RDFS spaces are orthogonal.

The MOF modeling space is the shining star of the MDA. Its meta-metamodel, the MOF, defines the core concepts used to specify all the metamodels that the MDA is based on (UML, CWM, and the MOF itself) and many other domain-specific metamodels. Software engineers are often confused about how these concepts are represented. The MDA includes a standard for representation (or serialization) of MOF-based models using XMI (which is an XML-based representation of objects), so when software engineers see XMI they often think it is UML (or the MOF).

The truth is a little different. The XMI document that they are looking at is an XML document that models concepts from the MOF modeling space in the XML modeling space using XML concepts such as “element” and “node”. XML is then modeled with concrete XML syntax in the EBNF space. The same applies to the Java Metadata Interface (JMI) standard [Dirckze, 2002]; it models MOF concepts using Java constructs that enable the implementation of MOF-based repositories. Both the MOF XMI and the JMI standards are in fact standards for modeling MOF space models in other, concrete modeling spaces.

In Fig. 5-5A we can see several modeling spaces included in the MDA and their orthogonal, representation-oriented relations that, depending on the context, many form a “modeling circle” that can often be confusing. The Mona Lisa, the painting, is modeled in the MOF space using the concept of Object. This concept exists only as an idea – the MOF space is a conceptual modeling space, and hence it needs some kind of syntax for representation. Concrete modeling spaces such as those based on EBNF (Java and XML) can be used to represent the MOF concepts. This is the second stage of the representation in Fig. 5-5 – the object monaLisa in the

MOF space is modeled in the EBNF space as the RefObject monaLisa (RefObject is a part of the JMI specification).

Such concrete concepts from XMI and JMI can be (and often are) also modeled using their corresponding MOF-based metamodels or UML profiles, bringing them back to the MOF modeling space. In Fig. 5-5, the monaLisaRefObject is an instance of the corresponding concept in the MOF-based Java metamodel in the MOF modeling space. The MOF space is involved two times in this “chain” of representations. First the model in the MOF space (monaLisa:modeldomain.Object) is in the M1 layer, but later it descends to the M0 layer in the hierarchy although the other MOF model is used in the M1 layer.

However, the MDA is not the only standard for model-driven architectures – there is also EMF. Figure 5-5B shows the same real-world concept, the Mona Lisa painting modeled in two different modeling spaces in parallel. It is often necessary to move completely from one modeling space to another by means of bridges; there is less room for confusion because one modeling space is translated into another without changing the modeling layers. In this case, metamodels from the MOF modeling space (UML, ODM, etc.) translated into the corresponding ECore-based metamodels will still be in the M1 layer.

5.5 A Touch of RDF(S) and MOF Modeling Spaces

The usage scenarios for parallel spaces most often pertain to conceptual MSs that model the same reality using different concepts. Each of these MSs is implemented in some other, more concrete MS, as a represented reality. In order to exchange models between conceptual MSs, it is necessary to provide transformations from one space to another. These transformations are also models [Bézivin et al., 2003], and should be developed in an MS that can represent both the source and the target MS. Moreover, the transformation also has to be represented in some concrete MS orthogonal to the source and target MSs, which leads from the conceptual model of the transformation to its implementation.

Figure 5-6 shows two parallel conceptual MSs, the RDF(S) MS and the MOF MS, and the space that represents them orthogonally, EBNF MS. MOF and RDF(S) model the real world in parallel, using modeling languages (UML, ODM, OWL or another language) that are defined using different meta-meta concepts. At the conceptual level, we could establish a transformation from one language to another, for example from UML to ODM and vice versa, in the same MS. An example of a transformation

modeling language for such purposes in the MOF is Query-View-Transformation (QVT) [OMG QVT, 2003]. RDF and RDF Schema, and three different dialects of OWL, namely OWL Full, OWL DL and OWL Lite are examples of languages in the M2 layer of RDF(S) MS. Efforts to develop query and transformation language in the RDF(S) MS are underway: Triple, RQL etc. [Haase et al., 2004].

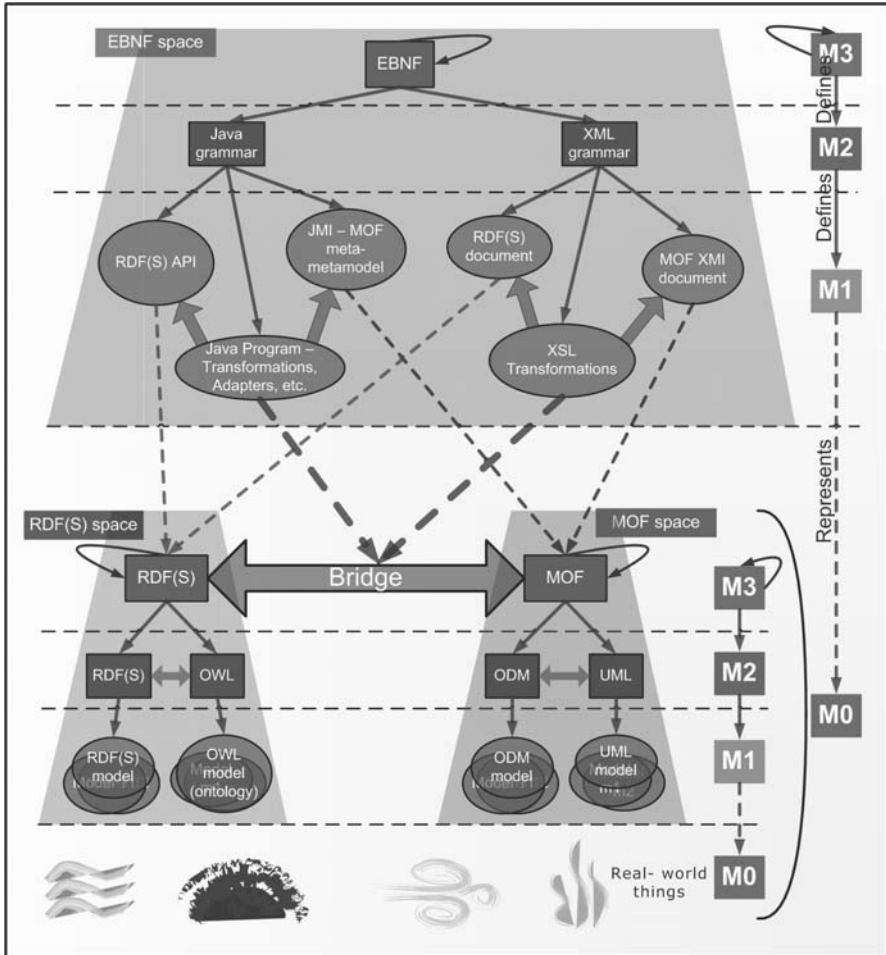


Fig. 5-6. Transformations between the RDF(S) MS and the MOF MS

We can also establish a transformation between the MSs, a bridge that transforms RDF(S) concepts to MOF concepts in the M3 layer. Using that bridge, we can transform any metamodel (language) defined in RDF(S) into its corresponding metamodel defined in the MOF (both are in the M2

layer). Of course, we can expect an information loss depending on how similar the meta-meta concepts (M3) are. RDF(S) concepts, `rdfs:Class` and `rdf:Property` are similar to, but not the same as the MOF Class, Association and Attribute.

Both the MOF and the RDF(S) spaces can be represented in other, more concrete MSs. They can be implemented using repositories, serialized into XML etc., which involves many MSs. For the sake of simplicity, we have skipped a few steps and have shown these spaces as Java program code and XML documents in the EBNF space. Models from the MOF space are modeled in Java code according to the JMI standard, and in XML according to the MOF XMI standard. For languages from RDF(S), XML is the natural method of representation. They can also be modeled using Java APIs (Jena etc.).

As the RDF(S)-MOF bridge is also a model; it can be also represented in a concrete MS representing meta-metamodels that should be bridged. Examples include an XSLT that transforms an MOF XMI document into RDF(S) XML document and vice versa, a set of Java classes that adapt JMI interfaces to RDF(S) API interfaces, and a Java program that does a batch transformation from a JMI-based code to an RDF(S) API-based one.

As Fig. 5-6 shows explicitly, a single bridge models a transformation between two MSs in the M3 layer, between RDF(S) and MOF meta-metamodels. Transformations between metamodels situated in a single MS in the M2 layer are internal to that MS. However, they can be implemented through concrete MSs (e.g. EBNF for XSLT).

5.6 A Touch of the Semantic Web and MDA Technical Spaces

Modeling Spaces are a concept inspired by the concept of a technical space (TS), which is defined as a working context with a set of additional concepts, body of knowledge, tools, required skills, and possibilities [Kurtev et al., 2002]. Fortunately, we can use MSs to enhance this fuzzy definition of TSs.

A technical space is a working context that includes various related MSs. Most often the TS is built around some MS, whereas the role of other MSs is supportive (e.g., implementation), or implicit (literature, know-how). For example, the MOF MS is at the center of the MDA TS. However, the MDA TS also partially includes other MSs: XML and EBNF in the area of XMI representation, EBNF in the area of repository implementation (JMI), an implicit MS that includes literature, etc.

Transformations, for example to plain Java, C++ or VB code, are also models belonging to one or several MSs that are included to some extent in the MDA TS.

Figure 5-7 shows the overlap between the MDA TS and the Semantic Web TS in some MSs (most of the MSs belonging to these TSs have been omitted for the sake of simplicity). The MDA TS is built around the MOF MS, which resides completely in the MDA TS. The MDA TS also includes OWL ontologies that model MOF-based concepts or contain knowledge about the MDA, and which are parts of the RDF(S) MS. On the other hand, the Semantic Web TS includes the RDF(S) MS. Additionally, it includes parts of the MOF MS related to the ODM metamodel and the Ontology UML Profile, and two-way transformations from these MOF-based models to OWL. These transformations are also part of the MDA TS. Recall that those transformations are also modeled, so they belong to some MSs as well. Some researches are trying to identify a way to enable transformations between different MSs in the M3 layer using just one two-way transformation for all three layers [Bézivin et al., 2005].

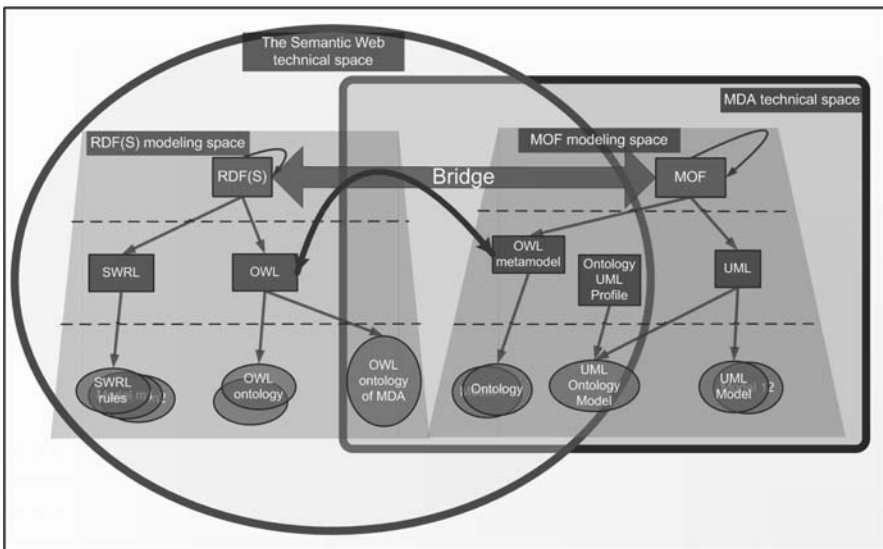


Fig. 5-7. The Semantic Web and Model Driven Architecture technical spaces

It follows from the above discussion that one technical space includes one or more modeling spaces and that every modeling space is a part of one or more technical spaces; a technical space is a means for grouping modeling spaces that have something in common or simply need to

interact. A bridge connecting two modeling spaces is also a means for connecting the surrounding technical spaces.

5.7 Instead of Conclusions

Modeling spaces abstract and generalize a vast number of diverse modeling approaches, and can help engineers see the big picture of what underlies the software that they make or use. They also clarify the structures of different modeling approaches, their similarities and dissimilarities, their mutual relations, and how they work together. By making a clear distinction between conceptual and concrete, and parallel and orthogonal modeling spaces, developers can successfully select mechanisms to automate the transfer and sharing of information, knowledge, and even other modeling spaces between different projects and applications. Likewise, understanding specific modeling spaces helps one to select suitable modeling and development tools for a specific project.

In this book, we focus on a few modeling spaces, most notably the RDF(S) MS, the MOF MS and XML, and a few technical spaces, namely the Semantic Web TS, the Model Driven Architecture TS and the XML TS. The above discussion helps you to visualize what is going on.

Part II The Model Driven Architecture and Ontologies

6. Software Engineering Approaches to Ontology Development

Ontologies, as formal representations of domain knowledge, enable knowledge sharing between different knowledge-based applications. Diverse techniques originating from the field of artificial intelligence are aimed at facilitating ontology development. However, these techniques, although well known to AI experts, are typically unknown to a large population of software engineers. In order to overcome the gap between the knowledge of software engineering practitioners and AI techniques, a few proposals have been made suggesting the use of well-known software engineering techniques, such as UML, for ontology development [Cranefield, 2001a]. However, software engineering approaches themselves do not enable the representation of ontology concepts derived from description logics, and such concepts are included in modern Semantic Web ontology languages (e.g., RDF, RDF Schema, and OWL).

6.1 A Brief History of Ontology Modeling

In this section we describe the existing efforts to enable the use of UML, as well as MDA-based standards, for ontological engineering. We explain the formal background of each approach and its relation to ontology languages. The idea is to recognize all similarities and differences between the two modeling spaces, and thus get a clearer picture about all issues that should be considered when bridging them.

6.1.1 Networked Knowledge Representation and Exchange Using UML and RDF

The idea of using UML in ontological engineering was first suggested by Cranefield [Cranefield, 2001a]. Cranefield found connections between the standard concepts of UML and those of ontologies: classes, relations,

properties, inheritance, etc. However, he also observed certain dissimilarities between them. The most important difference is related to the concept of a property— in UML, an attribute’s scope is the class that defines it, whereas in an ontology a property is a first-class concept that can exist independently of a class. Here are the main presumptions of Cranefield’s proposal for UML-based ontology development:

- UML class diagrams provide a static modeling capability that is well suited to representing ontologies, that is to say, modeling the taxonomy of the ontology’s concepts and the relations between those concepts (i.e., the schema level of the ontology). For an example, see Fig. 6-1.
- UML object diagrams can be interpreted as declarative representations of knowledge, that is to say, they can be used for modeling instances of an ontology (i.e., body of knowledge) [Chandrasekaran et al., 1999]. For an example, see Fig. 6-2.
- Object Constraint Language (OCL) can be used for specifying constraints on an ontology.
- The same paradigm can be used for modeling both ontologies and knowledge, which can be rather useful for software developers.

However, Cranefield recognized one significant current shortcoming of UML: the lack of a formal definition [Cranefield, 2001b]. The semantics of UML are defined by a metamodel, some additional constraints expressed in a semiformal language (i.e., OCL), and descriptions of the various elements of the language in plain English. Cranefield argued that the use of the basic features of class and object diagrams for representing ontologies and knowledge seems no more prone to misinterpretation than does the use of the Resource Description Framework – a language which underlies the Semantic Web but also lacks official formal semantics. Thus, he proposed an approach for integrating UML and RDF(S).

Cranefield’s approach is founded on the following set of standards, which he recognized as important technological requirements for UML representations of ontologies and knowledge:

- XML Model Interchange (XMI), which stands for an XML-based standard for serializing UML models;
- well-known UML tools (e.g. IBM Rational Rose, Poseidon for UML, and MagicDraw) that can export the present UML artifacts in the UML XMI format;

- the Resource Description Framework (RDF) and RDF Schema (RDFS) – together often referred to as RDF(S), which is a W3C XML-based standard for sharing ontologies on the Semantic Web;
- eXtensible Stylesheet Language Transformations (XSLTs), which Cranefield used to transform UML XMI into:
 - a set of Java classes and interfaces corresponding to those in the ontology; and
 - RDF(S) (actually UML classes are transformed into RDFS, and UML objects into RDF statements).

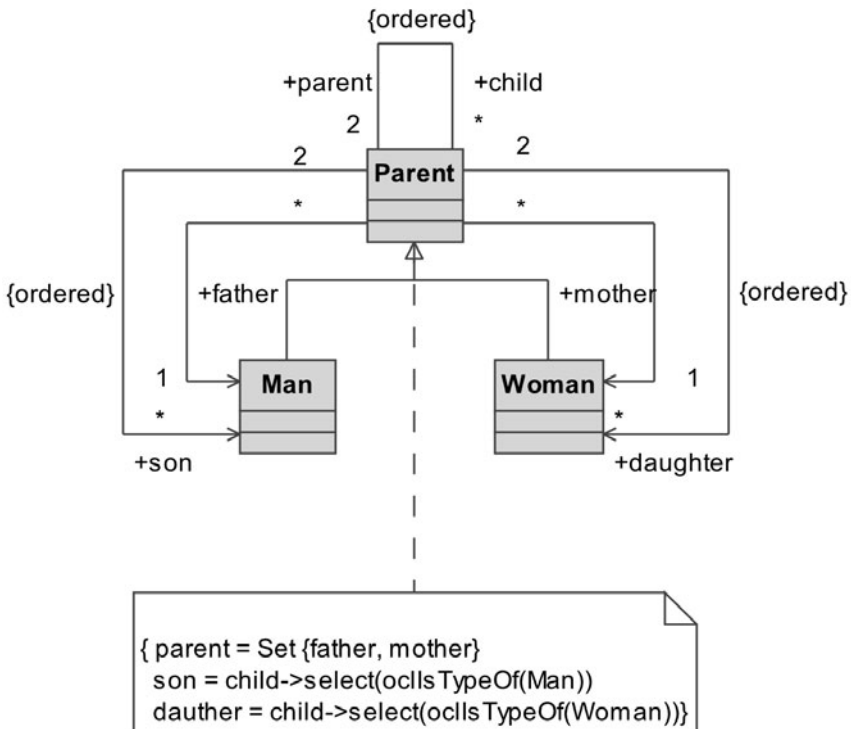


Fig. 6-1. A part of the Family ontology represented in the form of a UML class diagram, developed by Stephen Cranefield

The main problem that Cranefield had to deal with when generating an RDF schema that corresponds to a UML model is that RDF properties are first-class objects and are not constrained to the context of a particular class. This can lead to conflicting domain declarations if two (or more) different classes have equally named field attributes. He decided to prefix the name of each such field with the name of the class that it belongs to

(e.g., `Person.name`). However, the disadvantage of this solution becomes obvious in the presence of inheritance, when fields of one class may be represented by properties with different prefixes, some referring to the class itself and others to the parent class.

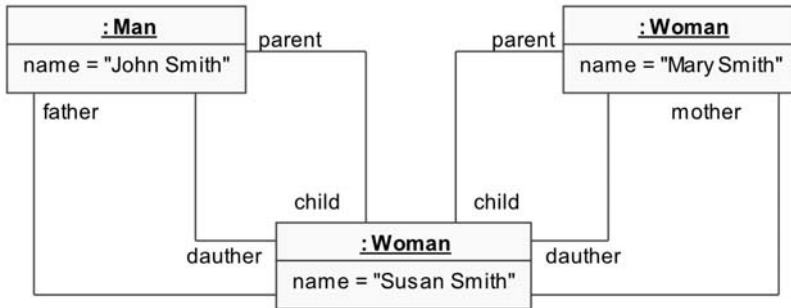


Fig. 6-2. A UML object diagram representing instances of the ontology classes shown in Fig. 6-1

Besides the problem of transforming UML attributes and associations into RDF properties, Cranfield solved additional problems such as the following:

- When a UML attribute/association end has a multiplicity upper limit greater than a specific value, the XSLT used transforms that into an RDFS bag.
- When association ends have a UML “ordered” constraint, the XSLT transforms them into RDFS sequences.

Finally, note that Cranfield needed to extend the RDFS definition generated by the XSLT. He did this because there is no mechanism in RDFS to parameterize a collection type (such as `rdf:Bag`) with the type of elements that it may contain. Therefore, a nonstandard property `rdfsx:collectionElementType` was introduced to represent this information.

Figure 6-3 presents the RDFS document generated by applying Cranfield’s XSLT to the UML classes shown in Fig. 6-1.

Concluding overview of the first and most cited UML-based approach to ontology development, we need to emphasize some of its limitations as well (especially since they propagate to the generated languages):

- One cannot deduce whether the same property has been attached to more than one class in the case of property inheritance.
- A hierarchy of properties cannot be created, even though this is allowed in RDFS.
- Standard UML cannot express advanced features of ontologies (e.g., OWL restrictions).
- The target RDFS ontology description does not have advanced restriction concepts (e.g., multiplicity).

```

<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF xml:lang="en"
  xmlns:rdfsx="http://nzdis.otago.ac.nz/2000/01/rdf-schema-extensions#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdfs:Class rdf:ID="http://nzdis.otago.ac.nz/0_1/family#Person"/>
  <rdf:Property ID="http://nzdis.otago.ac.nz/0_1/family#Person.name">
    <rdfs:domain rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Person"/>
    <rdfs:range rdf:resource="rdfs:Literal"/>
  </rdf:Property>
  <rdf:Property ID="http://nzdis.otago.ac.nz/0_1/family#Person.parent">
    <rdfs:domain rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Person"/>
    <rdfs:range rdf:resource="rdf:Bag"/>
    <rdfsx:containerElementType rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Person"/>
  </rdf:Property>
  <rdf:Property ID="http://nzdis.otago.ac.nz/0_1/family#Person.child">
    <rdfs:domain rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Person"/>
    <rdfs:range rdf:resource="rdf:Seq"/>
    <rdfsx:containerElementType rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Person"/>
  </rdf:Property>
  <rdf:Property ID="http://nzdis.otago.ac.nz/0_1/family#Person.father">
    <rdfs:domain rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Person"/>
    <rdfs:range rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Man"/>
  </rdf:Property>
  <rdf:Property ID="http://nzdis.otago.ac.nz/0_1/family#Person.son">
    <rdfs:domain rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Person"/>
    <rdfs:range rdf:resource="rdf:Seq"/>
    <rdfsx:containerElementType rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Man"/>
  </rdf:Property>
  <rdf:Property ID="http://nzdis.otago.ac.nz/0_1/family#Person.mother">
    <rdfs:domain rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Person"/>
    <rdfs:range rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Woman"/>
  </rdf:Property>
  <rdf:Property ID="http://nzdis.otago.ac.nz/0_1/family#Person.daughter">
    <rdfs:domain rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Person"/>
    <rdfs:range rdf:resource="rdf:Seq"/>
    <rdfsx:containerElementType rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Woman"/>
  </rdf:Property>
  <rdfs:Class rdf:ID="http://nzdis.otago.ac.nz/0_1/family#Man">
    <rdfs:subClassOf rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Person"/>
  </rdfs:Class>
  <rdfs:Class rdf:ID="http://nzdis.otago.ac.nz/0_1/family#Woman">
    <rdfs:subClassOf rdf:resource="http://nzdis.otago.ac.nz/0_1/family#Person"/>
  </rdfs:Class>
</rdf:RDF>

```

Fig. 6-3. An RDF schema generated from the UML model shown in Fig. 6-1 using the XSLT that Cranefield developed

Regardless of all the aforementioned constraints of this approach, however, it is a quite useful solution and can be considered as a kind of agile UML-based approach to ontology development that bridges two different modeling spaces.

6.1.2 Extending the Unified Modeling Language for Ontology Development

Cranefield's pioneering work on using software engineering techniques for ontology development inspired other authors, and soon more advanced solutions to this problem started to emerge. Baclawski and colleagues have stressed that many mature UML tools, and models and much expertise already exist and can be applied to knowledge representation systems, not only for visualizing complex ontologies but also for managing the ontology development process. They have introduced two UML-based approaches to ontology development [Baclawski et al., 2002a; Baclawski et al., 2002b]. This and the next subsection describe both of these solutions.

In their first proposal, Baclawski extended the UML metamodel with the constructs of the RDFS and DAML+OIL languages. Though their solution focuses only on the similarities and differences between the UML and DAML+OIL languages, they stated that it can be applied to other knowledge representation languages, such as

- logical languages that express knowledge as logical statements (e.g., KIF);
- frame-based languages that are similar to object-oriented database languages;
- graph-based languages that include semantic networks and conceptual graphs.

Baclawski et al. tried to solve the problem of using UML for developing ontologies by defining appropriate mappings between the two modeling languages. Therefore they defined general-level mappings between UML and DAML+OIL (see Table 6-1). Additionally, they went into a profound theoretical discussion of their similarities and differences of these languages. Here we list only some of their most important findings (for details see [Baclawski et al., 2002a]):

- Knowledge representation languages do not have metalevel separation (e.g., an instance of a class may be a class).

- The concepts of classes and instances are common features of the two languages. This allows for consistent mappings of these concepts.
- Properties are a stand-alone concept, which is the main problem for transforming between the two languages.
- UML multiplicity constraints on associations can affect the membership of objects in classes related through an association. In DAML+OIL, constraints on properties are imposed somewhat indirectly by specifying that a class is a subtype of a class called “restriction.”

Table 6-1. Rough overview of the relations between the concepts of the DAML+OIL and UML languages

DAML concept	Similar UML concept
Ontology	Package
Class	Class
As sets (union, intersection, etc.)	Not supported
Hierarchy	Class generalization relations
Property	Aspects of attributes, associations and classes
Hierarchy	None for attributes, limited generalization for associations, class generalization for relations
Restriction	Constraints of association ends, including multiplicity and roles, implicitly as a class containing the attribute
Data types	Data types
Instances and values	Object instances and attribute values

- The semantics of the specialization/generalization relation are not equivalent:
 - UML’s relation tends to provide behavioral reuse of modeled concepts. For example, in a graphical tool we would model the class Square so as to be the parent of the class Rectangle; the former class would have one attribute describing the dimensions of a graphical figure, while the latter class would have two attributes (width and height).
 - DAML+OIL’s relation is set-theoretic (e.g. Rectangle is a parent of Square), which should closely reflect the relations between concepts in reality.

- Finally, Baclawski et al. concluded that the mapping between these two relations is consistent, even though they are not semantically equivalent.
- RDF has a number of container notions: Bag, Seq and Alt. The semantics of these notions are not very clear, and DAML+OIL has largely replaced them with the notion of a List. UML does have containers (in OCL), and it also has ordered associations which implicitly define a list within the context of an association.
- DAML+OIL has the ability to construct classes using boolean operations (union, intersection and complement) and quantifiers.
- RDF allows one property to be a subproperty of another. UML has the ability to specify that one association is a specialization of another, although this construct is rarely used.
- Constructs in Semantic Web ontology languages are linked together either through the use of URIs or by using the hierarchical containment relationship of XML. UML uses names in a large number of namespaces. For example, each class has its own namespace for its attributes and associations. This is enriched by an ability to specify private, protected, and public scopes. Any mapping from UML to DAML+OIL or the reverse must have a mechanism for ensuring that names are properly distinguished.

Besides the aforementioned incompatibilities between UML and DAML+OIL, Baclawski and colleagues have stressed another important aspect. Ontology languages are monotonic, whereas UML and object-oriented languages are nonmonotonic. A system is monotonic if adding new facts can never cause previous facts to be negated. Such defined monotonicity is a necessity for realizing the goals of the Semantic Web. It supports representation of the diversities common to such a heterogeneous environment. Baclawski et al. considered modeling the concept of a *person* in order to depict this difference. They assumed that it has been specified that every person must have a name. Consider what would happen if a particular *person* object did not have a name. In UML, this situation would be considered as a violation of the requirement that every person must have a name, and a suitable error message would be generated. In a monotonic logic, on the other hand, one cannot draw such a conclusion. Baclawski et al. concluded that this distinction between UML and ontology languages makes it effectively impossible to define a mapping between these languages that would be able to preserve their semantics. However, it does

not prevent one from defining a mapping that preserves semantic equivalence, and that is all that we are attempting to achieve.

A natural solution to bridging these two different languages is to extend UML with primitives that are equivalent to ones from DAML+OIL. However, solutions that use UML profiles, such as DUET, do not preserve semantics. Accordingly, Baclawski and colleagues made a specific recommendation for modifying the UML metamodel, thus enabling one to model using first-class properties, as well as to construct classifiers using boolean operations and quantifiers. In Fig. 6-4, we illustrate how MOF-based concepts of the UML metamodel (e.g., *Classifier* and *GeneralizableElement*) were extended with concepts from DAML+OIL (e.g., *DerivedClassifier*, *Restriction*, *Union*, *Intersection*, *Complement*, and *Property*) using the MOF's generalization/specialization relation. Fig. 6-5 shows the associations among the newly added concepts in the UML metamodel.

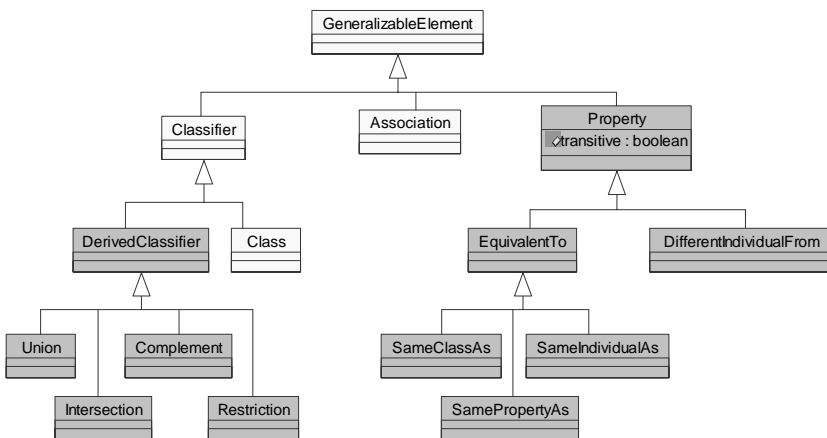


Fig. 6-4. The hierarchy of the metaclasses that are part of the extensions of the UML metamodel for DAML+OIL proposed in [Baclawski et al., 2002a]

Since the development of a new tool able to support this extended UML metamodel would be expensive and time-consuming, Baclawski and colleagues proposed a UML profile based on their metamodel. Models based on that UML profile can be developed using tools for standard UML that are already available. In Fig. 6-6, we give an example of a UML diagram that Baclawski and colleagues developed in order to illustrate how to specify an association between the *Faculty* class and the *Organization* class. The *affiliation* association end belongs to the *location* property, which states that a *location* can be a *University*, a *School*, or an *Institute* that is not a *Company*.

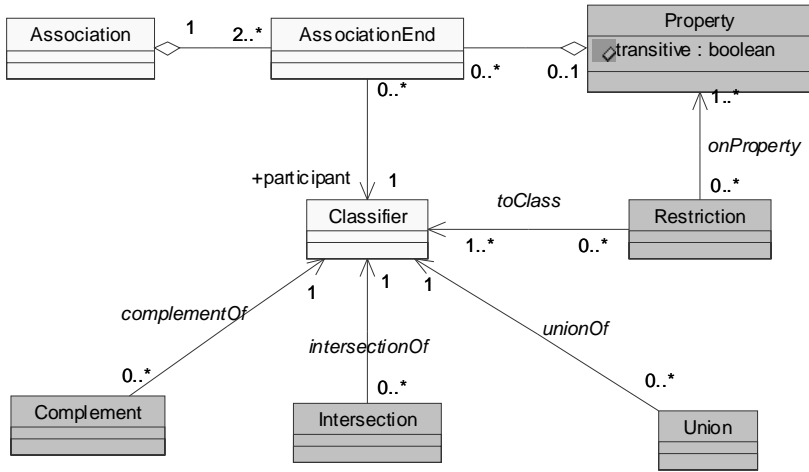


Fig. 6-5. The associations among meta-classes that are part of the extensions of the UML metamodel for DAML+OIL proposed in [Baclawski et al., 2002a]

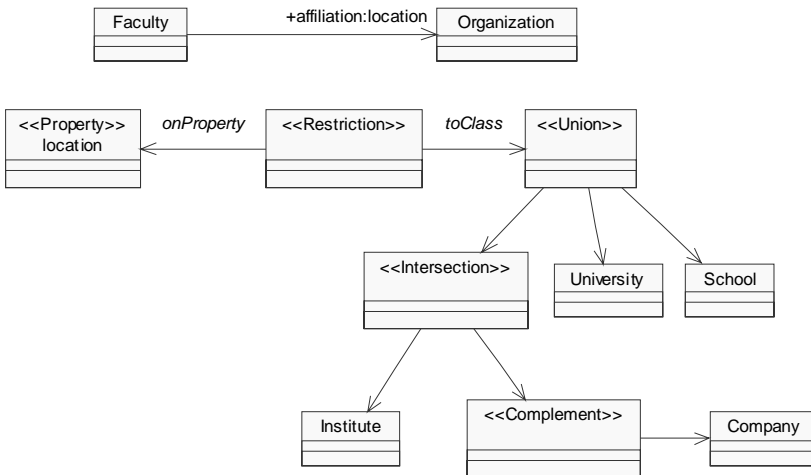


Fig. 6-6. An example of a UML model based on the UML extension for ontology development

Although this approach is the most comprehensive theoretical contribution to identifying relations between UML and DAML+OIL, it still has some shortcomings:

- DAML+OIL has evolved into OWL, which is the current W3C standard.
- Baclawski et al. did not develop any tool that would be able to map the extended UML into a Semantic Web language. Though standard UML tools could be used for developing ontologies, we still need a transformation tool for mapping models from one modeling language into another.
- An extended UML metamodel is an awkward solution, as we do not need all UML concepts for ontology development. Actually, this issue was also recognized by Baclawski et al. themselves, and so they proposed an improved solution based on an independent ontology metamodel.

6.1.3 The Unified Ontology Language

Baclawski and colleagues realized that their initial solution was fairly awkward because it introduced some new concepts to the UML metamodel. Therefore, they have developed an independent ontology metamodel using the MOF; they have named it the Unified Ontology Language (UOL) [Baclawski et al., 2002b]. This metamodel first appeared in public at an OMG meeting organized to initiate work on the Request for Proposals for specification of the Ontology Definition Metamodel [Baclawski et al., 2002b; OMG ODM RFP, 2003]. Although this metamodel was inspired by DAML+OIL, it also took the OWL language into account.

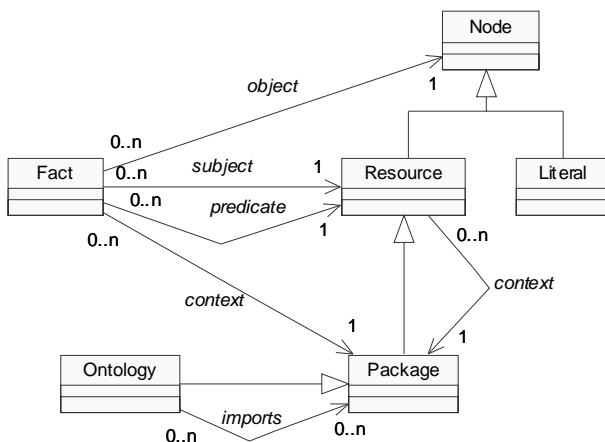


Fig. 6-7. MOF-compliant metamodel of the Unified Ontology Language (UOL)

The proposed UOL should satisfy the following requirements:

- It must be a MOF-based modeling language (i.e. metamodel).
- It must have a bounded two-way mapping between the core UML and the core UOL. The two-way mapping must preserve semantic equivalence on levels 0 and 1 of the MDA.
- The core UOL must include the same notions as the core UML, namely Package, Class, Binary association, Generalization, Attribute, and Multiplicity constraints.

Figure 6-7 gives a UML class diagram of a MOF-based metamodel for UOL which satisfies the aforementioned requirements. Although it seems rather small, it is a very high-level metamodel grounded on RDF and its triple-based paradigm. Furthermore, it is a solid starting point for a standard MOF-based ontology language. It is important to say that a similar solution has been used for defining the present ODM proposal.

6.1.4 UML for the Semantic Web: Transformation-Based Approach

Kateryna Falkovych and her colleagues [Falkovych et al., 2003] have proposed a transformation approach to the semantic extraction of ontologies from UML models. Their initial presumption is that UML and ontologies complement each other. That is to say, UML is designed for building models by human experts, while OWL is designed to be used at run time by intelligent processing methods. However, Falkovych et al. report that the process of translation is less than trivial, owing to the differences between the two languages.

With the above presumptions Falkovych and colleagues have proposed a transformation of UML diagrams into DAML+OIL ontologies which enables the semantics of UML concepts to be preserved. The main motivation for developing such a transformation is the existence of large sources of ontological knowledge already available in the UML design documents of existing applications. Furthermore, one can reason about UML models transformed into ontologies.

Taking into account the theoretical discussion in [Baclawski et al., 2002a], Falkovych and her associates developed rules for transforming UML models into DAML+OIL ontologies. They also stressed differences between properties in an ontology and UML associations/attributes. Since association is unique in UML, it is mapped into `daml:ObjectProperty` with a unique identifier for the property

name. A UML attribute is mapped into `daml:DatatypeProperty` with a unique identifier attached. The rationale for mapping an attribute only to `daml:DatatypeProperty` and not to `daml:ObjectProperty` is that usually the type (range) of an attribute is a data type. Additionally, Falkovych et al. developed a taxonomy of association types in order to preserve semantics of UML associations and to distinguish association ends (see Fig. 6-8). Their taxonomy distinguishes between four subtypes of association: binary association, unidirectional association, aggregation, and composition, which are decomposed further into specific subtypes introduced for the purpose of mapping. This taxonomy is encoded as a separate DAML+OIL ontology, so that one can refer to it from the UML models transformed into DAML+OIL ontologies. Note also that all association types at the lowest level of decomposition (not shown in Fig. 6-8) can either have a name or be unnamed and can have role names attached to association ends.

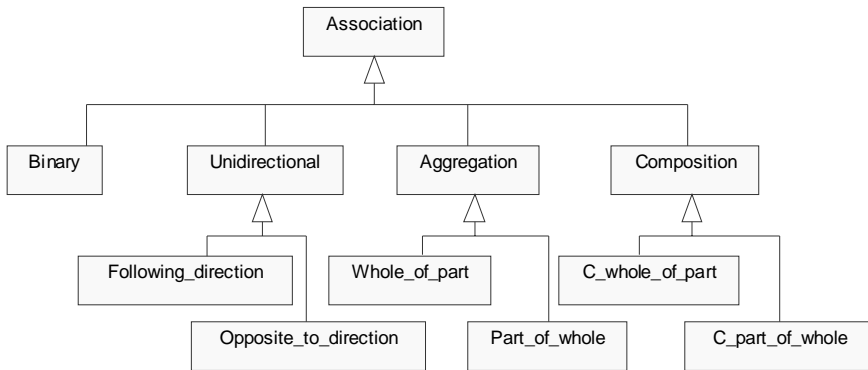


Fig. 6-8. Taxonomy of UML association types

Figures 6-9 and 6-10 depict a UML binary unnamed association connecting two UML classes (*Boundary point* and *Boundary segment*) and its association ends (*has* and *form*). Note that the binary association is mapped into two `daml:ObjectProperty` elements. These two properties have the name of an association with a unique identifier attached to it, and they are distinguished by adding an underscore symbol (`_`) to one of them (i.e., `G.2` and `_G.2`). Furthermore, these two properties are inherited (using `rdfs:subClassOf`) from the corresponding class (*binary_unnamed*) in the taxonomy shown in Fig. 6-8. It is important to point out that association ends (*has* and *form*) are also mapped onto object properties, which are inherited from the properties representing the UML association (`G.2` and `_G.2`). In fact, the original names of these two properties

are extended by adding their UML unique identifiers, and thus their final names are *has_G.4* and *form_G.3*.

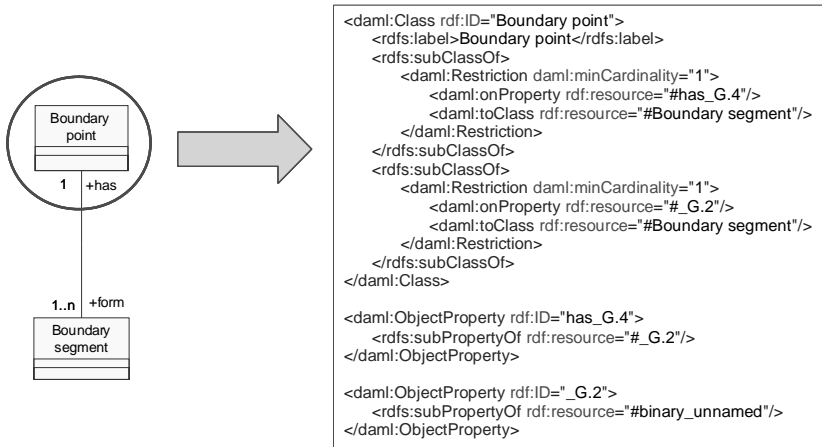


Fig. 6-9. DAML+OIL representation of the *Boundary point* UML class and its association and association end

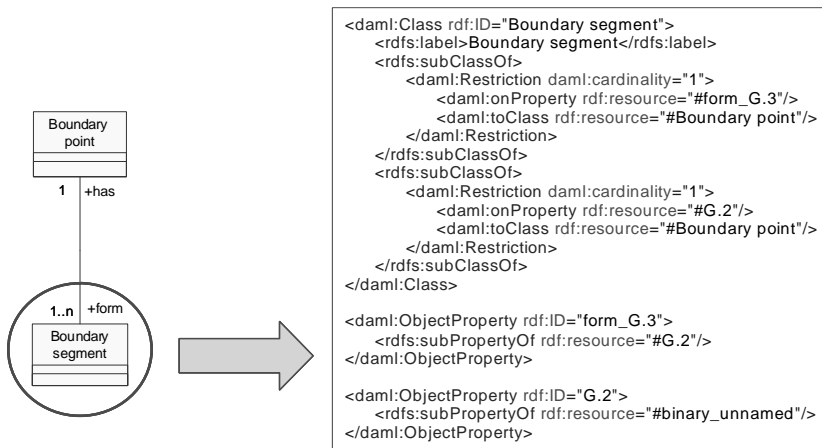


Fig. 6-10. DAML+OIL representation of the *Boundary segment* UML class and its association and association end

Falkovych and her associates have implemented this mapping from UML models to DAML+OIL practically using XSLT. They found that XSLT was a quite cumbersome solution for more complex mappings, as well as very sensitive to the format of the input file.

Besides the very positive results of this approach, we have found that some limitations of this solution are the following:

- The lack of mechanisms for formal property specification in UML (e.g., for defining property inheritance or an *inverseOf* relation between properties). Hence one cannot express properties and other ontology primitives in UML, as was possible with the UML extension proposed by Baclawski and colleagues [Baclawski et al., 2002a]. One might expect this limitation, as the primary goal of the solution was to map UML onto ontologies, and not in the opposite direction;
- The use of UML class diagrams, which contain only graphical artifacts of the real UML elements included in a model (e.g., all associations titled with the same name are assumed to represent the same property, although each association is a distinct model element in UML). Of course, this diagram problem can be partly overcome with XMI for UML 2.0, which supports diagram representation.

6.1.5 The AIFB OWL DL Metamodel

The next proposal that we analyze in this chapter is the one put forward by Sara Brockmans and her colleagues [Brockmans et al., 2004]. These researchers were inspired by the OMG ODM RFP and the initial submissions to that RFP, given in [ODM DSTC, 2003; ODM Gentleware, 2003; ODM IBM, 2003; ODM Sandpiper&KSL, 2003]. Their basic idea was to enrich ODM with several separate metamodels, one for each knowledge representation (e.g., OWL DL), in order to achieve better readability and usability. Additionally, Brockmans et al. proposed the use of OMG metamodel mapping facilities to perform mapping between different knowledge representations [OMG QVT, 2003]. They argued for such an approach rather than for having one core ontology metamodel that would be a common denominator of different knowledge representations and serve for mapping purposes. In fact, they are aware of the fact that their solution results in a higher number of mappings, but argue that each individual mapping is a lot more lightweight and easier to specify and maintain.

On the basis of the above ideas, Brockmans and colleagues developed a metamodel based on OWL DL using MOF2. This metamodel has a one-to-one mapping to the abstract syntax of OWL DL and thereby to the formal semantics of OWL DL. They also defined a UML profile on top of the metamodel. It is interesting to note that this UML profile contains graphical icons for ontology-specific primitives (e.g., restriction, union, and someValuesOf), similarly to the UML profile for developing Web applica-

tions described in Chap. 4 [Conallen, 2002]. One can use this UML profile to develop ontologies in most of the present UML tools (e.g., MagicDraw). However, to the best of our knowledge, the currently available tools do not provide support for transforming either the metamodel or the UML profile into any ontology language (e.g., OWL). Therefore, ontology developers are prevented from automatically employing ontologies developed using UML tools in Semantic Web ontology-based applications.

While this metamodel and UML profile are based on experience from the initial submission to the OMG ODM RFP and on the other MDA-based solutions listed earlier in this chapter, the main contribution of this approach lies in the proposal of the introduction of a separate metamodel for each knowledge representation. In fact, the most recent ODM draft specification at the time of writing (October 2005) [OMG ODM RFP, 2005] has adopted a similar solution. This draft consists of two core metamodels: description logics (nonnormative) and common logics. It also contains three metamodels that represent the abstracts syntax of the languages commonly used by the Semantic Web community (RDFS, OWL, and Topic Maps); and two more traditional software engineering metamodels, Entity Relationship and UML2. In addition, mappings between a number of the metamodels are provided. A more detailed discussion of this OMG ODM draft specification is provided later in the book.

6.1.6 The GOOD OLD AI ODM Proposal

Finally, the authors of this book have also proposed a solution to the OMG ODM RFP. This proposal provides a comprehensive response to the ODM RFP that defines a metamodel based on OWL Full [Djurić et al., 2005a], a corresponding ontology UML profile (see Chap. 9) [Djurić et al., 2005b], the required transformations (see Chap. 10) [Gašević et al., 2005]. On top of these definitions an ontology editor called AIR [Djurić et al., 2006a] has been developed (see Chap. 12). More details about this proposal, the tools developed, and their practical usage for developing ontologies are given later in this book.

6.2 Ontology Development Tools Based on Software Engineering Techniques

In describing the most relevant approaches to applying software engineering techniques to ontology development in the previous section, we aimed to reflect the most important issues differentiating these two modeling

spaces used. In this section we outline some well-known tools that enable the development of ontologies using software engineering languages. In fact, this section discusses only UML-based tools, owing to the wide acceptance of UML and its CASE tools by the software engineering community.

6.2.1 Protégé

Protégé is the leading ontological engineering tool [Noy et al., 2001]. It has a complex software architecture, easily extensible through plug-ins. It is freely available [Protégé, 2005]. Many components providing interfaces to other knowledge-based tools (Jess, Argenon, OIL, PAL Constraint, etc.) have been implemented and integrated in Protégé. In the same manner support is provided for various ontology languages and formats such as XML, DAML+OIL (back ends), and OIL (tab).

Formally speaking, Protégé has a MOF-compatible metamodel (see Fig. 6-11). The main reason for having a shared metamodel is that generic tools operating on any MOF-compliant model can be built [Protégé XMI, 2004]. The metamodel is designed as a MOF-compliant metamodel of Protégé ontologies that resides in the same metalevel as UML. While Protégé has basically been developed on top of an adapted version of the OKBC metamodel (see Sect. 1.7 for details of OKBC), until now there have been no attempts to map OKBC to the MOF. Since the MOF is the dominant standard in the object-oriented-software-technology community (and even more widely), one of the benefits of having a MOF-compliant representation of Protégé ontologies is improved interoperability with other MDA-based systems. Thus, the Protégé metamodel is a MOF-based definition of OKBC.

This metamodel is extensible and adaptable. This means that Protégé can be adapted to support a new ontology language by adding new meta-classes and metaslots to the Protégé ontology. The introduction of these new metamodeling concepts enables users to add the necessary ontology primitives (e.g., the Protégé class has different features from the OWL class). In this way it can, for instance, support RDFS [Noy et al., 2000] and OWL [Knublauch et al., 2004].

As a support for sharing MOF-compatible models, Protégé has two back ends: UML [Protégé UML, 2004] and XMI [Protégé XMI, 2004]. These two back ends use the NetBeans MetaData Repository (MDR) (available from <http://mdr.netbeans.org>). NetBeans is an implementation of the Java Metadata Interface (JMI) specification [Dirckze, 2002]. JMI basically takes a MOF metamodel and generates Java interfaces that can be used to

access model instances at run time. These interfaces provide an alternative mechanism to access and build Protégé metamodels. Using those interfaces, it would be possible to plug Protégé into other tools, such as Java IDE NetBeans. Note also that the NetBeans MDR has features for exporting/importing XMI models, metamodels, and metametamodels.

The Protégé XMI back end uses an XMI schema (see Sect. 4.7) compliant with the Protégé MOF-defined metamodel shown in Fig. 6-11. In fact, this back end relies on features of the NetBeans MDR for importing and exporting XMI. One can find an example of a Protégé ontology represented in the Protégé XMI format in [Protégé XMI, 2004].

The Protégé UML back end exchanges UML v1.4 models (i.e., classes, and their relations), that is to say, models encoded in versions 1.1 and 1.2 of UML XMI. The back end uses the following rules to perform mapping between UML models and its own MOF-compatible metamodel [Protégé UML, 2004]:

- Each UML class is represented by one Protégé class of the same name, and with the same role (i.e., abstract or not abstract).
- The generated Protégé classes are arranged in an inheritance hierarchy as represented in the UML model (UML allows multiple inheritance). Since Protégé does not support concepts such as “interface”, all UML interfaces are handled as classes.
- Each attribute of a UML class is represented by a slot of a suitable type. The new slot will have the name of the attribute, unless this name has already been taken by a different slot (from another UML class) with a different type. In this case, the slot will be renamed according to the format <attributeName>@<className>, which ensures unique slot names.
- The new Protégé slots obtain their multiplicity (which allows multiple values or not) and their minimum and maximum cardinalities from the UML model.
- During export of UML primitive slots (e.g., int, string and symbol) are converted to simple attributes that are attached to all UML classes where the slot is used. Nonprimitive (instance) slots are translated into UML associations, whereby inverse slots are used to create bidirectional associations.
- Metaclasses are recognized by means of the <<metaclass>> stereotype for classes.

The developers of the Protégé UML backend also list a few limitations, namely:

- Instances (i.e. UML objects and ontology class instances – individuals) cannot be shared.
- UML associations of higher order (e.g., ternary relationships) have not been tested sufficiently.

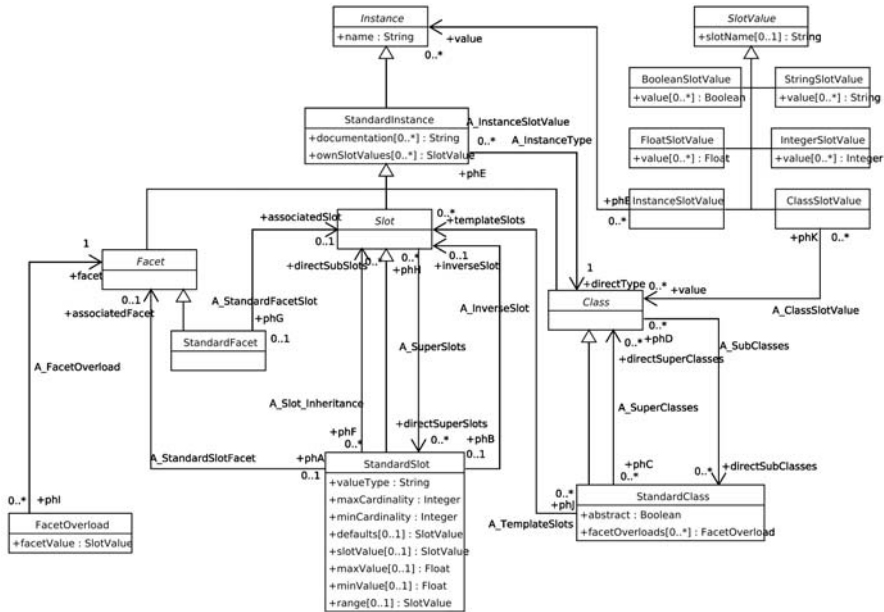


Fig. 6-11. Protégé MOF-compatible metamodel, based mainly on OKBC [Protégé XMI, 2004]

- Inherited Protégé slots/UML attributes (defined for parent classes) are not recognized;
- Protégé does not provide real metamodel support for UML. Specifically, the Protégé metamodel (see Fig. 6-11) must be extended with specific concepts of UML, as has been done for various ontology languages (e.g., OWL and RDF) and knowledge representation languages (e.g., Jess and Algernon). For example, the *StandardClass* of the Protégé metamodel has been extended with the *UMLClass* in order to emphasize the differences between those two concepts. In fact, this is the same principle that Baclawski and colleagues [Baclawski et al., 2002a] applied when extending the UML metamodel to support the DAML+OIL language. Of course, here we have an opposite case, where the Protégé metamodel, which can be regarded as an ontology language, is extended

with concepts from the UML metamodel. Actually, a Protégé profile is created (somewhat like a UML profile).

It is obvious from the above discussion that one can transform UML models into many different ontology languages thanks to the abundance of Protégé plug-ins and back ends. For instance, one can import a UML model represented in the UML XMI format into Protégé. Afterwards, one can either store that model in OWL or RDF(S) using the OWL plug-in, or transform it into a knowledge base format native to Jess (Java Expert System Shell; see <http://herzberg.ca.sandia.gov/jess/>), a rule-based engine, using JessTab (<http://www.ida.liu.se/~her/JessTab/>). The model can then be used in applications based on Semantic Web reasoners such as Jena [McBride, 2002] or OWLJessKB [Kopena & Regli, 2003]. In fact, this model-sharing scenario reflects the future capacities of ontology-driven software development in the context of the Semantic Web [Knublauch, 2004].

Regarding support for the present well-known UML tools, Protégé can share models directly with Poseidon for UML and MagicDraw, as they support versions 1.1 and 1.2 of UML. However, the IBM Rational Rose XMI add-in supports only XMI for UML v1.3, and thus direct model-sharing with Protégé is not possible. As a way around this problem, the developers of the Protégé UML back end recommend users to first import IBM Rational Rose UML XMI models into Poseidon for UML, and then export them into XMI models. XMI models created in such a way can be imported by the Protégé UML back end. In this chapter, we do not give detailed guidelines for sharing models between Protégé and UML tools, but detailed guidelines for Poseidon for UML, MagicDraw are provided in Chap. 12 in the form of a step-by-step procedure.

6.2.2 DUET (DAML UML Enhanced Tool)

A software tool called DUET [DUET, 2005] enables the importing of DAML ontologies into IBM Rational Rose and ArgoUML and the exporting of UML models into the DAML ontology language. The tool is actually implemented as an add-in for IBM Rational Rose (see Fig. 6-12) and as a plug-in for ArgoUML. It is freely available.

This tool uses a quite simple UML profile that contains stereotypes for modeling ontologies (based on a UML package) and properties (based on a UML class). The mappings from the UML profile onto DAML shown in Table 6-2 are taken from [DUET, 2005]. We advise readers to look also at the mappings from the DAML language to the DUET UML profile in

[DUET, 2005]. Note also that DUET uses an XSLT that transforms RDFS ontologies into equivalent DAML ontologies. In this way, an RDFS ontology can be imported into UML tools through the DAML language.

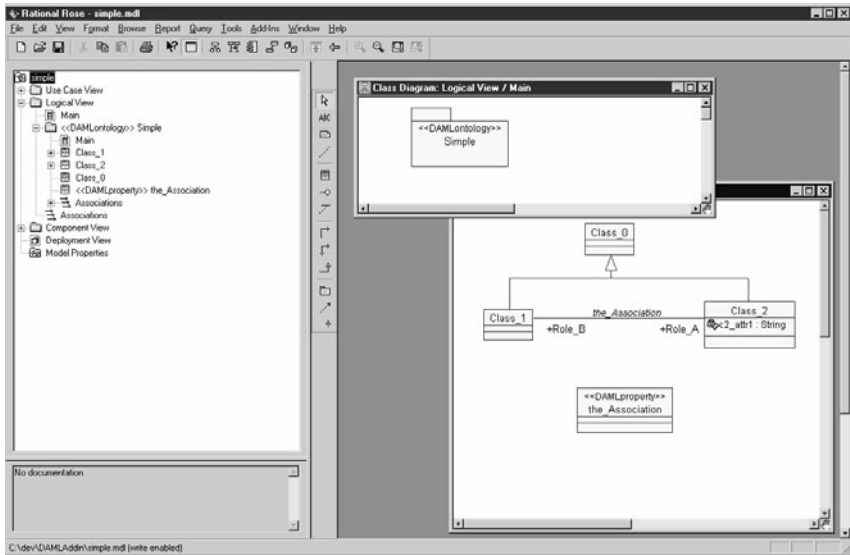


Fig. 6-12. A screen shot of the DUET IBM Rational Rose add-in

DUET is the first UML tool extension that enables sharing of ontologies between an ontology language (i.e., DAML) and a UML tool in both directions. However, since this tool uses a UML profile-based approach, it is not aware of the semantics of UML concepts, as would be the case if it had been based on a metamodel. Instead, DUET just provides a graphical notation. Furthermore, this is a specialized DAML language tool, which can lead to the loss of some semantic information when one is dealing with other languages such as OWL, an official W3C specification. To the best of our knowledge, this tool has not been updated in the last two years (at the time of writing, October 2005).

6.2.3 An Ontology Tool for IBM Rational Rose UML Models

Xpetal is a freely available tool implemented in Java that transforms IBM Rational Rose models from the *mdl* format to RDF and RDFS ontologies [deVos & Rowbotham, 2001; deVos et al., 2001]. The tool has developed as a Java API that can be used from other applications or launched from a command prompt (see [XPetal, 2002] for details). The target RDFS ontol-

ogy is constructed according to the UML-to-RDF mappings in the OMG DAF specification [OMG DAF, 2005]. Figure 6-13 shows a simple UML model of Petri nets that we developed as a starting point for creating a Petri net ontology. After transforming the model from the *mdl* format into the RDFS format, we imported it into Protégé.

Table 6-2. Mappings from the DUET UML profile to the DAML language [DUET, 2005]

UML	DAML	Discussion
<<DAMLontology>> Package	Ontology	Packages stereotyped <<DAMLontology>> become DAML ontologies. Package name becomes Ontology URI.
Class	Class	UML classes without stereotypes become DAML classes. UML class ownership of attributes and aAssociations is not strongly represented in the DAML classes. No operation represented.
<<DAMLproperty>> Class	Property	A UML class stereotyped <<DAMLproperty>> becomes a DAML property or one of its subclasses; this subclass is determined by additional contextual information.
<<DAMLdatatype>> Class	Datatype	UML classes stereotyped <<DAMLdatatype>> become DAML datatypes.
Data type	XML Schema or DAML datatype	UML data types are generally mapped into XML Schema or user-defined DAML datatypes.
Associations and roles	Object property and restriction	UML associations and roles map to object properties, and restrictions.
Attributes	Datatype property	Attributes map to datatype properties and restrictions. Attributes can only have datatypes as their range.
Generalization	subClassOf or subPropertyOf	The mapping of a UML generalization depends on the stereotype of the UML classes involved.
Import	Import	Package imports map directly to ontology imports.
Fully qualified names	Namespaces	Associations, roles, and classes from different packages are referenced using their fully qualified names; the package component of these names is mapped into XML namespaces and imported ontologies.

Comparing this tool with other related approaches discussed in this book, we have found that it has limitations similar to those that we mentioned when discussing Cranefield's software (XSLTs). Specifically, the

tool uses only standard UML and does not provide a convenient solution for representing properties, their relations, advanced class restrictions, etc. In fact, the implementation of the tool is even more limited than that of Cranefield's software, since it is oriented exclusively toward IBM Rational Rose, whereas Cranefield's XSLT is applicable to any UML XMI document and is independent of UML tools.

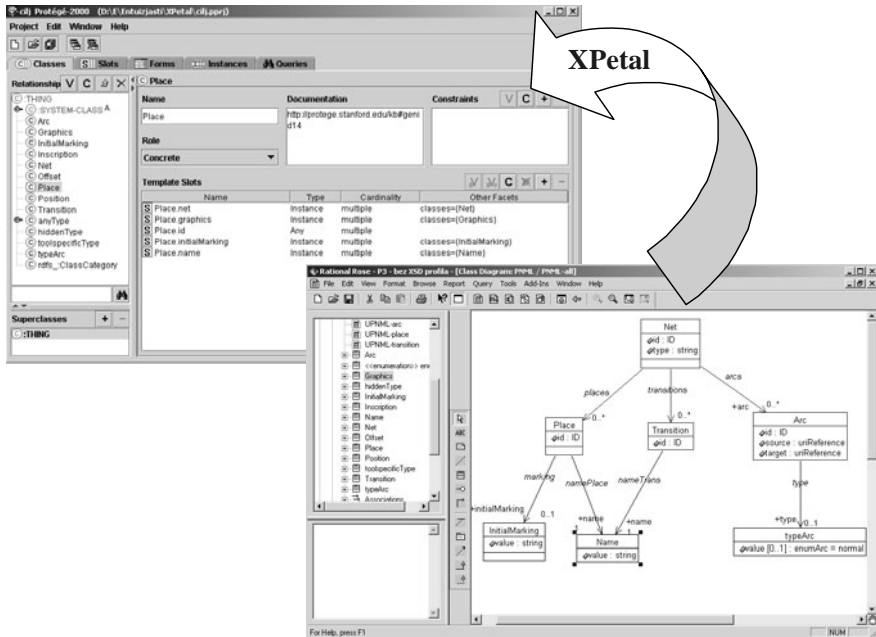


Fig. 6-13. Using XPetal to transform a UML model developed in IBM Rational Rose into an RDF Schema document that is then imported into Protégé

6.2.4 Visual Ontology Modeler (VOM)

The Visual Ontology Modeler (VOM) is a result of a collaborative work between Sandpiper Software Inc. and the Knowledge System Laboratory at Stanford University. To the best of our knowledge, this tool is not freely available. The tool extends IBM Rational Rose and enables ontology development with user-friendly wizards that automate the creation of a logical model and related diagrams (see Fig. 6-14).

The tool is based upon a UML profile for ontology development that is closely related to Protégé's metamodel for ontologies and to Gruber's Frame ontology [Gruber, 1993]. Apart from the use of well-known UML-

based graphical notations for ontology development, VOM supports widely accepted Semantic Web ontology languages such as DAML+OIL and OWL.

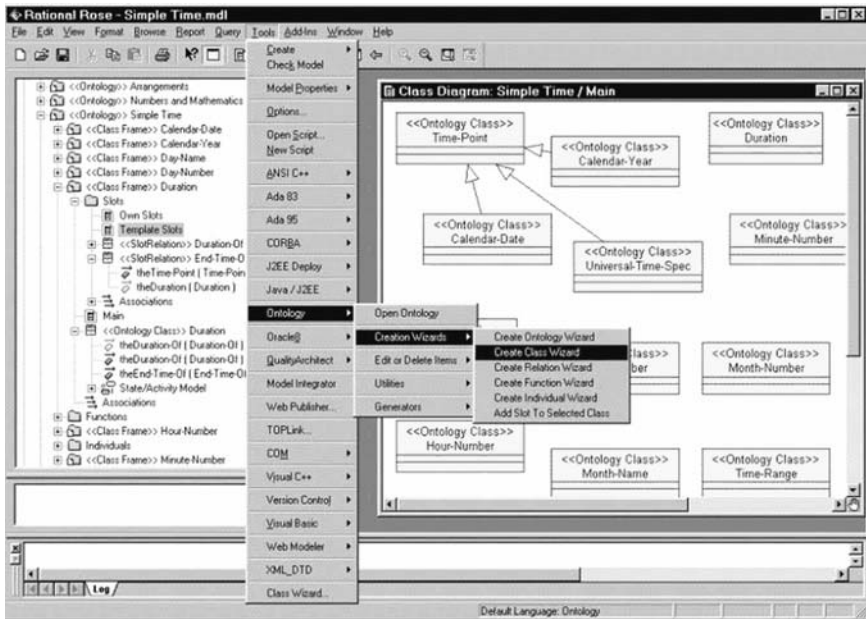


Fig. 6-14. The graphical user interface of the Visual Ontology Modeler (VOM), developed as an IBM Rational Rose add-in

Although this approach uses UML, we cannot say that is a real MDA-based approach. Rather, it can be described as an MDA-compatible approach, since it is based not on an ontology metamodel, but on a UML profile. Compatibility with the MDA is achieved through support for this UML profile and XMI. In fact, however, such an approach the use of advanced features of ontology languages (e.g., disjointWith, complementOf, and sameClassAs). The transformations between UML models and the supported ontology languages are built into the programming logic of the IBM Rational Rose add-in, so that only IBM Rational Rose can perform them.

6.3 Summary of Relations Between UML and Ontologies

In the course of various different approaches and tools for ontology development using software engineering techniques, we have tried to emphasize

the state-of-the-art solutions to this problem. The goal of this section is to provide a concise summary of all the tools and approaches described in detail in the previous two sections. We also aim to indicate the most important differences between the two modeling approaches (ontologies and MDA) from the point of view of the most relevant languages (UML, OWL, and DAML+OIL).

6.3.1 Summary of Approaches and Tools for Software Engineering-Based Ontology Development

Table 6-3 summarizes the approaches analyzed in Sect. 6.1 and the tools presented in Sect 6.2. Specifically, this table reviews the MDA-based formal definitions (i.e., metamodels) of the approaches and tools discussed, the kinds of model interchange format that they use (e.g., UML XMI), the proposals for implementing the mappings (e.g., XSLT), and the target ontology languages (e.g., OWL). Table 6-3 also indicates how each specific approach and tool implements the transformation between MDA-based languages and ontology languages. This information is relevant because it reveals whether a certain solution (i.e., software component) can be reused across different applications independently of the tool it was originally implemented for. For example, XSLT-based solutions and XPetal have potential for reuse. Finally, the table gives URLs for each approach or tool (if any exists) where one can find accompanying on-line resources.

6.3.2 Summary of Differences Between UML and Ontology Languages

Here, we aim to summarize the most relevant incompatibilities between MDA-based approaches and ontology languages. As we have already mentioned Baclawski and colleagues gave the most comprehensive list of incompatibilities [Baclawski et al., 2002a] when they proposed an extension of UML for the DAML language, and our summary mainly relies on that study. In Table 6-4, we present an outline of the incompatibilities between ontology languages and UML given in [Baclawski et al., 2002a].

Table 6-3. An overview of the present UML- and MDA-based ontology development approaches and tools, and of the techniques that they apply to support transformation to Semantic Web languages

Approach	Metamodel	MDA sharing format	Transformation mechanism	Target language
<i>CraneField</i> [CraneField, 2001a]	Standard UML	UML XMI	XSLT	RDFS, Java classes
<i>Baclawski et al.</i> [Baclawski et al., 2002a; Baclawski et al., 2002b]	Standard UML, UML profile, and MOF-based metamodel	(not given – UML XMI and MOF XMI can be used)	–	DAML
<i>Falkovych et al.</i> [Falkovych et al., 2003]	Standard UML	UML XMI	XSLT	DAML+OIL
<i>AIFB OWL DL</i> [Brockmans et al., 2004]	MOF-based metamodel and UML profile	(not given – UML XMI, and MOF XMI can be used)	–	OWL DL
<i>GOOD OLD AI ODM</i> [Djurić et al., 2005a; Djurić et al., 2005b]	MOF-based metamodel and UML profile	ODM XMI UML XMI	XSLT	OWL
	http://www.sfu.ca/~dgasevic/projects/UMLtoOWL/			
<i>Protégé</i>	Protégé metamodel	Protégé XMI	Programmed (built-in component)	OWL, RDF(S), DAML+OIL, XML, UML XMI, Protégé XMI, ...
	Standard UML	UML XMI		
	http://protege.stanford.edu			
<i>Visual Ontology Modeler (VOM)</i> [Ceccaroni & Kendall, 2003; Kendall et al., 2002]	UML profile	IBM Rational Rose	Programmed (built-in component)	OWL, DAML+OIL, RDFS
	http://www.sandsoft.com/products.html			
<i>DUET</i>	UML profile	IBM Rational Rose, ArgoUML	Programmed (built-in component)	DAML+OIL
	http://codip.grci.com/wwwlibrary/wwwlibrary/DUET_Docs/			
<i>Xpetal</i>	Standard UML	IBM Rational Rose mdl files	Programmed (reusable component)	RDFS
	http://www.langdale.com.au/styler/xpetal/			

Table 6-4. Summary of incompatibles between UML and ontology languages, based mainly on [Baclawski et al., 2002a]

Difference	Description
Monotonic and non-monotonic	Ontology languages are monotonic, whereas UML and object-oriented languages are nonmonotonic. A system is monotonic if adding new facts can never cause previous facts to be falsified.
Metalevels	Ontology languages do not have a rigid separation between metalevels. For example, in OWL Full, an instance of a class can be another class.
Specialization/generalization	The UML generalization/specialization relation stimulates behavioral reuse, while the idea behind of this relation in ontology languages is set-theoretical.
Modularity	Ontology languages do not have profiles, packages, or any other modularity mechanism supported by UML and object-oriented languages.
Containers and lists	UML packages are not at the same metalevel as RDF containers (Bag, Seq, and Alt), and the presentation features of packages also make them unsuitable for representing RDF containers.
Property	A property is a first-class (independent) modeling element in ontology languages, while the UML notion of an association/attribute is not a first-class concept.
Class constructors	Ontology languages have the ability to construct classes using boolean operations (union, intersection and complement) and quantifiers. In UML, there is no corresponding primitive.
Cardinality constraints	This incompatibility originates from the first-class nature of ontology properties. In ontology languages one can specify a cardinality constraint for every domain of a property all at once, whereas in UML this must be specified separately for each association (end) belonging to the property.
Subproperties	Ontology languages allow one property to be a subproperty of another. UML has the ability to specify that one association is a specialization of another, although this construct is rarely used. However, UML does not allow inheritance of class attributes.
Namespaces	While Semantic Web ontology languages use URIs to relate their constructs, UML has various ways of connecting its constructs (e.g., each class is a namespace of its attributes, which can be specified further as having a private, protected, or public scope).

One additional aspect is also very important when comparing UML and ontology languages. Most of the authors cited here refer to UML as a graphical (diagramming) notation [Cranfield, 2001b; Falkovych et al.,

2003]. This can be explained by the fact that UML was initially developed to serve as a graphical language for software modeling. However, the notion of UML models has evolved. UML models as defined in [Seidewitz, 2003] and [Selic, 2003] correspond semantically to the notion of ontologies defined in [Hendler, 2001]. Therefore, one must have in mind this new concept of UML models when considering how to bridge gaps between ontologies and MDA modeling spaces. Of course, the UML graphical notation is still useful as a standardized and well-known syntax and can be also beneficial for the graphical representation of ontologies.

6.3.3 Future Development

Although the lessons learned from the approaches analyzed above are rather useful, none of them gives an ultimate solution to the problem of the use of the MDA for ontology development. Having realized this fact, the OMG issued an RFP for adopting a standardized architecture called the Ontology Definition Metamodel [OMG ODM RFP, 2003]. This RFP specifies that the new OMG standards have to contain the following elements:

- a formal description of a new MDA-based ontology language called the Ontology Definition Metamodel;
- a related UML profile called the Ontology UML Profile; and
- the necessary transformations between the Ontology Definition Metamodel and the Ontology UML Profile, as well as transformations to contemporary Semantic Web languages (i.e., OWL).

We believe that such a set of standards (promoting usage of all MDA concepts) provides us with considerable benefits when defining a meta-modeling architecture and enables us to develop new MOF languages capable of expressing all concepts of ontology languages. In the following chapters, we describe in detail the present state of this OMG effort [OMG ODM RFP, 2005].

7. The MDA-Based Ontology Infrastructure

Ontologies and the Model-Driven Architecture (MDA) are two modeling approaches that are being developed in parallel, but by different communities. They have common points and issues and can be brought closer together. Many authors have attempted to bridge the gaps and have proposed several solutions. The result of these efforts is the recent OMG's initiative for defining an ontology development platform.

7.1 Motivation

To be widely adopted by users and to succeed in real-world applications, knowledge engineering and ontology modeling must catch up with trends in mainstream software. It must provide a good support in the form of software tools and integration with existing and upcoming software tools and applications must be eased, which will add value on both sides. To be employed in common applications, knowledge management by software must be taken out of laboratories and isolated high-tech applications and put closer to ordinary developers. This issue has been addressed in more detail in Cranefield's papers [Cranefield, 2001a] [Cranefield, 2001b].

With the development of the Semantic Web initiative, the importance of ontologies is increasing rapidly. Semantic Web researchers are trying to make ontology development and ontologies in general closer to software practitioners [Knublauch, 2004]. However, ontologies have more rigorous foundation closely related to well-known paradigms in AI (e.g. description logic, semantic networks, and frames). Thus, most of the current Semantic Web ontologies have been developed in AI laboratories. Accordingly, we need to answer some questions such as: How can we increase the number of ontology developers? How can we motivate software engineering practitioners to develop and use ontologies? Can we use software development tools to develop ontologies? Therefore, we need some ways to integrate software development and ontologies.

The integration of the ongoing software engineering efforts with the concept of the Semantic Web is not a new idea. Many researchers have

suggested using UML in order to solve this problem. However, UML is based upon an object-oriented paradigm, and has some limitations regarding ontology development. Hence, we can only use UML in the initial phases of ontology development. These limitations can be overcome using UML extensions (i.e. UML profiles) [Duddy, 2002], and other OMG standards, like the Model Driven Architecture. In addition, if we want to offer a solution consistent with the MDA proposals, we should also support the automatic generation of completely operational ontology definitions (e.g. in the OWL language) that are model driven [Selic, 2003]. Currently, the most important direction toward this goal is the one pursued by a dedicated research group within the OMG that is trying to achieve convergence of many different proposals for solutions to this problem [OMG ODM RFP, 2003]. The result of this effort should be a standard language (i.e. a metamodel) based on the MDA standards [Miller & Mukerji, 2003] and the W3C Web Ontology Language (OWL) recommendation [Bechhofer et al., 2004].

7.2 Overview

MDA and its four-layer architecture provide a solid basis for defining metamodels of any modeling language, so it is a straightforward choice to define an ontology-modeling language in the MOF. Such a language can utilize the MDA's support in modeling tools, model management and interoperability with other MOF-defined metamodels. The present software tools do not implement many of the concepts that are the basis of the MDA. However, most of these tools, which are mostly oriented toward the UML and M1 layer, are expected to be enhanced in the next few years to support the MDA.

Currently, there is an RFP (Request for Proposal) within the OMG that is aimed at defining a suitable language for modeling Semantic Web ontology languages in the context of the MDA [OMG ODM RFP, 2003]. The authors of this book have made a proposal for such an architecture in accordance with this RFP. In this approach to ontology modeling within the scope of the MDA, which is shown in Fig. 7-1, several specifications need to be defined:

- the Ontology Definition Metamodel (ODM)
- the Ontology UML Profile – a UML Profile that supports UML notation for ontology definition

- and the two-way mappings between: OWL and the ODM, the ODM and other metamodels, the ODM and the Ontology UML Profile and from the Ontology UML Profile to other UML profiles.

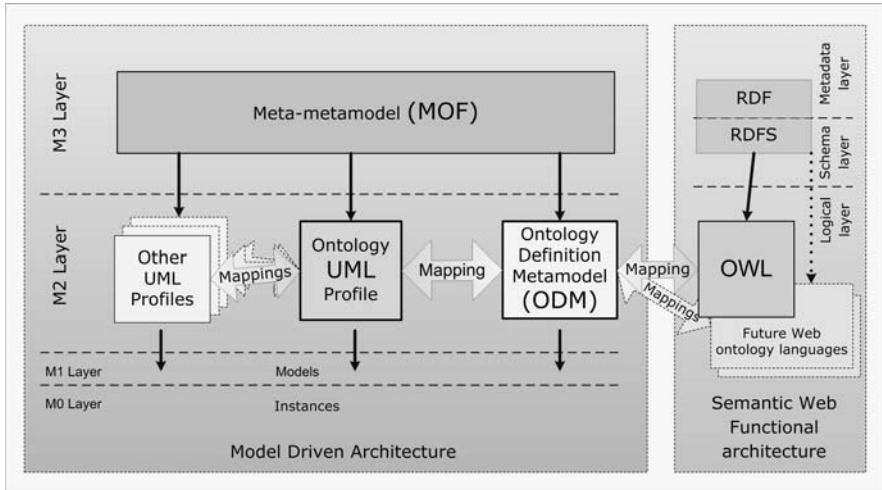


Fig. 7-1. Ontology modeling in the context of the MDA and the Semantic Web

The Ontology Definition Metamodel (ODM) should be designed to include the common concepts of ontologies. A good starting point for the construction of the ODM is OWL since that is the result of the evolution of existing ontology representation languages, and is a W3C recommendation. It is in the logical layer of the Semantic Web [Berners-Lee et al., 1999], on top of RDF Schema (in the schema layer).

In order to make use of the graphical modeling capabilities of UML, the ODM should have a corresponding UML profile [Sigel, 2001]. This profile will enable graphical editing of ontologies using UML diagrams as well as provide other benefits of using mature UML CASE tools.

Both UML models and ODM models are serialized in XMI format so the two-way transformation between them can be done using XSL transformation. OWL also has a representation in the XML format, so another pair of XSL Transformations should be provided for two-way mapping between the ODM and OWL. For mapping from the ODM into other metamodels or from the Ontology UML Profile into other, technology-specific UML Profiles, additional transformations can be added to support usage of ontologies in the design of other domains and vice versa.

Several official ([ODM DSTC, 2003; ODM Gentleware, 2003; ODM IBM, 2003; ODM Sandpiper&KSL, 2003]) and unofficial ([Djurić et al.,

2005a; Djurić, 2004]) proposals followed the RFP. The official proposals, proposed by IBM, Gentleware, etc. were joined together into a common submission [OMG ODM, 2004]. A revised joint submission followed in 2005 [OMG ODM RFP, 2005]. The discussion of the ODM in this book is based mostly on the most recent, third revised submission [OMG ODM RFP, 2005], and the discussion of the UML profile is based mostly on [Djurić, 2004] and [Djurić et al., 2005c].

7.3 Bridging RDF(S) and MOF

Before we start on a more detailed description of the ODM, we must clarify the differences between metamodeling in the Semantic Web world, which is based on RDFS constructs, and in the object-oriented MDA, which is based on the MOF. Obviously, if we want to perform a transformation from the RDFS MS to MOF MS, we need to make transformation rules, which have to determine which target concept (defined in MOF) we should obtain from a source concept (defined in RDFS). The main task is to identify the most important differences and similarities between the main constructs in both spaces and decide how to overcome those differences. These concepts are briefly compared in Table 7-1, which groups the most similar concepts from both spaces.

Both good and bad news can be derived from this comparison. The good news is that there is a significant similarity: they are both a sort of entity-relationship based world. In RDFS, `rdfs:Class` is a sort of entity, while `rdf:Property` is a sort of relation. In the MOF, `Classifier` represents an entity that is related to other entities via associations or attributes. Basically, a concept (at level M2 in the RDFS MS) that is modeled as an `rdfs:Class` (at level M3 in the RDFS space) becomes concept (at level M2 in MOF space) that is a MOF Class (at level M3 in MOF space), a concept modeled as an `rdfs:Property` becomes an MOF Association or an MOF Attribute and so on. For Example, `owl:Class` (M2) is defined as an `rdfs:Class` (M3) in RDFS. In the MOF, it will be an `OWLClass`, an MOF Class. `rdf:Property` is also an `rdfs:Class`, so in the MOF MS it becomes an MOF Class `RDFProperty`. An example of an `rdf:Property` is `rdfs:subclassOf`; it should be transformed to an MOF Association `RDFSsubclassOf`.

Table 7-1. A brief description of the basic MOF and RDF(S) metamodeling concepts

MOF element	Short description	RDF(S) element	Short description
Element	Element classifies the elementary, atomic constructs of models. It is the root element within the MOF Model.	rdfs:Resource	Represents all things described by RDF. The root construct of majority of RDF constructs.
DataType	Models primitive data, external types, etc.	rdfs:Datatype	Mechanism for grouping primitive data.
Class	Defines a classification over a set of object instances by defining the state and behavior they exhibit.	rdfs:Class	Provides an abstraction mechanism for grouping similar resources.
Classifier	An abstract concept that defines classification. It is specialized by Class, DataType, etc.		In RDF(S), rdfs:Class also have a function that is similar to a MOF concept of Classifier.
Association	Expresses relationships in the metamodel between pairs of instances of Classes	rdf:Property	Defines a relation between subject resources and object resources.
Attribute	Defines a notional slot or value holder, typically in each instance of its Class.		
TypedElement	A TypedElement is an element that requires a type as part of its definition. A TypedElement does not itself define a type, but is associated with a Classifier. Examples are object instances, data values, etc.		In RDF(S), any rdfs:Resource can be typed (via the rdf:type property) by some rdfs:Class

The bad news is that corresponding concepts (rdfs:Class and Classifier and rdf:Property and Association or Attribute) have different natures. The concept of Class in RDFS (rdfs:Class) is not completely identical to the concept of Class that is defined in UML and the MOF. Every rdfs:Class is a set of resources, called a class extension. These resources are instances

of that class. Two classes can have the same class extension but still be different classes. Classes in RDFS are set-theoretic, while traditional object-oriented classes are more behavioral. Unlike an object-oriented class, an `rdfs:Class` does not directly define attributes or relations to other resources, and there is no concept similar to that of methods. Attributes and relations are defined as properties.

In RDF, a property is a concept that represents a relation between a subject resource and an object resource. Therefore, it might look similar to the concept of attribute or association in the traditional, object oriented sense. However, the important difference is that `rdf:Property` is a stand-alone concept; it does not depend on any class or resource as the associations or attributes are in the MOF. In ontology languages, a property can be defined even with no classes associated with it. That is why a property can not be represented as an ordinary association or attribute, its closest object-oriented relatives.

7.4 Design Rationale for the Ontology UML Profile

In order to customize UML for modeling ontologies, a UML profile for ontology representation, called the Ontology UML Profile, must be defined.

In developing our Ontology UML Profile we used thhhe experience of other UML profile designers (e.g., see [Juerjens, 2003]). Applying this experience to our case, we wanted our Ontology UML Profile to:

- offer stereotypes and tags for all recurring ontology design elements, such as classes, individuals, properties, complements, unions, and the like;
- make specific ontology-modeling and design elements easy to represent in UML diagrams produced by standard CASE tools, thus keeping track of ontological information in UML models;
- enable ontological knowledge to be encapsulated in an easy-to-read format and offer it to software engineers;
- make it possible to evaluate UML diagrams of ontologies and indicate possible inconsistencies;
- support Ontology Definition Metamodel, hence to be able to represent all ODM concepts.

Currently, several different approaches to ontology representation in UML have been proposed. We note two major trends among them:

- Extending UML with new constructs to support specific concepts of ontologies (Property for example) [Baclawski et al., 2002a]. This approach is outdated.
- Using standard UML and defining a UML profile for ontology representation [Baclawski et al., 2002a]. This approach is followed in all recently proposed solutions.

We believe that ontology representation in UML can be achieved without nonstandard UML extensions, and hence our approach belongs to the latter of the above two trends. In the Ontology UML Profile presented in this book, specific concepts of ontologies are annotated using the standard UML extension mechanisms described above. Models created with such a UML profile are supported by standard UML tools, since they do not add nonstandard concepts to UML, and thus they are UML models.

The definition of a UML profile in the context of the MDA four-layer metamodeling architecture means extending UML at the metamodel layer (M2). One can understand these extensions as a new language, but also understand UML as a family of languages [Duddy, 2002]. Each of these languages uses UML notation with the four UML extension mechanisms. Recent UML specifications enable the use of graphical notation for specifying stereotypes and tagged definitions [Kobryn, 2001]. Thus, all stereotypes and tagged values that are defined in this book can be shown in this way.

The notation used for creation of stereotype in Ontology UML Profile («OWLClass» stereotype) accommodates UML's Class («metaclass»). Having this graphical notation for the UML extension mechanism can be useful for explaining certain relations between UML constructs and new stereotypes, and also between stereotypes themselves.

Since stereotypes are the principal UML extension mechanism, one might be tempted to think that defining Ontology UML Profile is a matter of specifying few stereotypes and using them carefully in a coherent manner. In reality, however, it is much more complicated than that. The reason is that there are a number of fine details to take care of, as well as the existence of some conceptual inconsistencies between the MDA and UML that may call for alternative design decisions.

8. The Ontology Definition Metamodel (ODM)

There were four separate proposals for the ODM in response to the OMG's ODM RFP [OMG ODM RFP, 2003] submitted by the following OMG members: IBM [ODM IBM, 2003], Genteware [ODM Genteware, 2003], DSTC [ODM DSTC, 2003], and Sandpiper Software Inc and KSL [ODM Sandpiper&KSL, 2003]. However, none of those submissions made a comprehensive proposal. For example, none of them proposed XMI bindings for the ODM, none of them proposed mappings between the ODM and OWL, and only IBM [ODM IBM, 2003] and Genteware [ODM Genteware, 2003] proposed an Ontology UML profile. Accordingly, the OMG partners decided to join their efforts, and the current result of their efforts together, is the ODM joint submission [OMG ODM, 2004]. A revised joint submission, which is the latest in the time of writing, followed in August 2005 [OMG ODM RFP, 2005]. In this chapter we introduce the main concepts of that common initiative.

8.1 ODM Metamodels

To create a joint ODM, the submitters decided to organize it as a composition of several metamodels. We show the architecture of the current ODM submission (third revised submission) in Fig. 8-1.

In order to support well-known Semantic Web ontology languages, the ODM has two separate metamodels, namely metamodels for OWL and RDFS. These languages are W3C standards which form a basis for the Semantic Web, thus are the central part of the ODM. Other metamodels have two-way mappings to and from RDFS/OWL. This means that they are also used as a mediator between yet other metamodels.

Since the ODM needs an expressive logic language that can be used on the Semantic Web to describe string-based expressions, the ODM contains the Common Language (CL) metamodel. Apart from W3C ontology languages, the ODM supports other ontology language standards (ISO standards) by defining a Topic Maps (TM) metamodel. The central metamodel in the joint submission (the one from 2004) was a Description

Logics (DL) metamodel since most of the current ontology languages (e.g. OWL) are based upon some of the DL classes. The role of this metamodel was to mediate all metamodels defined in the ODM. However, the next submission (2005.) moved this metamodel somewhat to the side, leaving it as a nonnormative metamodel, while the central role in the ODM is now played by RDFS and OWL metamodels. Finally, to provide connection with many existing systems based on databases, the ODM defines an Entity Relationship (ER) metamodel.

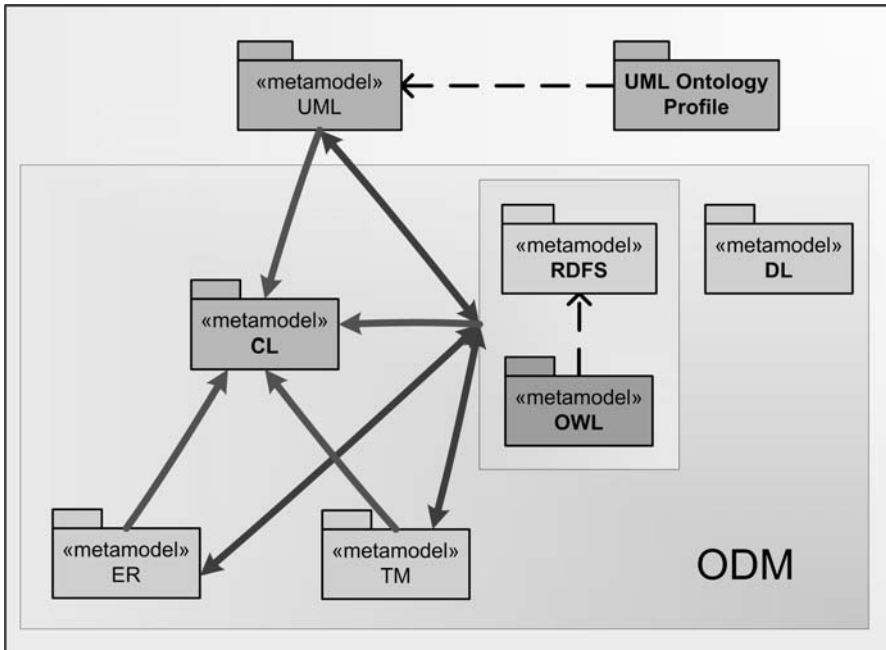


Fig. 8-1. The ODM Metamodels

Taking into account the importance of UML, the ODM developers want to create mappings between the standard UML metamodel and the RDFS and OWL metamodels, and hence establish a way to employ present UML models in ontology development. Furthermore, these developers define Ontology UML Profile (OUP) using standard UML extension mechanisms. The purpose of OUP is to enable the use of the standard UML graphical notation for developing ontologies.

8.2 A Few Issues Regarding the Revised Joint Submission

Before we start a more detailed explanation of the OMG's Ontology Definition Metamodel, we shall briefly discuss a few issues that might be interesting in relation to the upcoming version 2 of the MOF standard.

The current ODM proposal was tailored with respect to the ECore metamodel, the heart of Eclipse Modeling Framework (EMF). While ECore is similar to the MOF (particularly EMOF) there are differences. ECore is much simpler and is targeted toward implementation in CASE tools. Therefore, there might be some minor inconsistencies in the ODM with version 2 of the MOF standard. As the ODM and MOF version 2 are still works in progress, some of these issues, if not all of them are probably going to be resolved in the final version. In this section, we shall mention them briefly. Note that we have done some minor changes to the version of the ODM presented in this book regarding these issues, but those changes do not have any impact on the essence of the discussion. Many of these minor flaws that we are going to mention probably arise from the fact that there are many tools that operate with MOF standards, and the proposals have been constructed using CASE tools (EMF for example), which all have their non-standard way of handling some things.

There are many associations having the same name in the metamodel proposal that reside in the same package. An example of such a situation is that two different associations in the DL metamodel both have the same name – contains. However, MOF specification clearly states that the name of any element within the same namespace must be unique. A solution could be found by renaming one of these associations, or by refactoring by extracting the class. The first solution seems less intrusive for this discussion, but the second is more elegant from the point of view of software engineering.

Many associations were left unnamed in the ODM joint submission. The MOF specification states that every Type must be named, and Association inherits Type, so this condition must be fulfilled. Also note that the UML2 Infrastructure specification allows unnamed Types but EMOF adds a constraint that every Type must be named. In this discussion, we have added names to associations, but this problem can be also solved with CASE tools, which can generate random names. However, random names are not convenient if a human has to work with them. Would it be pleasant to have to work with associations named “mnrt90490789” or “FF0AD3458B0BB”?

Associations model relationships between classes. If classes are subjects and objects, associations should represent verbs. Thus, their names should be verbs. A good rule would be that the ODM should name associations after the names of the properties that are their counterparts in RDF(S), even if they are not verbs. In our opinion, if an association does not have a direct RDFS counterpart, a verb should be used for its name. In the joint submission, most of the names of properties were used for naming association ends, not associations.

8.3 The Resource Description Framework Schema (RDFS) metamodel

Resource is one of the basic RDFS concepts; it represents all things described by RDF and OWL. It may represent anything on the Web: a Web site, a Web page, a part of a Web page, or some other object named by a URI. Compared with the concepts of ontologies, it could be viewed as a root concept, the Thing. In the RDFS MS, `rdfs:Resource` is defined as an instance of `rdfs:Class`. Since we use the MOF as a meta-metamodeling language, this concept will be defined as an instance of MOF Class named `RDFSResource`. This is the root class of most of the other concepts from RDFS and OWL metamodels that will be described: `RDFSClass`, `RDFSProperty`, `RDFStatement` etc. The hierarchy of concepts in the RDFS metamodel is shown as a class diagram in Fig. 8-2.

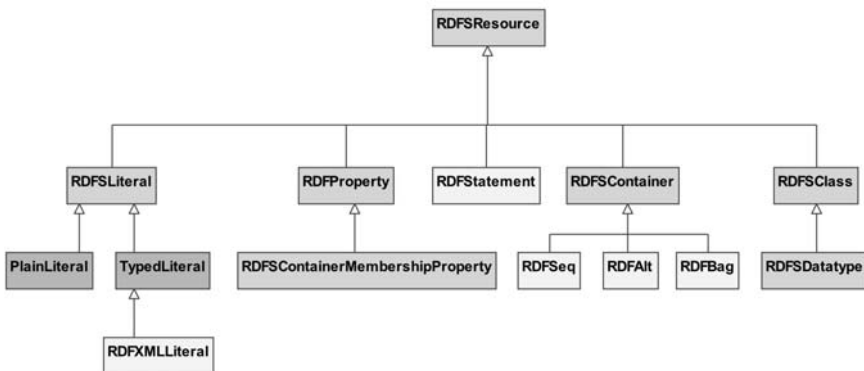


Fig. 8-2. RDFS metamodel - hierarchy of concepts

Other class diagrams depict these concepts in more detail. Note that, since the RDFS is self-defined and `rdfs:Class` inherits `rdfs:Resource`, in the RDFS MS `rdfs:Resource` is an instance of itself. However, in the MOF

MS, RDFSResource is an instance of MOF Class, not RDFS Resource, because the MOF is a meta-metamodel, not RDFS.

Among shown concepts, the most important ones are

- RDFSResource,
- RDFSClass,
- RDFProperty, and
- RDFStatement.

These are the basis for forming subject-predicate-object triples, such as “this section—describes—RDFS metamodel”.

Figure 8-3 shows the basic characteristics of the RDFSResource and RDFSClass concepts. RDFSResource has three attributes, all of String type, which are primarily intended to identify an instance of RDFSResource (and all concepts that inherit it):

- `localName`, for the name of the resource, unique within a namespace,
- `namespace`, for a namespace in which a resource resides and for grouping similar resources,
- `uri`, (unique resource identifier) which may be constructed from the namespace and `localName` and vice versa.

The RDFS specification also defines that an `rdfs:Resource` can have comments and labels attached. As these characteristics are described in RDFS MS using `rdf:Property` instances, namely `rdf:comment` and `rdf:label`, in MOF MS we model them using MOF Association concept. On the other end of these associations is `RDFSLiteral`, a concept that is used to represent simple data, such as numbers, text etc. Therefore, using these two associations, we can add various textual, numeric or other comments and labels to our data. One special kind of `RDFSLiteral` is `RDFXMLLiteral`, which represents XML textual data.

RDFS also defines a concept that is used for grouping similar resources – `rdfs:Class`. In an MOF-based metamodel, it can be represented using the MOF Class named `RDFSClass`, which, following the specification, inherits `RDFSResource`. This is determined by `RDFtype` (`rdf:type` property from RDFS MS). An `RDFSResource` can have many types, and multiple `RDFSResource` instances can have the same type. `RDFSsubClassOf` models inheritance in the M1 layer, ontological inheritance, among various instances of `RDFSClass`. An `RDFSClass` instance can inherit many other `RDFSClass` instances and can be inherited by many other `RDFSClass` instances, forming complex inheritance hierarchies. Do not confuse

RDFSsubClassOf with generalization between RDFSClass and RDFSResource. It is the same property in RDFS MS because RDFS is both metamodel and meta-metamodel, but in MOF MS, we use generalization association to model inheritance in the M2 layer, not RDFSsubClassOf.

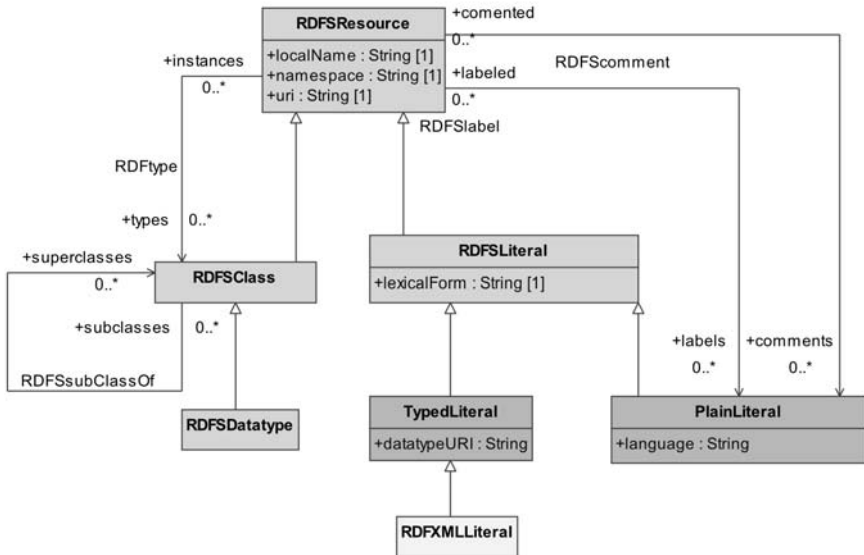


Fig. 8-3. RDFSClass and RDFSResource

The next important concept in the RDFS metamodel that needs to be explained is the RDFStatement, shown in Fig. 8-4. A statement is a subject-predicate-object triple that expresses some fact in a way similar to the way facts are expressed in the English language. The fact that Bob Marley was Jamaican, is expressed through a statement, whose subject is “Bob Marley”, whose predicate is “was”, and whose object is “Jamaican”. Following the definition of rdf:Statement definition from the RDFS MS, in the MOF MS, a statement is modeled as an MOF Class RDFStatement. The subject, predicate and object of an RDFStatement are determined using the associations RDFsubject, RDFobject and RDFpredicate, which are the counterparts in the MOF MS of the properties rdf:subject, rdf:object and rdf:predicate.

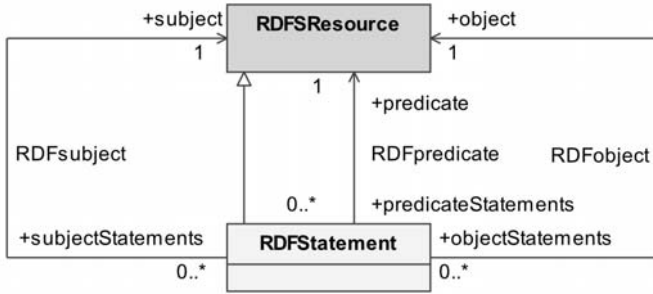


Fig. 8-4. RDFS Statement

From the definition of RDFStatement, we can see that all three associations, subject, object and predicate, link RDFStatement to RDFSResource. This design is wide open to any kinds of statement, even one that does not mean anything, for example “Bob Marley”-“Cuba”-“Jamaican”. Therefore, the predicate should usually be a resource that represents a verb, such as “be” or “memorize”, or some characteristic of a resource, like “name”. In the RDFS MS, such a resource is an rdf:Property, a concept that represents a type of relationship between resources.

in the M2 layer, RDFS defines two properties, rdfs:domain and rdfs:range that connect an rdf:Property with an rdfs:Class, making it possible to distinguish various types of relations between various types of resources in the M1 layer. In the MOF MS, RDFS metamodel represents a property using MOF Class named RDFProperty, a descendant of RDFSResource, as shown in Fig. 8-5.

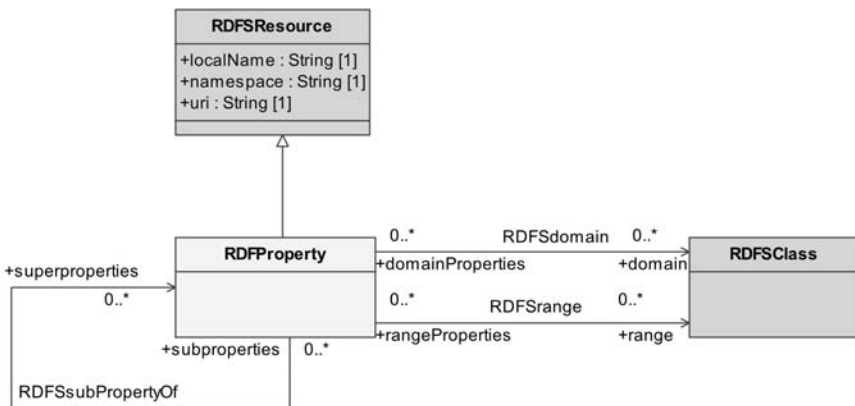


Fig. 8-5. RDFProperty

RDFSdomain and RDFSrange associations represent corresponding `rdfs:domain` and `rdfs:range` relations in the RDFS MS. RDFSdomain determines which types of resources can be at the “source” end of a relation and RDFSrange determines the type of the “destination” end. Another difference between properties and their object-oriented counterparts is that properties can form complex hierarchies, just as classes do. This is modeled by RDFSsubPropertyOf association, a counterpart in the MOF MS of `rdfs:subPropertyOf` in the RDFS MS.

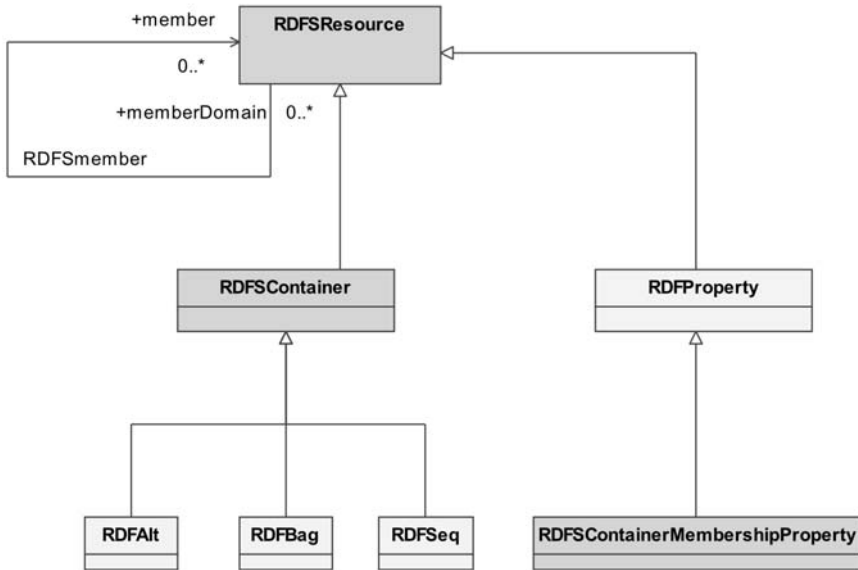


Fig. 8-6. RDFS Containers

There is often a need to group resources not by type, but by some arbitrary similarities. For example, we could make a group from all authors of this paper, or all basketball teams that are members of the NBA league. For such cases, RDFS uses `rdfs:Container`. In the MOF MS, we model this as an MOF Class `RDFSContainer`, which inherits `RDFSResource` (Fig. 8-6). `RDFSContainer` also has a few descendants defined: `RDFSBag` (a group of unordered resources), `RDFAlt` (for resources that are alternative to each other) and `RDFSeq` (a group of resources in which order is important). `RDFSmember` association models the containment of one `RDFSResource` in another `RDFSResource`.

RDF containers are open in the sense that the core RDF specifications define no mechanism to state that there are no more members. The RDF

Collection vocabulary of classes and properties can describe a closed collection, i.e. one that can have no more members. A collection is represented as a list of items, a representation that will be familiar to those with experience of Lisp and similar programming languages. RDFList modeled in the MOF MS is shown in Fig. 8-7. An RDFfirst relation connects an RDFSList with its first element, which can be any RDFSResource. RDFSrest connects an RDFSList with a sublist containing other elements, recursively forming the order of all elements.

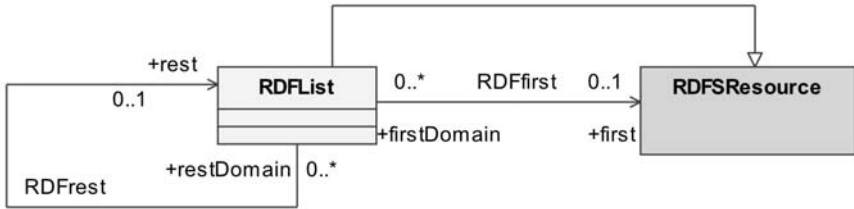


Fig. 8-7. RDFS Collections

In Fig. 8-8 we can see a few properties from RDFS modeled in the MOF MS as associations: RDFSseeAlso, RDFSisDefinedBy and RDFvalue. These properties are also known as “utilities”. RDFSseeAlso points to another resource that could be useful to look at. RDFSisDefinedBy points to another RDFSResource that defines the first one. RDFvalue association is used for describing structural values.

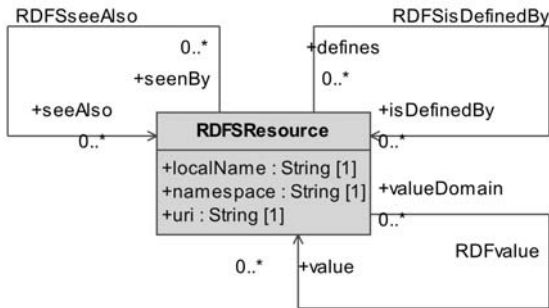


Fig. 8-8. RDFS Utilities

Ontology, shown in Fig. 8-9, is a concept that is not explicitly defined in RDFS (modeled in RDFS MS). Ontology is a concept similar to the concept of a Package in UML. It is a concept used for grouping other

concepts that belong to similar domains. Containment is denoted by “contains” association, which links one Ontology to one or more RDFSResource.

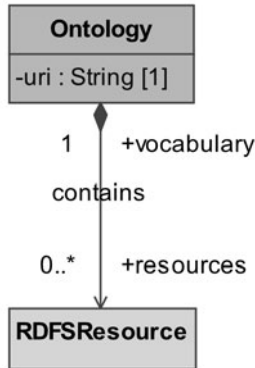


Fig. 8-9. The Ontology concept

Giving graphical diagrams here as an example of the use of an RDFS metamodel would not be easy, because RDF(S) does not have a standard graphic representation. A plain UML diagram given here would not represent concepts from RDFS metamodel, but concepts from UML metamodel, which are not the same thing. Fortunately, in the next section we shall define a UML profile for modeling RDFS concepts using a standard UML extension mechanism. In that section, we shall provide a few examples of RDFS and OWL ontologies.

8.4 The Web Ontology Language (OWL) Metamodel

The Web Ontology Language is built on top of RDF(S), using RDF(S) as both a meta-metamodel (M3) and a metamodel that is a base for extension (M2). In the MOF MS, the first dependency (the use of RDFS as a meta-metamodel) is replaced by using the MOF as a meta-metamodel. The second dependency means that the concepts of the OWL metamodel extend the concepts of the RDFS metamodel. Figure 8-10 shows the hierarchy of OWL concepts – we can see that most OWL concepts inherit the RDFS concepts RDFSResource, RDFProperty and RDFClass. RDFSClass is a base concept of the OWL concepts that represents classes (OWLClass, OWLRestriction and OWLDeprecatedClass), RDFProperty is

inherited by many concepts that represent properties in OWL (OWLObjectProperty, OWLDatatypeProperty and so on).

A small difference compared to OWL in the RDFS MS is that OWLOntology inherits Ontology, which is not defined in RDF(S). There are also some other concept in OWL metamodel that are not explicitly defined in the OWL language (RDFS MS), and their names are not prefixed with “OWL” in this metamodel.

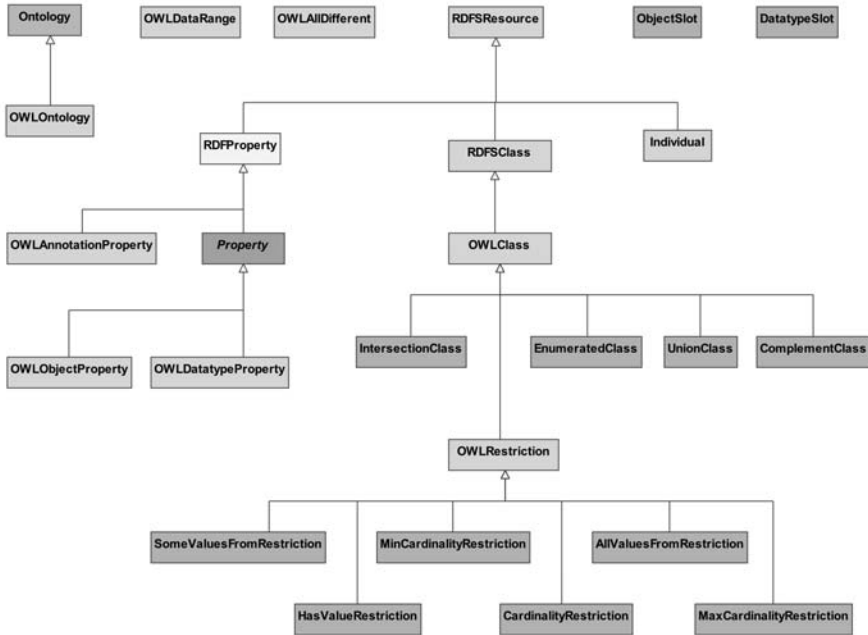


Fig. 8-10. The OWL Hierarchy

Classes provide an abstraction mechanism for grouping resources with similar characteristics. Like RDF classes, every OWL class is associated with a set of individuals, called the class extension. As OWL in many cases refines the concept of a class, it needs to model it separately, for example to inherit RDFSClass together with a new concept, OWLClass. Fig. 8-11 shows how OWLClass is modeled.

OWLClass inherits RDFSClass. Because OWL Full fully supports RDFS, we can also use the RDFSClass concept if we model using OWL Full. However, it is more convenient to use the OWLClass concept, because it is valid in OWL Lite and OWL DL also.

OWLClass is a set of individuals, which are modeled in OWL with the owl:Thing concept. Thanks to the fact that OWLClass extends RDFSClass,

it also extends the associations of RDFSClazz' from the RDFS metamodel. One of these associations is RDFtype, which enables various RDFSResources to state that some OWLClass is their type. Other associations in which RDFSClazz takes part, among which RDFSsubTypeOf is one of the most important, are also inherited.

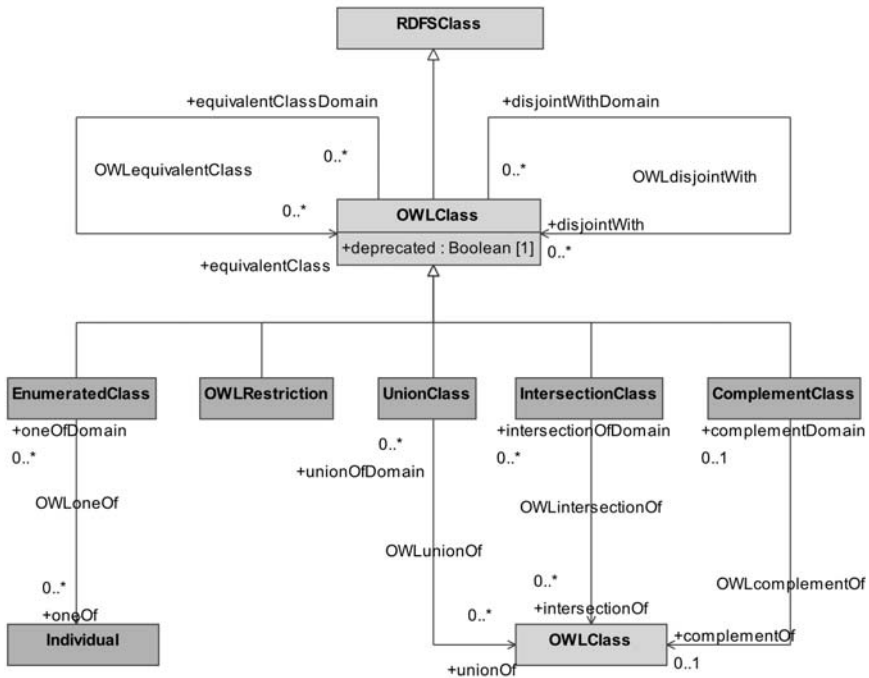


Fig. 8-11. OWLClass

Besides defining a class by name and connecting individuals using RDFtype association, classes can be defined in OWL in several other ways. Enumeration is defined by exhaustively enumerating its instances so OWLoneOf association is provided for building classes by enumeration. As a class can be constructed in OWL as a complement of another class or as a union or intersection of other classes, OWLcomplementOf, OWLunionOf, and OWLintersectionOf associations are provided. These associations and the multiplicities of their association ends are also shown in Fig. 8-11.

The OWL metamodel also defines two associations, which are counterparts of properties of the OWL language: OWLequivalentClass and OWLdisjointWith. OWLequivalentClass asserts that two classes have the

same class extension, and `OWLdisjointWith` asserts that two class extensions do not have any common individual.

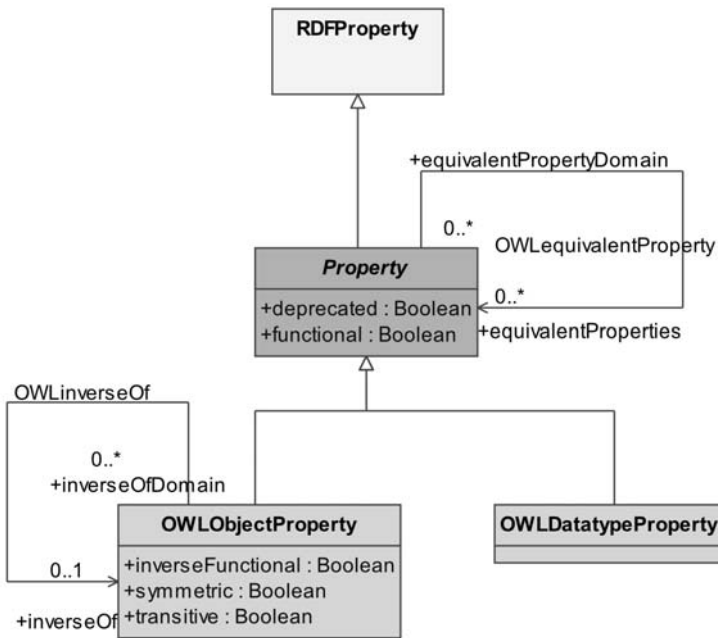


Fig. 8-12. OWL Properties

`OWLRestriction` is a special kind of `OWLClass`, and thus it inherits the `OWLClass` concept. It is not a “real” class, but a concept that enables constraints in OWL. Before we describe how restrictions are modeled, we should look at the details of the concepts that model properties in OWL, because `OWLRestriction` is a concept tightly connected to properties.

OWL refines the concept of `rdf:Property` by distinguishing two basic kinds of properties, `owl:ObjectProperty` and `owl:DatatypeProperty`. Recall from our discussion of the RDFS metamodel that `RDFProperty` has a domain and range that could both be `RDFSClass` and that `RDFSDataType` is a subclass of `RDFSClass`. This means that RDF does not distinguish relations between classes from relations between data types. The OWL metamodel, following the OWL language derived from the RDFS MS, introduces two distinct types of properties: `OWLObjectProperty` (a relationship between two `OWLClasses`) and `OWLDatatypeProperty` (a relationship between `OWLClass` and `RDFSDataType`).

The OWL metamodel in the joint submission includes the abstract MOF Class Property (see Fig. 8-12) in which is not included in the standard OWL language as a common superclass of OWLObjectProperty and OWLDatatypeProperty. This concept was introduced to solve the impedance mismatch between the RDFS and MOF concepts. In particular, MOF Association is dependent on the classes that are at its ends, and a class has to “know” which association ends it hosts. However, adding OWLequivalentProperty association directly to RDFProperty, as it is defined in OWL’s owl:equivalentProperty would imply a difference in the RDFS metamodel when it is used alone and when it is used as a base of the OWL metamodel.

Properties in OWL cannot have a Datatype as a domain, only as a range. That is why inverseOf association is applicable only to OWLObjectProperty. OWL also defines several other refined kinds of properties, namely owl:FunctionalProperty for relationships that have functional characteristics, owl:TransitiveProperty for transitive relationships, owl:SymmetricProperty for symmetric relationships and owl:InverseFunctionalProperty. These types of properties are represented in the MOF MS as a Property or OWLObjectProperty with suitable attributes with names analogous to those in the RDFS MS.

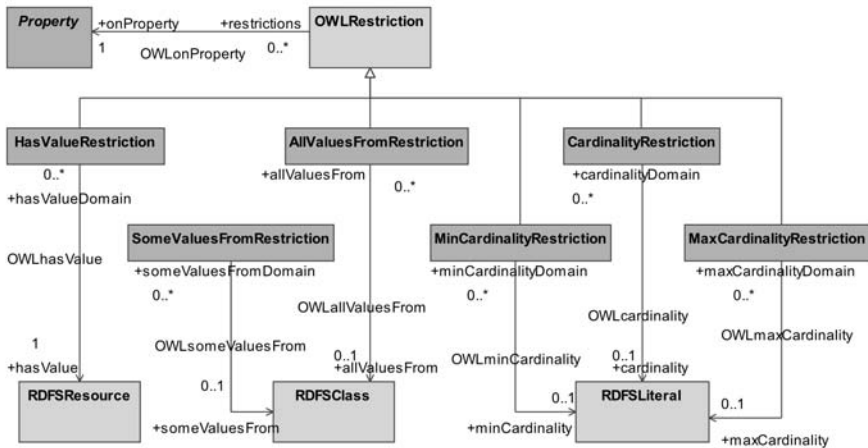


Fig. 8-13. OWLRestriction

OWLRestriction, shown in Fig. 8-13, is an anonymous class of all individuals that satisfy certain restrictions on their properties. Obviously, it is an MOF counterpart of the owl:Restriction concept from the RDFS MS. OWLonProperty association connects OWLRestriction and a property on

which that restriction is applied. There are two kinds of restrictions on properties: value constraints and cardinality constraints. A value constraint puts constraints on the range of the property when applied to this particular class description. Value constraints are modeled using `OWLhasValue`, `OWLsomeValuesFrom` and `OWLallValuesFrom` associations. A cardinality constraint puts constraints on the number of values a property can take, in the context of this particular class description. Cardinality constraints are modeled using `OWLminCardinality`, `OWLcardinality` and `OWLmaxCardinality` associations.

We have mentioned that OWL makes a significant distinction between named individuals (“objects”), and plain data values. In the OWL metamodel, individuals are modeled using the `Individual` concept, which is shown in Fig. 8-14. As `Individual` inherits `RDFSResource`, it also inherits all its associations with other concepts, of which one of the most important is `RDFtype`, which connects `RDFSResource` with its type, `RDFSClass`. In the case of `Individual`, `RDFtype` association has a constraint that the other end has to be an `OWLClass`. OWL also introduces properties that are used to state that some individuals are the same as (`owl:sameAs`) or different from (`owl:differentFrom`) others. In the OWL metamodel these properties are modeled as the associations `OWLsameAs` and `OWLdifferentFrom`.

Connecting many instances of `Individual` to a number of individual `owl:differentFrom` (or `OWLdifferentFrom` in the MOF MS) connections would overcrowd the model. That is why OWL introduces the `owl:AllDifferent` class and `owl:distinctMembers` property to connect that class with all individuals that are different from each other. In the OWL metamodel, this class and property are modeled as the `OWLAllDifferent` MOF Class and `OWLdistinctMembers` MOF Association.

A data range represents a range of data values. It can be either a datatype or a set of data values. Data ranges are used to specify a range of datatype properties. They are modeled as the `OWLDataRange` MOF Class (see Fig. 8-15), which is a descendant of `RDFSDatatype`, connected with `RDFSLiteral` via the `OWLoneOf` association.

OWL groups similar concepts together in an ontology using the `owl:Ontology` concept. This concept does not extend any other concept explicitly, but as RDFS specifies that everything can be a resource, it is an `owl:Ontology` as well. However, joint submission defines `OWLOntology` as a descendant of `Ontology`, which is not an `RDFSResource`. You can see the details of `OWLOntology` and its associations in Fig. 8-16. It includes `OWLimports` association, which enables the use of data from other ontologies in an ontology that can import them. Other associations, `OWLbackwardCompatibleWith`, `OWLIncompatibleWith`, and

OWLpriorVersion are intended to support different versions of an ontology.

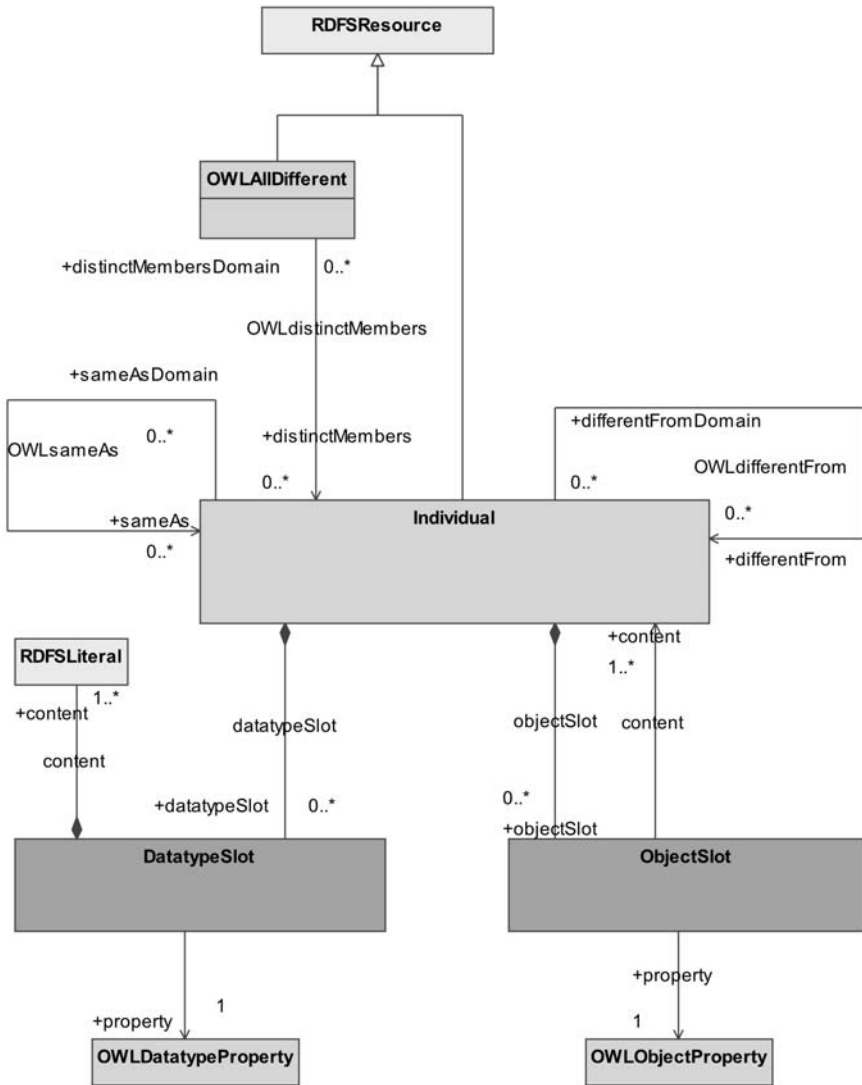


Fig. 8-14. Individuals

An OWLversionInfo statement generally has a string giving information about the version as its object, for example RCS/CVS keywords. An OWLpriorVersion statement contains a reference to some other ontology.

This identifies the specified ontology as a prior version of the containing ontology. An `OWLbackwardCompatibleWith` statement also contains a reference to another ontology. This identifies the specified ontology as a prior version of the containing ontology, and indicates further that it is backward compatible with the containing ontology. In particular, this indicates that all identifiers from the previous version have the same intended interpretations in the new version. An `OWLIncompatibleWith` statement contains a reference to another ontology. This indicates that the containing ontology is a later version of the referenced ontology, but is not backward compatible with it.

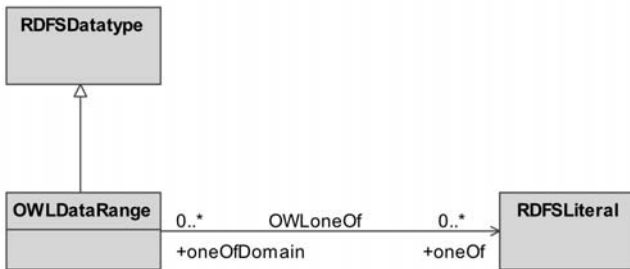


Fig. 8-15. OWLDataRange

Figure 8-17 shows the OWL Utilities – concepts that are not part of the base of OWL but are intended for version control, deprecated concepts, etc. Deprecation is a feature commonly used in versioning software (for example, see the Java programming language) to indicate that a particular feature has been preserved for backward-compatibility purposes, but may be phased out in the future. Deprecating a term, means that the term should not be used in new documents that commit to the ontology. This allows an ontology to maintain backward-compatibility while phasing out an old vocabulary. In the OWL language both classes and properties can be deprecated. OWL DL allows annotations on classes, properties, individuals and ontology headers. That is why `OWLAnnotationProperty` is included in the OWL metamodel. An example of annotation property is the `dc:creator` property defined in the DC vocabulary (<http://dublincore.org/>) if one wants to use it to annotate who the creator of an OWL ontology concept is.

This support for annotations, ontology headers, importing, and version information in the OWL metamodel is the first comprehensive attempt to cover all those features in the OWL language. In previous similar solutions, only IBM's submission to the ODM RFP [ODM IBM, 2003]

had considered such features, and hence that submission was used as a basis for the present OWL metamodel.

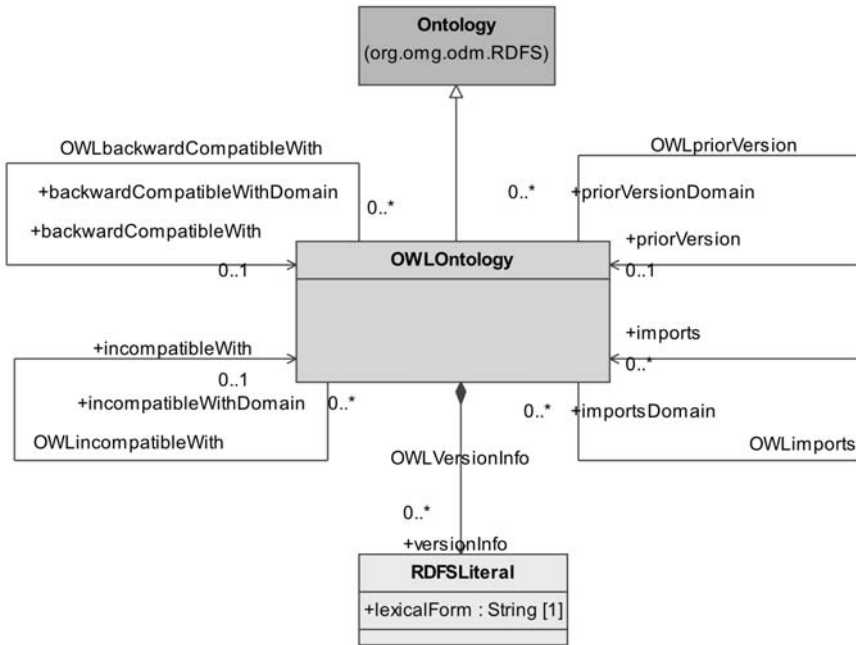


Fig. 8-16. OWL Ontology

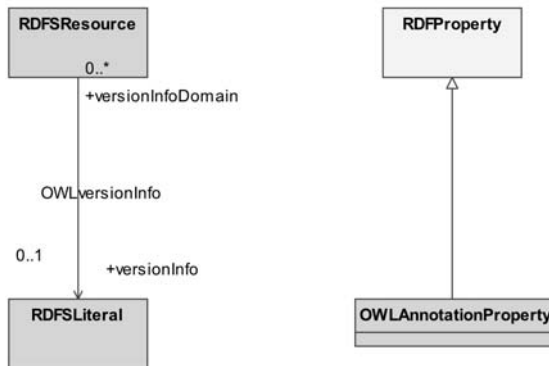


Fig. 8-17. OWL Utilities

The OWL metamodel together with the RDFS metamodel forms the basis for ontology development in the Model Driven Architecture. The main advantage is that it is compatible with the Web Ontology Language, which is one of the key technologies of the Semantic Web.

9. The Ontology UML Profile

UML profile is a concept used for adapting the basic UML constructs to a specific purpose. Essentially, this means introducing new kinds of modeling elements by extending the basic ones, and adding the new elements to the modeler's repertoire of tools. Also, free-form information can be attached to the new modeling elements. The Ontology UML Profile extends UML in a standard way to enable ontology modeling in the widely used UML modeling tools.

9.1 Classes and Individuals in Ontologies

The Class is one of the most fundamental concepts in the ODM and the Ontology UML Profile. As we noted in the discussion about the essential concepts of the ODM, there are some differences between the traditional UML Class or the concept of a Class in object-oriented programming languages and an ontology class as it is defined in OWL (`owl:Class`). Fortunately, we are not trying to adopt UML as a stand-alone ontology language, since that might require changes to the basic concepts of UML (Class and others). We only need to customize UML as a support to the ODM.

In the ODM, the concepts that represent classes, i.e. `RDFSClass`, `OWLClass`, `AllDifferent` and `Restriction` are modeled using the MOF Class concept. These constructs in the Ontology UML Profile are inherited from the UML concept that is most similar to them, UML Class. However, we must explicitly specify that they are not the same as UML Class, which we can do using UML stereotypes. An example of Classes modeled in the Ontology UML Profile is shown in Fig. 9-1.

`RDFSClass` and `OWLClass`, ontology classes identified by a class identifier have the stereotype `«RDFSClass»` or `«OWLClass»`, `OWLAllDifferent` has the stereotype `«OWLAllDifferent»` and `OWLRestriction` has the stereotype `«OWLRestriction»`.

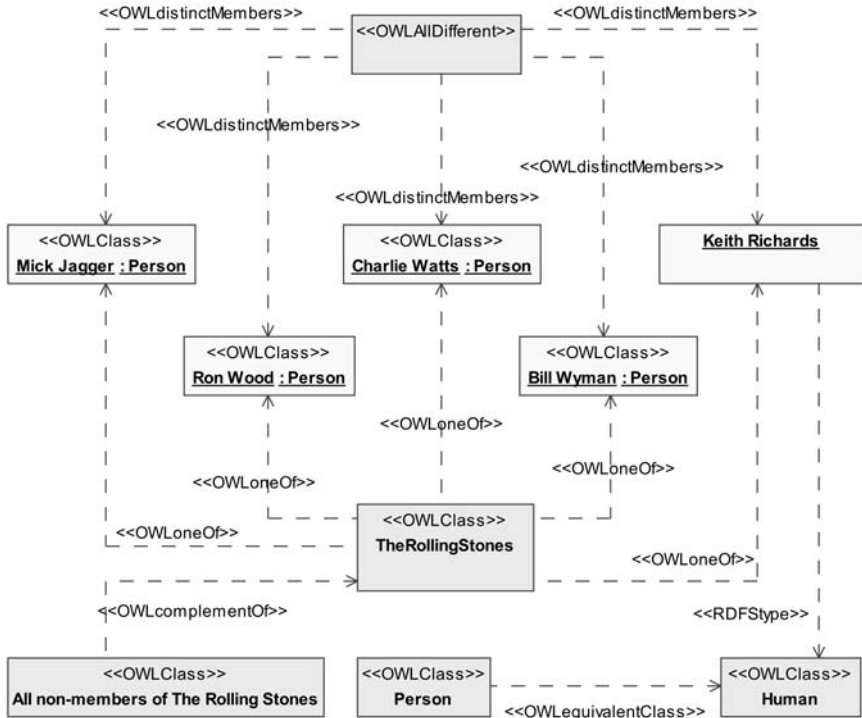


Fig. 9-1. Class Diagram showing relations between Ontology Classes and Individuals in the Ontology UML Profile

Figure 9-1 shows various types of ontology classes modeled in UML. The «OWLClass» Person is an example of an owl:Class class that is identified by a class identifier, while TheRollingStones is an enumeration. There is a class “All non-members of The Rolling Stones” that represents the complement of The Rolling Stones – all individuals whose type is not The Rolling Stones belong to this class. AllDifferent is an auxiliary class whose members are different individuals. Also shown is the «OWLClass» Human and the «equivalentClass» Dependency, which means that Person and Human are classes that have the same class description (i.e. all Persons are Humans and vice versa). Note that in object-oriented modeling it would be highly unusual to model The Rolling Stones as a class rather than as an object of type RockNRollBand. However, ontology classes are not behavioral, but sets, and how would you call a set of all members of The Rolling Stones? Obviously – The Rolling Stones.

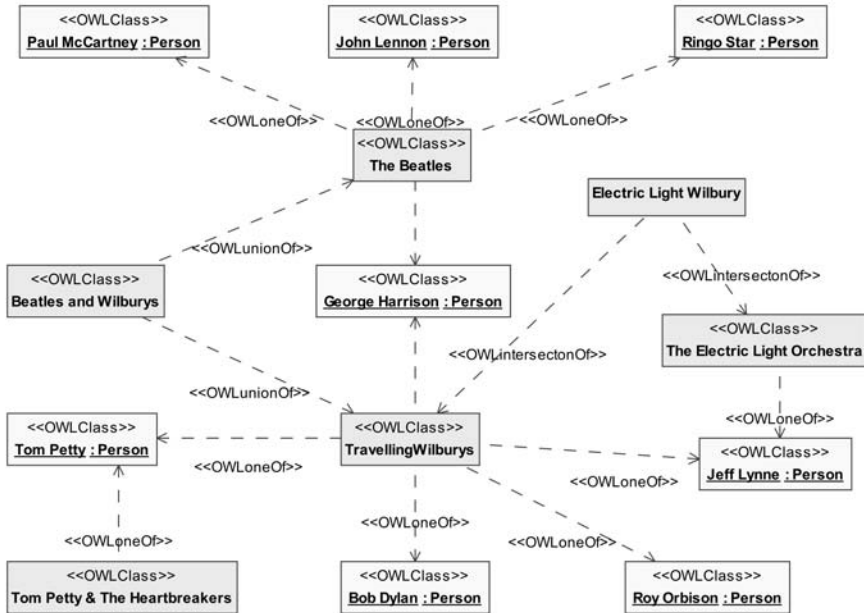


Fig. 9-2. Constructing union and intersection in Ontology UML Profile

In the ODM, an instance of `OWLClass` is an `OWLThing`, an individual. An instance of `RDFSCClass` is an `RDFSResource`, which means that it can be anything. In UML, an instance of a `Class` is an `Object`. `OWLThing` and `UML Object` have some differences, but they are similar enough, and so in the Ontology UML Profile, an `OWLThing` is modeled as a `UML Object`, which is shown in Fig. 9-1 and Fig. 9-2. The stereotype of an object must match the stereotype of its class (`<<OWLClass>>` in this case). The `<<OWLThing>>` stereotype could be added as well. We can state that an individual has a type in three ways:

- By using an underlined name of an individual followed by “:” and its `<<OWLClass>>` name. For example, `Mick Jagger:Person` is an individual (`OWLThing`) whose type is `Person`. This is the usual UML method of stating an `Object`’s type.
- By using a stereotype `<<RDFSType>>` between an individual and its `<<OWLClass>>`. This method is also allowed in standard UML using the stereotype `<<instanceOf>>`. For example, Keith Richards has `<<RDFSType>>` dependency link to `Human`, which is equivalent with `Person` (`<<OWLequivalentClass>>`). Thus, he is also a `Human`, just like other members of `The Rolling Stones`.

- Indirectly, through logical operators on «OWLClass». If an «OWLClass» is a union, intersection or complement, it is a class of individuals that are not explicitly defined as instances of it. For example, in Fig. 9-2 Bob Dylan is not explicitly defined as a member of the Beatles and Wilburys union class, but it is its member since he is a member of Travelling Wilburys, which is connected with the Beatles and Wilburys through an «OWLUnionOf» connection. A similar thing applies to Jeff Lynne and Electric Light Wilbury Class. Since he is a member of Travelling Wilburys and The Electric Light Orchestra, he is a member of Electric Light Wilbury, an «OWLIntersectionOf».

Again, do not confuse an instance-of relationship between a UML Object and a UML Class or between an OWLThing and an OWLClass (all in the M1 layer) with the relationship between, for example, an instance of OWLClass (M1) and an OWLClass concept (M2). The later is a linguistic instance-of relation, an instance-of relation between concepts from different layers (the definition of OWLClass and a concrete OWLClass, for instance Tom Petty & the Heartbreakers). The ontological instance-of relation is an instance-of relation between concepts that are in the same linguistic layer, but in different ontological layers (for instance, «OntClass» Person and the object George Harrison are at different ontological layers since Human is the class (type) of George Harrison). For a more detailed discussion of ontological versus linguistic instance-of relations, see [Atkinson & Kühne, 2003].

9.2 Properties of Ontologies

The concept of Property is one of the most unsuitable concepts in ontologies for model with object-oriented languages and UML. The problem arises from a major difference between Property and the UML concepts similar to it, Association and Attribute. Since Property is an independent, stand-alone concept, it can not be modeled directly with Association or Attribute, which cannot exist on their own. Some authors [Baclawski et al., 2002a] have suggested extending UML with new constructs to support a stand-alone Property, introducing aspect-oriented concepts into UML. In our view, this solution is rather extreme, since it demands non-standard changes to UML.

Since Property is a stand-alone concept it can be modeled using a stand-alone concept from UML. That concept could be the UML Class' stereotype «RDFProperty», «OWLObjectProperty», or

«OWLDatatypeProperty». However, Property must be able to represent relationships between Resources (Classes, Datatypes, etc. in the case of UML), which a UML Class alone is not able to do. If we look at the definition of a Property in the ODM more closely, we can see that it accomplishes representation of relations through its range and domain. We have found that in the Ontology UML Profile, the representation of relations in accordance with the ODM model should be modeled with the UML Association's or UML Attribute's stereotypes «domain» and «range». In order to increase the readability of diagrams, the «range» association is unidirectional (from a Property to a Class).

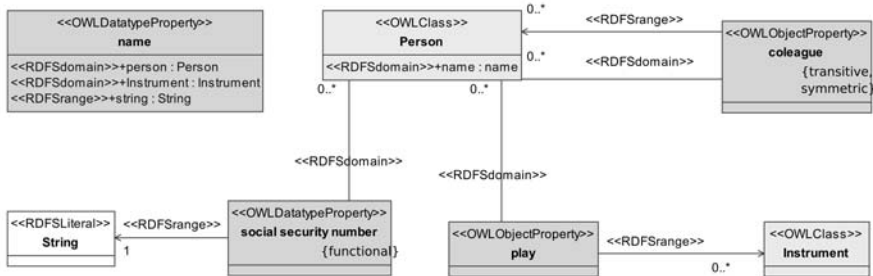


Fig. 9-3. Ontology Properties shown in a UML Class diagram

OWL defines two types (subclasses) of Property – OWLObjectProperty and OWLDatatypeProperty. OWLObjectProperty, which can have only individuals in its range and domain, is represented in Ontology UML Profile as the Class' stereotype «OWLObjectProperty». OWLDatatypeProperty is modeled with the Class stereotype «OWLDatatypeProperty».

An example of a Class Diagram that shows ontology properties modeled in UML is shown in Fig. 9-3. It contains four properties: two «OWLDatatypeProperty»s (name and socialSecurityNumber) and two «OWLObjectProperty»s (play and colleague) UML Classes. In cooperation with «RDFSdomain» and «RDFSrange» UML Associations, or «RDFSdomain» and «RDFSrange» UML Attributes, these properties are used to model relationships between «OWLClass» UML Classes. Tagged values describe additional characteristics, for example, the «OWLObjectProperty» colleague is symmetric (if one Person is a colleague of another Person, the other Person is also a colleague of the first Person) and transitive (if the first Person is a colleague of the second Person, who is a colleague of the third Person, the first and the third Person are colleagues).

There is an important issue that must be clarified with respect to this diagram. In UML, relations are represented by Associations (represented graphically as lines) or Attributes, which looks nice and simple. Ontology UML Profile diagrams may look overcrowded, since each relationship requires a box and two lines to be properly represented. The solution used here uses standard graphical symbols, but UML allows custom graphical symbols for a UML profile. For example, a custom graphical symbol for a Property could be a tiny circle with lines, which reduces the space required on a diagram. Additional custom settings, such as distinct colors for «OWLClass» (green), «OWLObjectProperty» (orange) and «OWLDatatypeProperty» (orange), can be used to increase the readability of diagrams. For the sake of readability, the UML profile that we have used allows two styles of presentation of the domain and range of an «OWLDatatypeProperty». An example of the first style (a UML Class with two UML Associations) is socialSecurityNumber, and an example of the second one (a Class with Attributes as its domain or range) is name. The second style is allowed only for an «OWLDatatypeProperty» whose range multiplicity is equal to or less than one. So, if an «OWLDatatypeProperty» has a range multiplicity of 0..1 or 1, the style using Attributes can be used to reduce the clutter.

9.3 Statements

OWLStatement is a concept that represents concrete links between ODM instances – individuals and data values. In UML, this is done through Link (an instance of an Association) or AttributeLink (an instance of an Attribute). A Statement is a kind of instance of a Property, which is represented by a UML Class stereotype («OWLObjectProperty» or «OWLDatatypeProperty»). Since an instance of a Class in UML is an Object, a Statement in the Ontology UML Profile is modeled with the Object's stereotype «OWLObjectProperty» or «OWLDatatypeProperty» (the stereotype of an Object in UML must match the stereotype for its Class' stereotype). UML Links are used to represent the subject and the object of a Statement. To indicate that a Link is the subject of a Statement, LinkEnd's stereotype «RDFsubject» is used, while the object of the Statement is indicated with LinkEnd's stereotype «RDFobject». LinkEnd's stereotypes are used because, in UML, Link cannot have a stereotype. These Links are actually instances of Property's «RDFdomain» and «RDFrange». In brief, in the Ontology UML Profile a Statement is represented as an Object with two Links – the subject Link and the object

Link, which is shown in Fig. 9-4. The Persons represented, Mick Jagger and Keith Richards, are colleagues. Keith Richard also plays an Instrument, guitar.

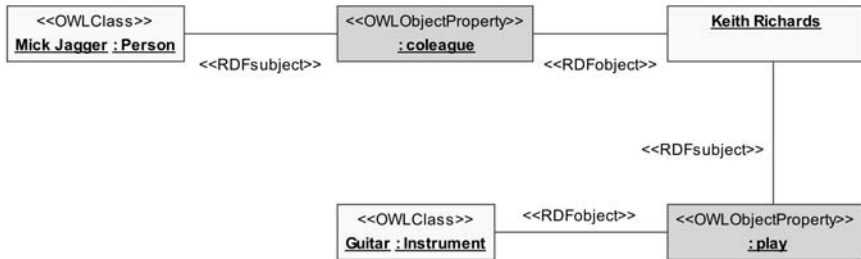


Fig. 9-4. Individuals and Statements shown in a UML Object diagram

As with Properties of an ontology, the diagram's readability can be further increased by using distinct colors and custom graphical symbols. A tiny circle can be used instead of the standard box for representing a Statement in order to reduce clutter in a diagram.

9.4 Different Versions of the Ontology UML Profile

The ODM specification (and especially the part that deals with the Ontology UML Profile) is still under development. For that reason, the final version of the Ontology UML Profile will probably be different than the version we have described. However, the version described here should be very useful for getting a feeling for what it is like to create ontologies with UML. It is very easy to get accustomed to a similar profile once you have got a feel for working with one profile.

To show you what the differences could look like, we shall show you diagrams of an ontology similar to those which we have just talked about. There is another reason why we are showing these diagrams here. Some of the tools described and the discussion in this book refer to this older version of the Ontology UML Profile [Djurić et al., 2005b] which is called GOOD OLD AI Ontology UML Profile . This profile was later updated (but there is no point in updating tools until the official specification has been finished).

So, here is how classes look like (Fig. 9-5):

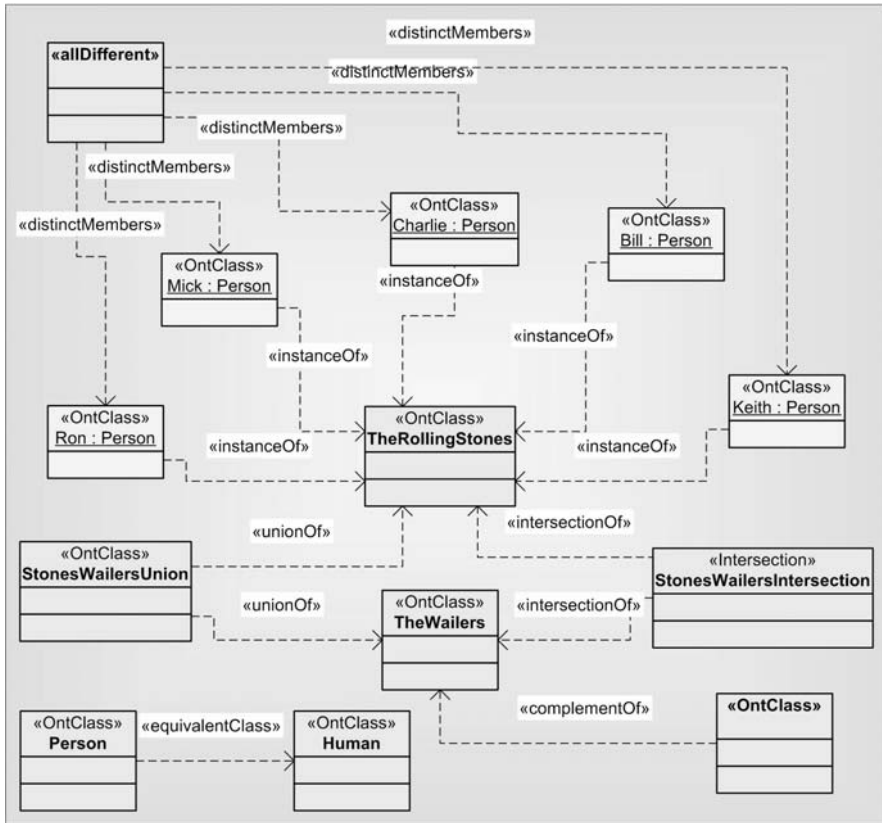


Fig. 9-5. Class Diagram showing relations between Ontology Classes and Individuals in the Ontology UML Profile

Properties (Fig. 9-6):

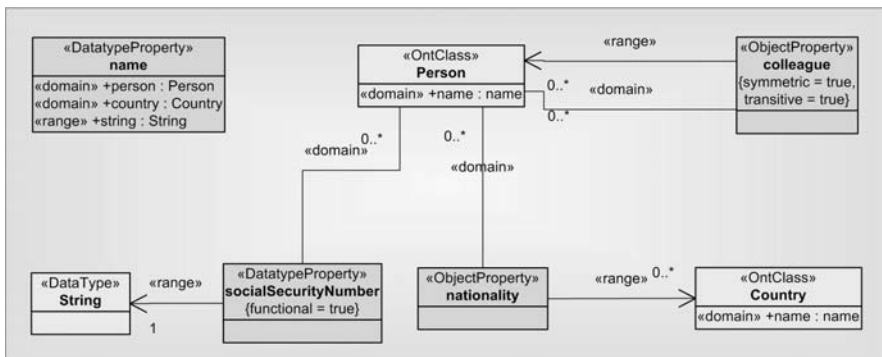


Fig. 9-6. Ontology Properties shown in UML Class Diagram

Statements (Fig. 9-7):

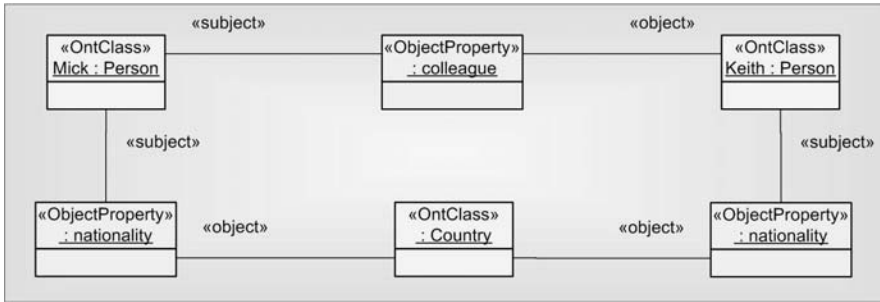


Fig. 9-7. Individuals and Statements shown in a UML Object Diagram

Of course, when the specification has been completed, you should look at the specification document for the exact details. Some of these details, especially the most important ones, will probably be the same or almost the same as those we have described in this chapter. However, there might be many less important details that are a little different. Something that is important, however, is that you can start from the examples that we have shown you and very quickly catch up with the specification.

10. Mappings of MDA-Based Languages and Ontologies

The MOF-based ontology metamodels of ontology languages presented in this book, namely the ODM and the Ontology UML Profile (OUP), are defined in the context of the MDA's metamodeling architecture. However, such a definition is not sufficient; they need to interact with real-world ontologies, for example with OWL ontologies. It is obvious that we need to develop transformations to support conversions between MDA ontology languages and OWL. In fact, this has also been requested by the OMG ODM RFP, but neither the present approach nor the current ODM draft specification itself provides a thorough solution to this problem, i.e., first proposing a conceptual solution and then digging into implementation and technological details. In this chapter, we describe all these transformations in terms of the modeling and technical spaces that we described earlier in the book (see Chap. 5). Accordingly, we first identify all modeling spaces related to this problem and depict their mutual relations. Using those relations, we recommend tools and techniques for implementing these transformations, and finally describe one of such implementation.

10.1 Relations Between Modeling Spaces

Figure 10-1 shows all modeling spaces that we have recognized as important for the MDA standards and the present Semantic Web ontology languages (OWL and RDF(S) in the first place) to be used cooperatively. In the MOF modeling space, we have defined the ODM and the OUP. It is important to note that the ODM is defined in the M2 layer, while the OUP resides in both the M1 and M2 layers according to [Atkinson & Kühne, 2002]. In Fig. 10-1, we show them both in the M2 layer for the sake of clarity. Concrete real-world models are located in the M1 layer and consist of classes and their instances. According to [Atkinson & Kühne, 2003], we must have more than one ontological layer in the M1 layer, and in the case of UML we have two ontological layers: one for classes and one for class instances (i.e., objects). For all MDA layers, one can use XMI, an XML-

compatible format for sharing metadata that we have described in Chap. 4. In the bottommost layer (M0, not shown in the figure, but below M1), these are things from reality.

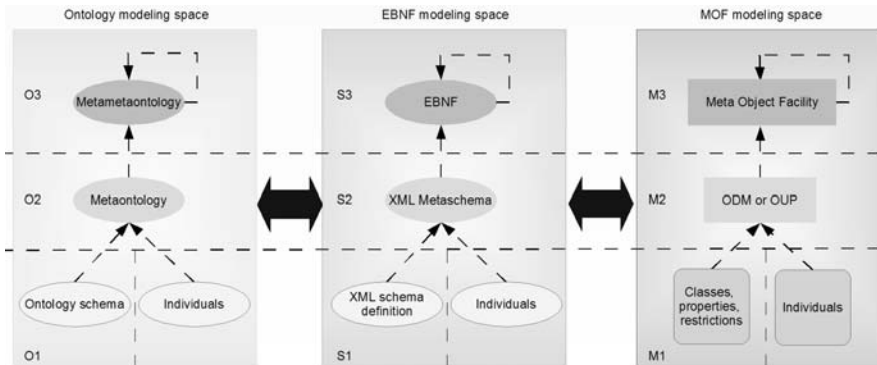


Fig. 10-1. M2-based mappings between the ontology modeling space and the MOF modeling space through the EBNF modeling space

The ontology modeling space includes the W3C recommendation for the Web Ontology Language (OWL). This ontology language is based on XML and RDF(S), and thus an XML format is being used for exchanging OWL ontologies. In this modeling space, we can identify various abstraction layers in order to find its relations to the MOF modeling space. The layer above bottommost is denoted as O1. In the O1 layer we build ontologies, i.e., we create classes, properties, relations, and restrictions. Ontological instances are located in the O1 layer in the ontology modeling space. We use an analogy between the topmost layer defined in the ontology modeling space and the results given in [Decker et al., 2000]. In that paper, an ontology language (Ontology Inference Layer, or OIL) was described using another ontology language (RDF(S)). In fact, Decker et al. created a metaontology. Finally, we can say that there is a metaontology that defines OWL and that this metaontology is in the O2 layer. We also refer to this modeling space, as well as the RDF(S) modeling space described in Chap. 5 relying on a similar assumption. Actually, we have generalized the treatment, in that the O3 layer is defined using RDF(S) as well. Note that such a layered organization of ontologies is already known in AI as Brachman's distinction of knowledge representation systems [Brachman, 1979]. However, this organization does not make a separation between ontological and linguistic relations [Atkinson & Kühne, 2003], as the schema and instances of an ontology are defined in different layers, similarly to what the MDA modeling framework initially did (M1 for

models and M0 for instances). Using definitions of ontological and linguistic relations as well as modeling spaces, we have recognized that we actually have the same situation in the ontology modeling space – two ontological layers in the O1 linguistic layer of the ontology modeling space. In the bottommost layer of the ontology modeling space (O0), there are things from reality.

We now make some important statements that one must take account of in order to provide transformations between these two modeling spaces:

1. The role of the MOF (M3) is epistemologically equivalent to the role of a metametaontology (O3).
2. The role of the O2 layer (metaontology) is equivalent to that of the metamodel layer (M2) (e.g., the ODM and the OUP), that is to say, they both specify ontology languages in terms of different modeling spaces.
3. The O1 layer has a role equivalent to that of the model layer (M1) of the MOF modeling space. In fact, this conclusion comes from Atkinson and Kühne's ontological and linguistic layers [Atkinson & Kühne, 2003], where one linguistic layer (in this case M1) can contain many ontological layers. Accordingly, the two ontological layers residing in the O1 linguistic layer (ontology schema and ontology instances) are equivalent to the classes and objects comprising the M1 layer in the MOF modeling space.

Formally, we can say that: $M3 \Leftrightarrow O3$, $M2 \Leftrightarrow O2$, and $M1 \Leftrightarrow O1$.

Since both the ontology and the MOF modeling spaces use XML for sharing their metadata, we can include a new modeling space in this discussion. Of course, this is the Extended Backus–Naur form (EBNF) modeling space that defines XML. This modeling space also has a layered organization very similar to the organization of both the ontology and the MOF modeling spaces, but this organization is defined in terms of syntax (not semantics) [Klein, 2001]. We can look at the EBNF modeling space in terms of the W3C XML Schema recommendation. The topmost level (S3) of the EBNF modeling space is a universal, well-known, and self-defined syntax also called EBNF. When we analyze the context of the XML technical space formally defined in the EBNF modeling space, we find that the S2 layer comprises a schema for schemas (i.e., a metaschema). This metaschema defines the validity of XML Schema definition documents. Domain-specific XML vocabularies (i.e., schemas) and concrete XML documents are defined in the S1 layer. Accordingly, we can draw conclusions about equivalences between these two modeling spaces: $S3 \Leftrightarrow$

O3, $S2 \Leftrightarrow O2$, and $S1 \Leftrightarrow O1$. Finally, note that there exist similar relations between the MOF and EBNF modeling spaces: $S3 \Leftrightarrow M3$, $S2 \Leftrightarrow M2$, and $S1 \Leftrightarrow M1$.

Avoiding elaborations of the most recent research issue of an M3-neutral infrastructure for bridging modeling spaces [Bézivin et al., 2005], we have used the above epistemological relations between these three modeling spaces to describe the transformations requested in the OMG ODM RFP. In fact, these transformations should be applicable to the model layer (M1, O1, and S1), but they are actually driven by their metamodels (i.e., metaontologies or metaschemas) [Bézivin et al., 2005].

10.2 Transformations Between Modeling Spaces

It is obvious from the descriptions above that we cannot provide direct mappings between the ontology modeling space and the MOF modeling space. In fact, this transformation can only be defined through the EBNF modeling space. It is important to define a *pair* of transformations in order to enable two-way mapping (one transformation for each direction) between all OWL ontologies and all ontologies represented in an MDA-based ontology language. The transformations can be based on “metadefinitions” of OWL (i.e. on its metaontology) and an MDA-compliant language (i.e., a metamodel). This transformation principle is compliant with Bézivin’s principle of metamodel-based model transformation [Bézivin, 2001]. Practically, in terms of the EBNF modeling space, these transformations are based on the XML schemas of both OWL and XMI (i.e., the XML schema of the UML and ODM XMI formats). Figure 10-2 shows the OUP/OWL transformation of an OUP model (i.e., an OUP document in XMI format) to its equivalent OWL ontology (i.e., an OWL document in XML format). This transformation maps the M1 layer into its corresponding OWL layer (O1). The most suitable implementation for this transformation is in XSLT, since in this case an XML document is converted into another XML document. The opposite transformation (from OWL to the OUP) can also be implemented in XSLT. However, we do not recommend this kind of implementation since we may use different XML representations (e.g., XML schemas) with different XML tag names to represent the semantics of an OWL ontology and its instances (for details see [Decker et al., 2000]). For example, the concept *book* can be represented by several different tags (*book*, *bookInfo*, etc.), but the ontology specifies that all these tags represent the same ontology concept. In this case we suggest using a

programming language to implement the transformation. For example, this transformation can be implemented in Java, but Java needs to be empowered with an OWL parser (e.g., Jena [McBride, 2002]). Note also that a concrete XSLT is a valid XML document. Accordingly, we can say that the XSLT language itself is defined using an XML schema that resides in the metaschema layer (S2), as the XML Schema language itself.

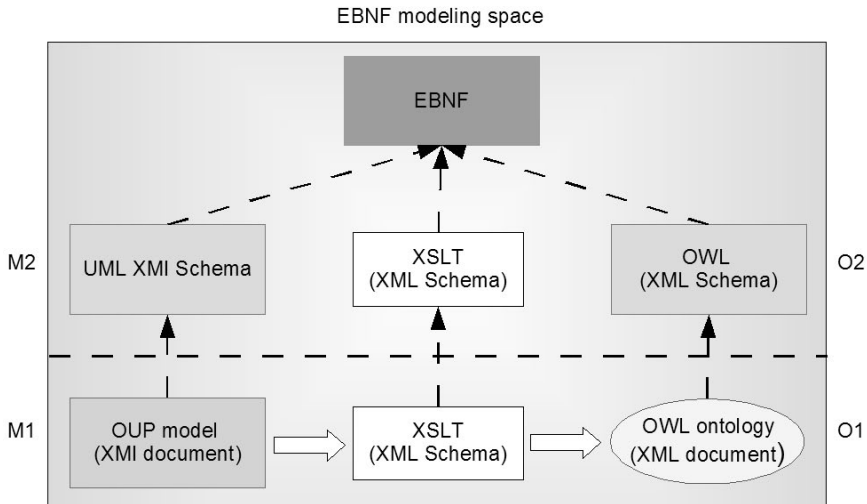


Fig. 10-2. An example of a transformation in the XML technical space: the transformation of the OUP into OWL

Figure 10-3 shows an example of some transformations within the MOF modeling space. This figure is organized according to the transformation schema proposed in [Kurtev & van den Berg, 2003] for transforming XML schemas to application models. In the MOF modeling space, we can only transform those ontology languages that have an MOF-compliant metamodel. Therefore, we illustrate the transformation between the OUP and the ODM. In terms of the MOF modeling space, the transformations between these languages should be implemented in one of the QVT languages (e.g., ATL [OMG QVT, 2003]), and the chosen QVT language should have its own metamodel. Although the OUP metamodel can reside in both the M1 and M2 layer, we have placed it in the M2 layer (see Fig. 10-3) in order to avoid problems that can arise when transforming between different metamodeling layers. As a matter of fact, this can happen if we use a standard UML profile for ontology development without the user's extensions (see [Atkinson & Kühne, 2002] for details). Note that this transformation can also be implemented through the EBNF modeling space in terms of XML schemas and XML documents.

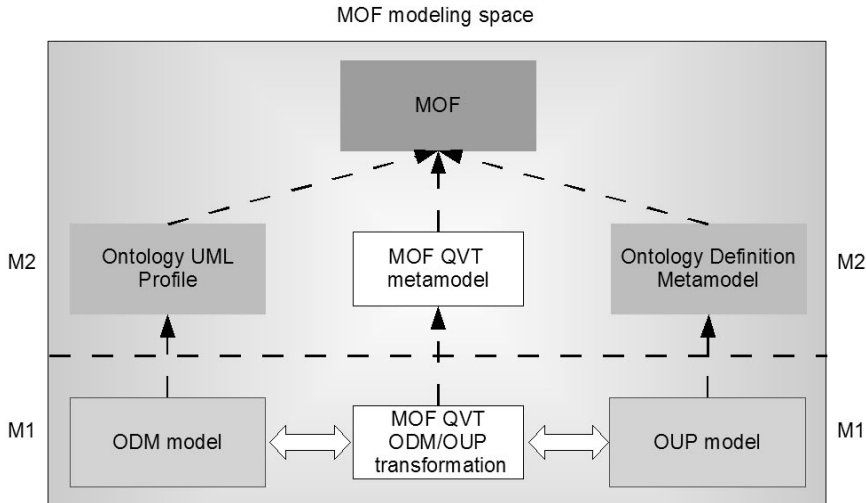


Fig. 10-3. Transformations in the MOF modeling space: transformations between OUP and ODM

Summarizing all these facts about transformations between the ontology and MOF modeling spaces, Fig. 10-4 gives some guidelines on how to perform transformations between every pair of the languages discussed above. In this figure, we indicate only the transformations between the languages considered that can be done by employing one transformation technique. At first sight, one might think that transformation between MDA-based languages (i.e., the ODM and the OUP) and OWL can be done within the MOF modeling space. However, that is not possible, since OWL is not part of the MOF modeling space, and it is not a MOF-based language. This means that a MOF-based transformation technology (e.g., QVT) cannot be applied to that pair of transformations. QVT can only be applied to MOF-based languages. So, we have to look for an intersection between those two languages. Since bridges exist between OWL and XML and between the MOF and XML, but not between OWL and the MOF, the transformation can only be defined in the EBNF modeling space as shown in Fig. 10-4. However, if one regards the OMG's ongoing model-to-text initiative [OMG MOF-to-Text, 2004] as a part of the MDA technical space, then transformations between the ODM and OWL are possible in terms of the MDA technical space, but they are still not part of the MOF-modeling space. Since the model-to-text initiative is in its initial stages of development (i.e., there is just a request for proposals to the OMG [OMG MOF-to-Text, 2004]) and there is not any test implementation of that proposal at the moment, we have not considered this case in Fig. 10-4.

Consequently, this figure should not be regarded as a final list of possible transformations.

		Target language				
		ODM		OUP		OWL
Source language	ODM	-		EBNF MS	MOF TS	EBNF TS
				XSLT	QVT	
	OUP	EBNF TS	MOF TS	-		EBNF TS
		XSLT	QVT			XSLT
	OWL	EBNF TS		EBNF TS		-
		Programmed*, XSLT		Programmed*, XSLT		

* preferred case (e.g. Java empowered with a library for parsing OWL)

Fig. 10-4. Overview of possible transformations between OWL, the ODM, and the OUP: modeling spaces in which the transformations can be done (EBNF and MOF modeling spaces) and implementation technologies for these transformations (XSLT, Query/View/Transformation (QVT), and programming languages)

10.3 Example of an Implementation: an XSLT-Based Approach

In the previous section, we explained the conceptual solutions for transforming between MOF-based ontology languages (i.e., the ODM and OUP) and OWL. In this section, we show an example of an implementation that transforms an OUP-based ontology to its equivalent OWL ontology [Gašević et al., 2004a; Gašević et al., 2004b; Klein & Visser, 2004]. The transformation provides us with Semantic Web ontologies that can be used in real-world Semantic Web applications. It was developed to support the GOOD OLD AI Ontology UML Profile (OUP) [Djurić et al., 2005b], since the official OMG UML Ontology UML Profile has not yet been adopted .

10.3.1 Implementation Details

The main idea of having a UML profile for ontology development is to use existing UML tools. In fact, current UML tools (e.g., IBM Rational Rose and Poseidon for UML) support mainly the XMI standard [OMG XMI, 2002] – an MDA XML-based standard for metamodel, metamodel, and model sharing. Since XMI is XML-based, one can employ XSLT to transform XMI documents into the target documents. These target documents can be written in an ontology language, for example OWL, which also has its own XML syntax. On the other hand, when we use an approach based on the XSLT principle, we do not need to modify a UML tool; instead, we just apply an XSLT to the output of the UML tool. Accordingly, we can use the well-defined XML/XSLT procedure shown in Fig. 10-5.

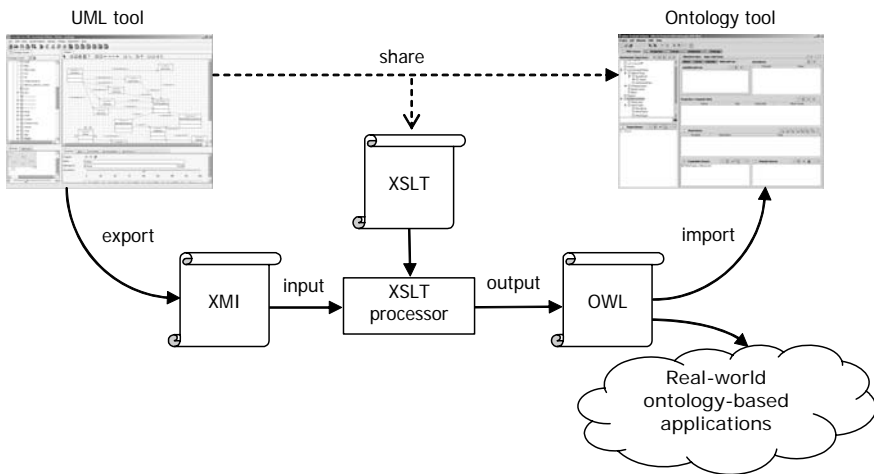


Fig. 10-5. Using the XSLT principle: extensions of present UML tools for ontology development

A UML tool can export an XMI document that an XSLT processor (e.g., Xalan; <http://xml.apache.org>) can use as input. An OWL document is produced as the output, and this format can be imported into a tool specialized for ontology development (e.g., Protégé [Protégé, 2005]), where it can be further refined. On the other hand, since we obtain an OWL document, we do not need to use any ontology tool; the document obtained can be used as the final OWL ontology.

The XSLT that we have implemented for mapping OUP models to OWL ontologies contains a set of rules (i.e., XSLT templates) that match the UML XMI constructs of the OUP models and transform them into

equivalent OWL primitives. In developing these rules, we faced some serious obstacles resulting from evident differences between the source and target formats. We note some of these obstacles below:

- The structure of a UML XMI document is fairly awkward, since it contains a full description of the UML model that it represents, for example classes, attributes, relations (associations, dependencies, and generalization), and stereotype descriptions.
- In some cases, the OUP uses more than one UML construct to model one OWL element. For example, to model the *someValuesFrom* restriction using OUP (see Fig. 10-7), we need three UML classes and three relations (i.e., one association and two dependencies). This is especially difficult, since each UML construct has a different stereotype.
- UML tools can only draw UML models, and they do not have an ability to check the completeness of an OUP ontology. Thus, the XSLT is required to perform this kind of checking of XMI documents. This is the only way to avoid the generation of erroneous OWL ontologies.
- The XSLT must make a distinction between classes that are defined in other classes (and cannot be referenced using their ID), and classes that can be referenced using their ID. Accordingly, we have included the *odm.anonymous* tagged value in the OUP, which help us differentiate between these two cases.

Taking all this into account, it becomes obvious that the XSLT that we have developed is too large to be included in this book. However, this transformation is available on the book's Web page together with supplementary materials.

10.3.2 Transformation Example

In order to illustrate the application of the above XSLT, we have built the well-known Wine ontology using the GOOD OLD AI Ontology UML Profile [Djurić et al., 2005b]. We show a part of this ontology, in Fig. 10-6. `WineDescriptor` is a class equivalent to the union of the classes `WineTaste` and `WineColor`, whereas the `WineColor` class is an enumeration of the instances of the `WineColor` class (i.e., individuals), `White`, `Rose`, and `Red`. Note that there are two anonymous classes (Union and Enumeration) in Fig. 10-6. These classes are defined through other classes (e.g., the anonymous Enumeration is defined in the class `WineColor`) and cannot be used outside of their definitions.

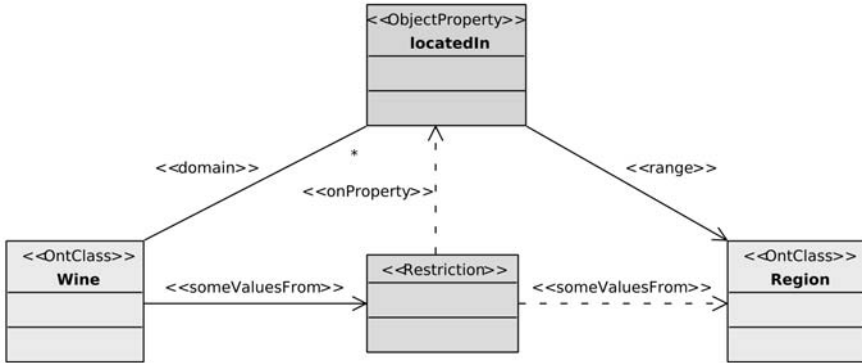


Fig. 10-7. An OUP class property and its restriction in the Wine ontology

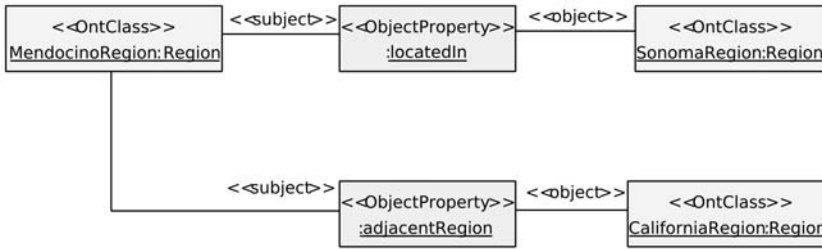


Fig. 10-8. The OUP fully supports ontology instances through OUP statements: an example from the Wine ontology

Figure 10-9 depicts an excerpt from the output OWL document generated by the XSLT. Figure 10-9a shows the OWL representation of the classes defined in Fig. 10-6. Note how OUP classes with the tagged value *odm.anonymous* are mapped into OWL (e.g., WineDescriptor has an equivalent anonymous class, defined as the union of the WineTaste and WineColor classes). Figure 10-9b shows the OWL definition of the locatedIn property, which has the Region class as its range, and both the Region and the Wine classes as its domain. On the other hand, the Wine class additionally restricts this property, using the OWL *someValuesFrom* restriction. Since the OUP provides full support for OWL statements, we are able to transform these statements into equivalent OWL constructs (i.e., full individual descriptions). Figure 10-9c shows the OWL representation of the statements shown in Fig. 10-8.

This example clearly illustrates the power of our solution to generate both a nontology schema [Devedžić, 2002] (classes, properties, etc.) and ontology instances (bodies of knowledge) [Chandrasekaran et al., 1999]. This feature is not supported in other MDA-based proposals for ontology development.

In the next subsection, we outline our first practical experience with this solution.

10.3.3 Practical Experience

We have already noted that the transformation that we have developed enables us to create complete OWL ontologies using standard UML tools – specific ontology development tools are not a necessity for ontology development anymore. Specifically, to use the OUP and our XSLT in practice, we need to employ a suitable UML tool that supports the following:

- Attaching stereotypes to all for the UML concepts that we have in the OUP. For instance, the present UML tools rarely allow objects and link ends to have a stereotype (e.g., the objects in Fig. 10-6).
- A convenient way to use tagged values and attach them to any UML element (e.g., the *odm.anonymous* tagged value in Fig. 10-6).
- Creating relations between different UML primitives, such as those shown in Fig. 10-6. We emphasize especially the importance of relations (e.g., UML dependencies) between a UML class and a UML object. This kind of relation is valid in the UML syntax, and can be represented on class diagrams (also called static structure diagrams in the UML specification [OMG UML, 2003a; OMG UML, 2003b]).
- The XMI standard for serialization of UML models, since our XSLT is based on the UML XMI format.

We have analyzed two UML tools: IBM Rational Rose¹ and Poseidon for UML.² Whereas Poseidon for UML satisfies all of the above mentioned requirements, IBM Rational Rose does not provide support for most of them (e.g., an object cannot have a stereotype, and a class and an object cannot be related using any UMLs relation). Additionally, Poseidon for UML uses the NetBeans MDR³ repository for storing MOF-compliant metamodels, and the definition of the MOF itself. This is an important feature, because by using model repositories we can benefit from all of the advantages of the MDA [Bock, 2003]. Furthermore, Poseidon for UML has already found its place in ontological engineering, since it is a UML tool recommended to be used with the Protégé UML back end for importing UML models [Protégé UML, 2004].

¹ <http://www.rational.com>

² <http://www.gentleware.com>

³ <http://mdr.netbeans.org>

<pre> <owl:Class rdf:ID="WineDescriptor"> <owl:equivalentClass> <owl:Class> <owl:unionOf rdf:parseType="Collection"> <owl:Class rdf:about="#WineTaste"/> <owl:Class rdf:about="#WineColor"/> </owl:unionOf> </owl:Class> </owl:equivalentClass> </owl:Class> <owl:Class rdf:ID="WineTaste"> <rdfs:subClassOf rdf:resource="#WineDescriptor"/> </owl:Class> <owl:Class rdf:ID="WineColor"> <rdfs:subClassOf rdf:resource="#WineDescriptor"/> <owl:equivalentClass> <owl:Class> <owl:oneOf rdf:parseType="Collection"> <WineColor rdf:about="#Red"/> <WineColor rdf:about="#Rose"/> <WineColor rdf:about="#White"/> </owl:oneOf> </owl:Class> </owl:equivalentClass> </owl:Class> </pre> <p style="text-align: center;">a)</p>	<pre> <owl:ObjectProperty rdf:ID="locatedIn"> <rdfs:range rdf:resource="#Region"/> <rdfs:domain rdf:resource="#Wine"/> </owl:ObjectProperty> <owl:Class rdf:ID="Wine"> <!-- ... --> <rdfs:subClassOf rdf:resource="#PotableLiquid"/> <rdfs:subClassOf> <owl:Restriction> <owl:onProperty rdf:resource="#locatedIn"/> <owl:someValuesFrom rdf:resource="#Region"/> </owl:Restriction> </rdfs:subClassOf> </owl:Class> </pre> <p style="text-align: center;">b)</p> <pre> <Region rdf:ID="SonomaRegion"/> <Region rdf:ID="CaliforniaRegion"/> <Region rdf:ID="MendocinoRegion"> <locatedIn rdf:resource="#SonomaRegion"/> <adjacentRegion rdf:resource="#CaliforniaRegion"/> </Region> </pre> <p style="text-align: center;">c)</p>
--	--

Fig. 10-9. The resulting OWL description: (a) classes generated for the OUP shown in Fig. 10-6; (b) object property OWL descriptors for the model shown in Fig. 10-7; (c) OWL statements obtained from Fig. 10-8

The second important decision is how to generate the OWL equivalent of an OUP model, since the same OWL definition (e.g., an OWL class) can be generated in more than one way (e.g., an OWL class can be defined using an unnamed class as either an equivalent class or a subclass). We decided to generate OWL ontologies using a technique similar to that employed by the Protégé OWL plug-in.⁴ Hence, we have managed to provide an additional way to import Poseidon's models into Protégé through OWL. Since Protégé has more advanced features for ontology development, it can be used to further improve and refine an OUP-defined ontology.

We have tested our solution using the Wine ontology [Noy et al., 2001]. First, we represented this ontology in Poseidon using the OUP. Parts of this ontology were used in the previous subsection in order to illustrate the OUP (e.g., Fig. 10-6, Fig. 10-7, Fig. 10-8). Then we exported this extended UML representation into XMI. After applying the XSLT, we obtained an OWL document. Finally, we imported this document into Protégé using the OWL plug-in. A screenshot that depicts a part of this imported OWL ontology is shown in Fig. 10-10.

⁴ <http://protege.stanford.edu/plugins/owl/>

We must admit that we have found a certain difference between the OWL generated by the XSLT and the OWL produced by Protégé. The difference was detected in the representation of OWL individuals. To represent individuals, Protégé uses `owl:Thing` with an attribute `rdf:type` that refers to its type (i.e., its OWL class). For example, `Red` is an instance of the `WineColor` class, and is represented as follows:

```
<owl:Thing rdf:ID="Red" rdf:type="#WineColor"/>
```

In our solution, an individual is represented using a tag that has the same name as its OWL class. For example, the same instance of `Red` is represented as follows:

```
<WineColor rdf:ID="Red"/>
```

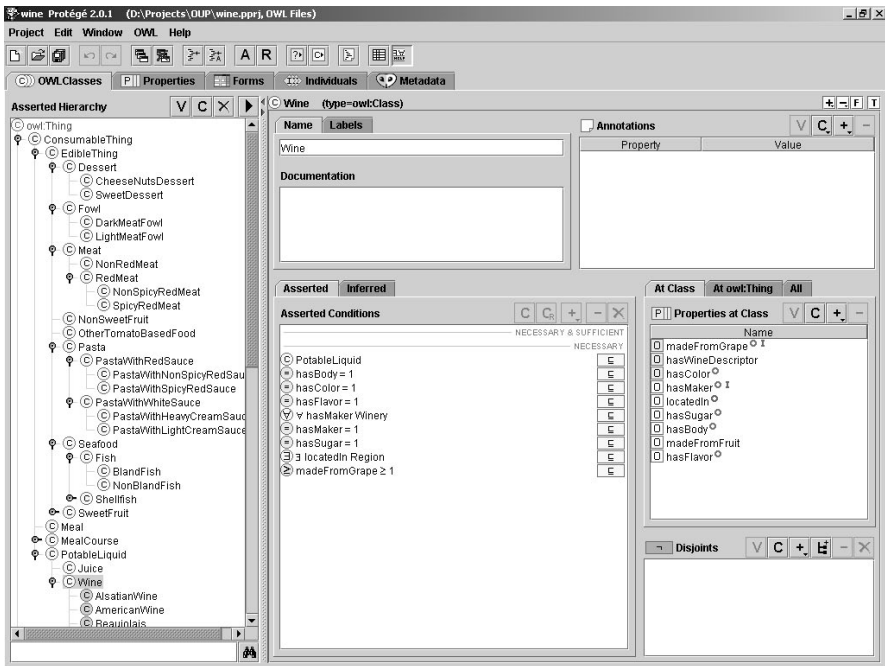


Fig. 10-10. An example of an OWL ontology generated from the OUP and imported into Protégé: the Wine ontology

We have found this difference unimportant, since these two representations of individuals have the same meaning in the OWL notation. Additionally, Protégé is able to recognize OWL instances defined in both ways since it uses Jena for parsing OWL ontologies.

The current XSLT version has a limitation in that it does not support packages (i.e., multiple OUP ontologies in one UML model). This means that it is unable to produce more than one OWL document (i.e., ontology). Actually, even though the OUP supports multiple ontologies within the same XMI project, the XSLT standard and XSLT processors introduce this limitation. Of course, this can be overcome using some nonstandard XSLT primitives (i.e., XSLT extensions) that provide for the production of multiple documents from one source XML document (e.g., the SAXON XSLT processor and its XSLT extensions).

In Chap. 13, we show additional examples of the practical usage of this XSLT in transformations of ontologies developed with OUP into the OWL language.

10.3.4 Discussion

One very important remark to be made is that the XSLT implemented is not a part of the OMG ODM RFP [OMG ODM RFP, 2003]. This RFP presumes transformations between the ODM and the OUP, as well as transformations between the ODM and OWL (see Fig. 10-11). This means that if one wants to transform an OUP-defined ontology into OWL, that ontology should first be transformed into the ODM, and subsequently from the ODM to OWL. However, as we have shown, it is also possible to implement everything using an XSLT because all ontology representations have an XML binding: the ODM, as a MOF-defined metamodel, uses XMI format; the OUP, as a UML profile, uses the UML XMI format; and OWL has an XML-based syntax. Our transformation from the OUP to OWL is a practical extension of the present UML tools that empowers them to be used for the development of ontologies expressed in a Semantic Web ontology language. It is a kind of bridge between ontological and software engineering, since the current MDA-compliant implementations (see Chap. 7) are at a very immature stage. Development of these ODM \leftrightarrow OUP and ODM \leftrightarrow OWL transformations is currently our primary activity.

Transformations from the OUP to the ODM and from the ODM to OWL offer the following advantages:

- When one wants to support a new ontology language (e.g. DAML+OIL) using the ODM-based principle, only one pair of transformations needs to be implemented: from the new language to the ODM, and from the ODM to the new language. If we want to support transformations between n different languages (such as the OUP and OWL), then it is necessary to implement $2n$ transformations. If, instead, we implement

transformations between every pair of ontology languages without using the ODM (e.g., between OWL and DAML+OIL), then we need n^2 transformations.

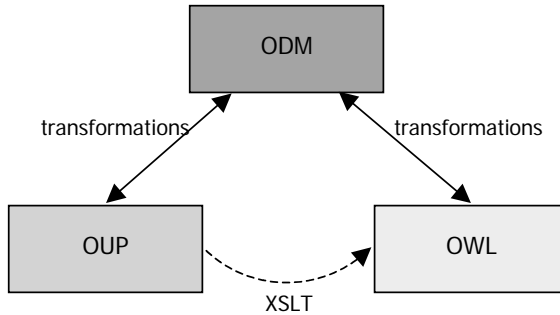


Fig. 10-11. Relations between the solution implemented and the transformations recommended in the OMG ODM RFP

- When transforming ontologies through the ODM, we can validate them against an ontology metamodel (i.e., the ODM). In this way, we can prevent transformation of an invalid ontology or issue a warning when an ontology is inconsistent. This feature is very important for OUP models, since existing UML tools are unable to check UML models that are based on a UML profile.
- Finally, one should note that the transformation mechanism for MOF-based ontologies is driven by ideas from [Bézivin, 2001], in which metamodel-based model transformations were proposed. In the case of MDA-based ontology languages, we have different metamodels (i.e., OWL and the OUP metamodel). However, the ODM serves as an integration point that decreases the number of transformations needed. Also, it is useful to have a central metamodel (in this case, the ODM) for each area.

Part III Applications

11. Using UML Tools for Ontology Modeling

In this chapter we shall give you a few tutorials on how to use some of the currently available UML tools to create ontologies using the Ontology UML Profile.

Before you start, you should be aware of some constraints that the tools that are available today have. The biggest and most important problem is that only a few tools can successfully exchange models with each other. The lack of a common model exchange standard when the first UML tools became widely popular at the end of the 1990s caused differences in the way they perform the serialization of models. Despite the fact that in the meantime UML XMI has been adopted as the standard for exchanging models, some of these tools still prefer proprietary non-XML formats (IBM Rational Rose for example). Even when they perform serialization into UML XMI, you must be aware that the UML XMI formats used are different, and so you cannot usually open a model created in one tool in another tool.

Today, most tools try to support UML2 as some sort of add-in built on top of the facilities of the old UML. Until UML2 is fully supported, which should solve the interoperability problem, you should be very careful when choosing your UML tool. Some are friendlier than others, especially those that are built on “standard” metadata repositories: NetBeans MDR (based on the OMG MOF 1.4 standard), and Eclipse EMF (based on Ecore, which is not standard but has wide support). The rule of thumb is that (currently) tools based on each of these repositories support some interoperability with other tools based on the same repository, but almost no interoperability with the competing repository.

This information may seem too technical to you, but it is important at the current stage of the evolution of UML and MDA tools. Remember, you can draw diagrams using any tool, you can even use chalk and a blackboard, but in the end you would like to be able to open your model in another tool, or to perform transformation to another technology.

For example, your team could start from an ontology, transfer it to a UML platform-neutral domain model, and then generate a Java implementation. To do all these things, you need to be aware of which

tools can “talk” to each other, and with current tools this is often a problem. This is the bad news.

The good news is that serious efforts are being made in the industry to enable interoperability. The OMG UML2 and MOF2 standards are the key specifications in that direction. Tools based on these standards should not have the above difficulties, but you will have to wait a few years until vendors implement them in their tools. Luckily, the Semantic Web and ontologies are still under development, too, so when they are ready for the big scene, proper MDA-based ontology tools will be available.

11.1 MagicDraw

One of the UML tools that can easily be used for ontology development is MagicDraw. This section gives a short tutorial on how to use it to create ontologies using the Ontology UML Profile.

The MagicDraw core is based on the NetBeans Metadata Repository, which means that it correctly supports serialization to UML XMI for version 1.4 of the UML standard. The current version (9.5) also supports UML2, but not at the core level (the same applies to other tools today).

So, let us start.

11.1.1 Starting with MagicDraw

Like almost all other popular UML tools, MagicDraw is written on the Java platform, so it can be run on any major operating system. Of course, you have to install the Java virtual machine if you have not done so yet. MagicDraw is available in several editions, some of them can be used free of charge.¹ To start, you could try the Community Edition, which is free and supports only class diagrams, which is what you need to work with the OUP anyway. Get it from <http://www.magicdraw.com> and follow the installation procedure.

After opening MagicDraw, you will see the usual structure of the user interface that almost all UML tools follow (Fig. 11-1), so it should not be hard to accustom yourself to it. On the upper left, there are various navigation trees, which enable you to have control over your model’s elements. The lower left corner contains a few property windows, where you can set detailed properties of your elements and their graphic

¹ We are not affiliated to MagicDraw’s creators in any way, so, please, if you have any question related to this tool, ask it at <http://www.magicdraw.com>.

representations on diagrams. The central part is the working area, where you can draw diagrams of your ontologies.

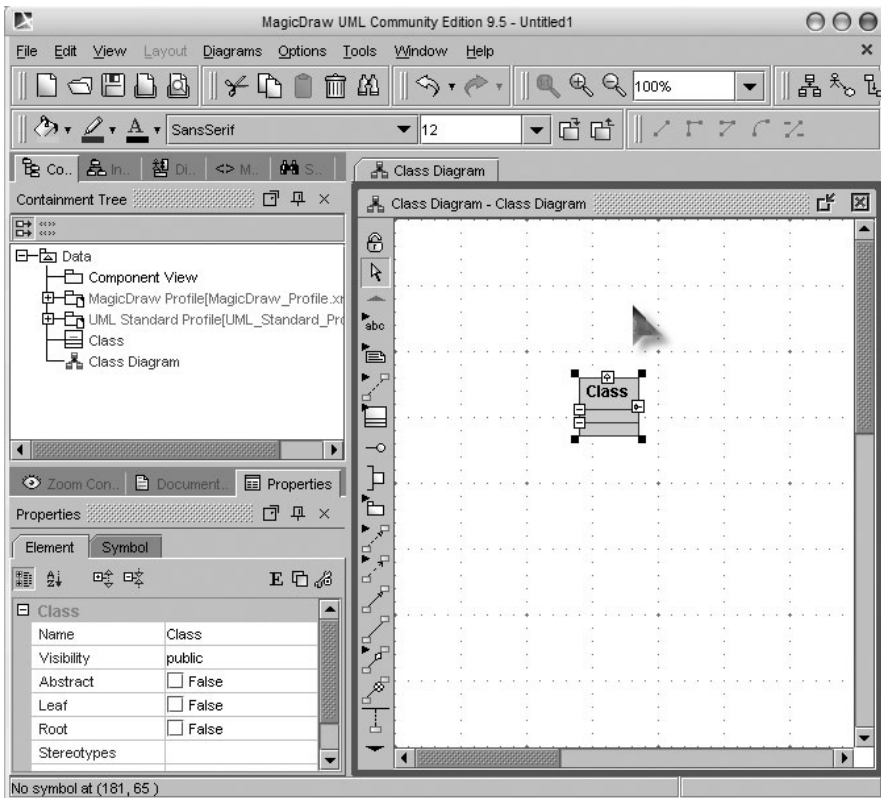


Fig. 11-1. MagicDraw – main parts of the main screen

When you start a new project, the most useful views on the left side are:

- The containment tree, where you can see the packages that your model contains. If you work with ontologies, those packages will represent various ontologies,
- The inheritance tree, which is useful if you want to see the inheritance of your model's elements,
- The diagrams tree, without which it would be very hard to browse in a project with many different diagrams. UML defines eight more kinds of diagram, but you will need only class diagrams to work with the OUP.

Below these views, you can find Properties view, which contains two tabs:

- Element, where you can set properties for your model's elements,
- Symbol, where you can determine what a shape that represents this element looks like on a particular diagram.

UML tools, of course, offer many more views, but we have mentioned only a minimal set that you simply have to use when you are working with UML models. An additional piece of advice when you are looking for some particular functionality is to try selecting an element that you need to work with and invoke the context menu with a right click. The two options in the context menu most often used will probably be "Specification" and "Symbol(s) properties". The best way to get used to the tool you have chosen is to move around it and try as many options as you can on a simple model to see what can be done.

11.1.2 Things You Should Know when Working with UML Profiles

We have already given you an introduction to UML profiles, so you know the theory. Now we shall teach you how to use it with real UML tools.

Basically, it is pretty simple. You use profiles mostly by adding stereotypes and tagged values to ordinary UML model elements. Most tools will let you create a new stereotype on the fly, i.e. they will let you add any string you like as a stereotype of your element. This can lead to a mess in your model. Some other tools will at least try to make you more responsible, so that you must create a new stereotype first and then add it to the element. MagicDraw is such a tool. However, the best choice is to create a reusable profile that contains all necessary elements and then use it as a template for your projects. We provide such a profile which you can import into MagicDraw.

So, if you have a reusable profile that you can import to your tool, you can start working on your models. If you do not have such a profile, you can create it by adding new elements whose type is the stereotype (by selecting *New element > stereotype*).

To make a profile available for use in your projects, place it into `<magicdraw install directory>/profiles`. In this case, you should place `Ontology UML Profile.xml.zip` in the profiles directory.

Now, when you have made the profile available, the procedure should be simple (see Fig. 11-2):

- Select *File>New Project* from the main menu and create an ordinary UML project;
- Select *File>Use profile/module* from the main menu to select the Ontology UML Profile from the list of available profiles.
- MagicDraw uses a MagicDraw profile and a UML standard profile – you can remove them from your project if you like, to have a less bloated project.

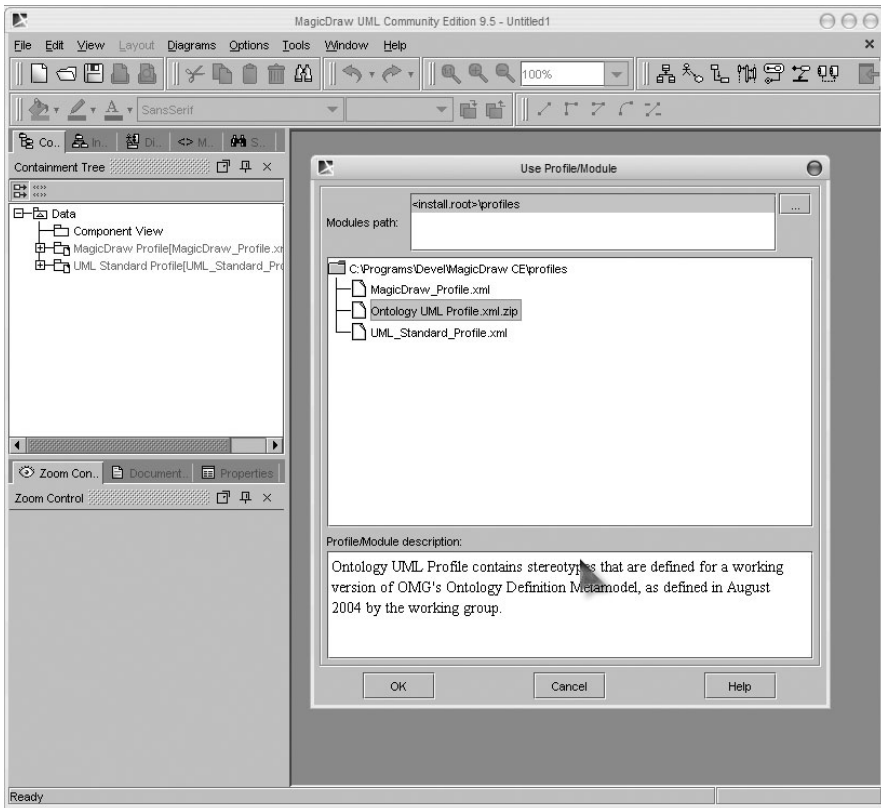


Fig. 11-2. Enabling the Ontology UML Profile in your project

Now, you are ready to work with ontologies using MagicDraw.

11.1.3 Creating a New Ontology

In the OUP, an ontology is represented by a package with a stereotype «OWLontology». To create the ontology, first create new class diagram and place the package in it. Then, choose Stereotype from the context menu of the package created and choose «OWLontology». This procedure is shown in Fig. 11-3.

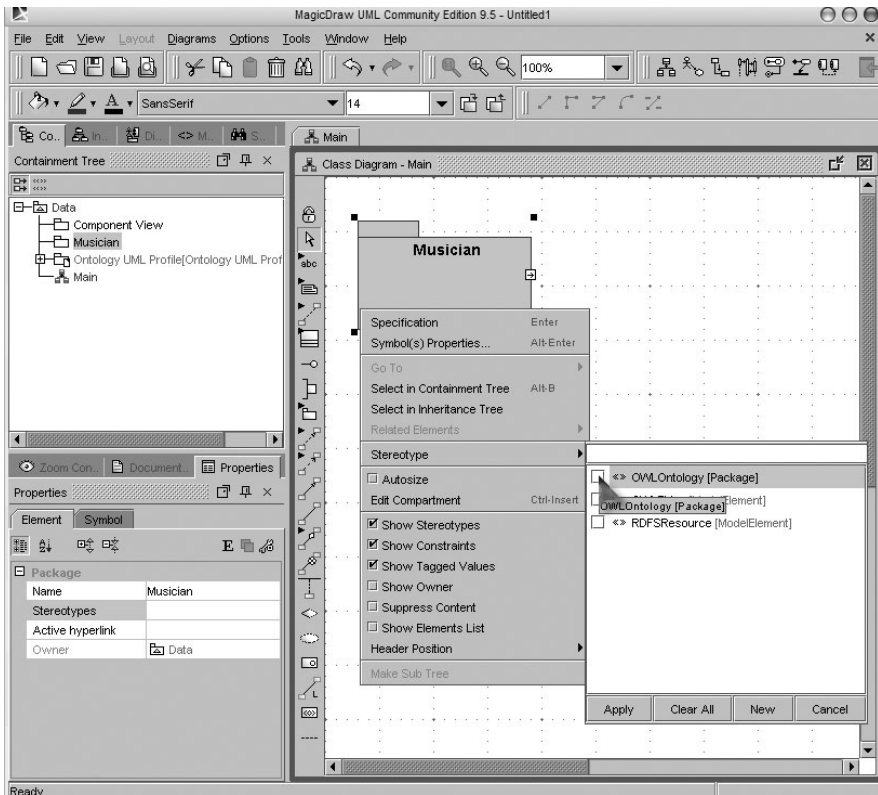


Fig. 11-3. Creating a new ontology

Now, you can use the Specification dialog to set some properties of the Musician ontology (Fig. 11-4), or the Symbol(s) properties to customize visual appearance of the ontology in the diagram (Fig. 11-5).

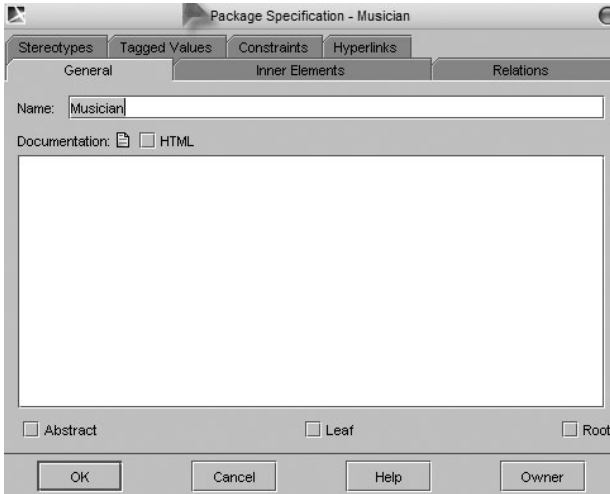


Fig. 11-4. Package specification diagram for the Musician ontology

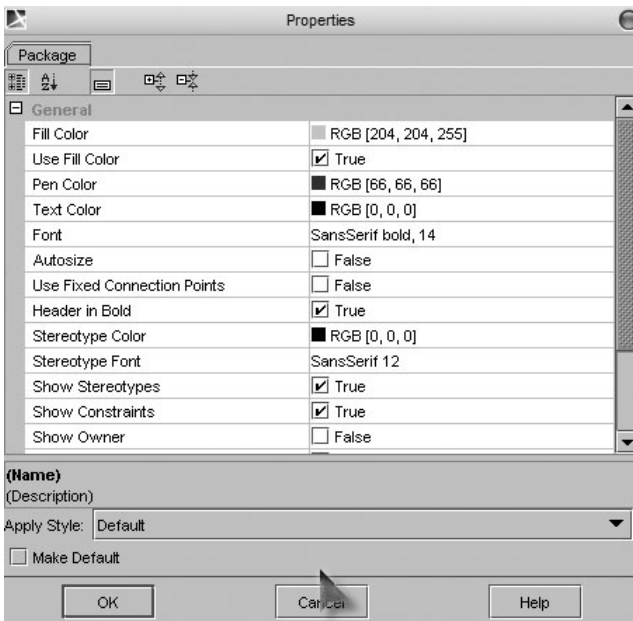


Fig. 11-5. Symbol properties for the Musician ontology

Of course, one model element («OWLOntology») can be placed in multiple diagrams, each time with different details. You can also use diagrams to show the relations of this element with other ontologies using «owlImports», «owlPriorVersion», «owlBackwardCompatibleWith» and

«owlIncompatibleWith». The easiest way to create such relationships is to drag the dependency between the selected packages and apply an appropriate stereotype to it (see Fig. 11-6).

Models (without diagrams) are intended to be read by machines, whereas diagrams are for you and your colleagues. Therefore, the symbol properties are not unimportant, and so you should choose carefully the colors and fonts and align the various elements in the diagrams.

The diagram shown in Fig. 11-6 shows that elements in the Musician ontology use elements in the People ontology. These elements could be an ontology class or property in Musician that extends some class or property in People, or a class in Musician that is connected with a class in People, or two classes in Musician connected via some property in People... There are countless possibilities. Similar logic applies to imports into UML: if you need something from a package (ontology) - import it.

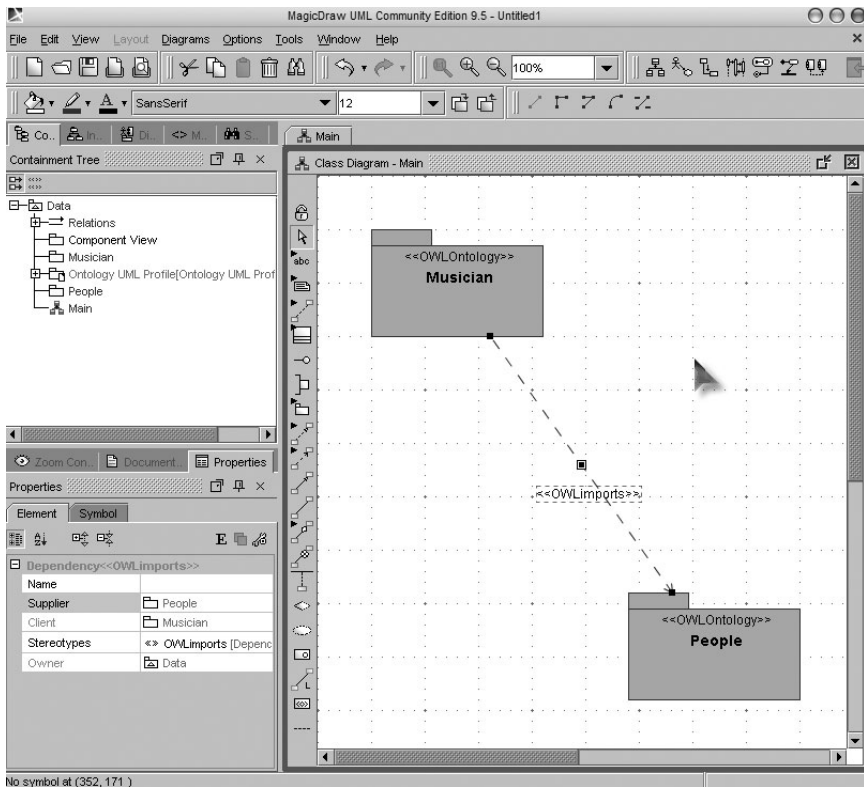


Fig. 11-6. Connecting ontologies – the Musician ontology imports the People ontology

11.1.4 Working with Ontology Classes

When you are creating an ontology, you will usually start by identifying concepts (entities), i.e. ontology classes. As we have already seen, ontology classes in the OUP are modeled using UML classes with a stereotype «OWLClass».

When creating a class, you should put it in an ontology. It would be awkward if your classes just wandered around. So, you should select the Musician ontology and choose *New Element* > *Class* (Fig. 11-7). The new class will be created in the Musician package (ontology). Of course, you should add a stereotype once when your new class has been created to denote that it is not an ordinary UML class, but an ontology class.

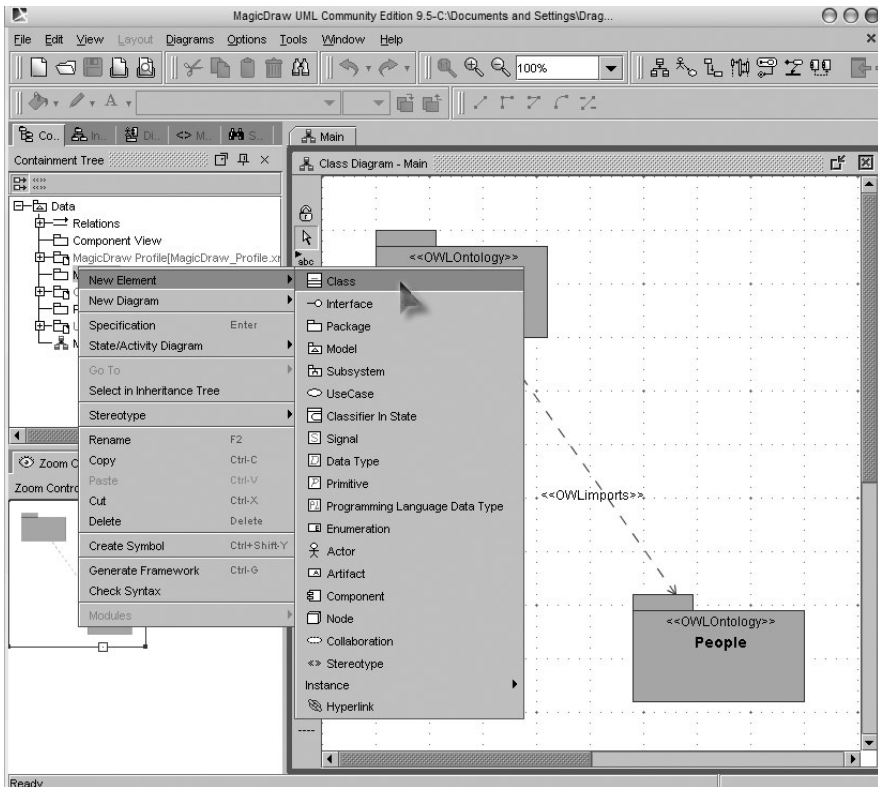


Fig. 11-7. Creating a new ontology class inside an ontology

You will surely notice that the class created is not visible in any diagram. Why is that? As you will probably remember, UML is not just about pictures, it also defines objects as representations of reality. So, the

class that you have just created, let us say this is a class that represents a musician, which belongs to the Musician ontology, exists as an entity, but is not visually represented on any diagram. You do not have to represent it on a diagram at all, but you probably should show it on one or more diagrams, with more or less details (not all details have to be represented).

As we do not want to place all elements in a Main diagram, we need to create at least one diagram per ontology (package). So, select the Musician package and create a new class diagram in it. For simple ontologies, one diagram should be enough. In the real world, you will probably need many diagrams per ontology, and many ontologies for the domain of your problem. To make the Musician class visible in a diagram, simply select it and use the drag-and-drop technique to place it on the diagram, as shown in Fig. 11-8. A box named “Musician”, with a stereotype `<<OWLClass>>` will be created.

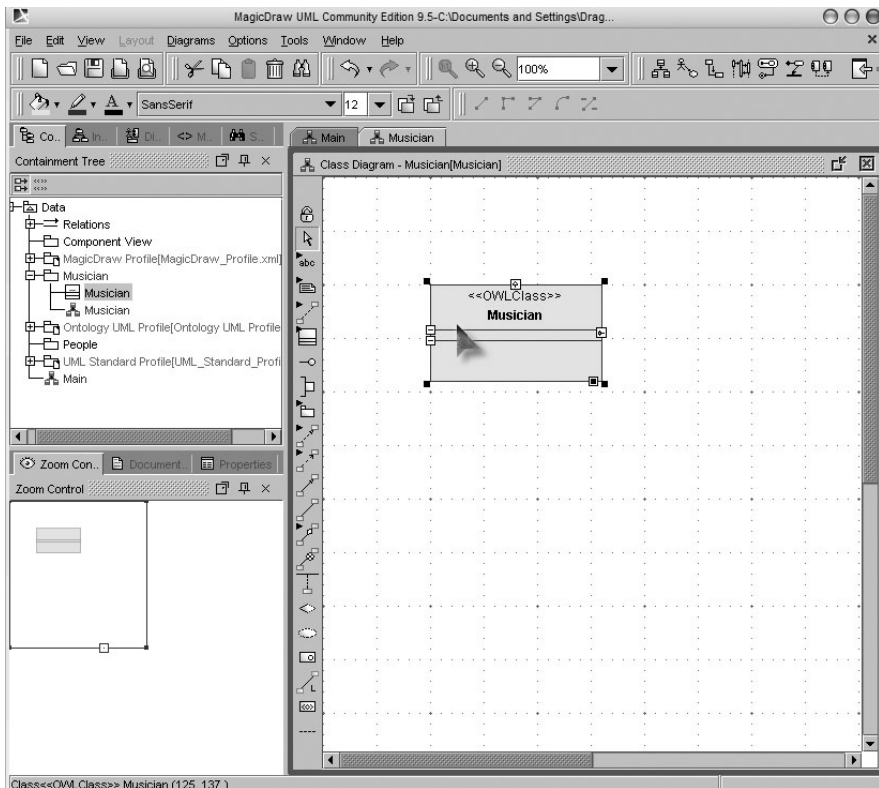


Fig. 11-8. Adding an existing element to a diagram

Almost all UML tools allow you to change details of the shape of boxes and lines. We recommend that you choose a different color for each of the main groups of elements. For example, all classes in our diagrams are light green², to make them distinct from individuals (yellow), properties (light brown) and other concepts. Of course, you do not have to use this color scheme, but it is wise to stick to some rules of your choice to make your diagrams readable when your project grows.

If we add more element shapes to a diagram, the underlying elements will be created automatically. They will be nested in the package where the diagram is placed. For example, if we create the ontology classes Instrument, Album, Event and Admirer by placing them in the Musician diagram, they will appear in the subtree of the Musician package (Fig. 11-9).

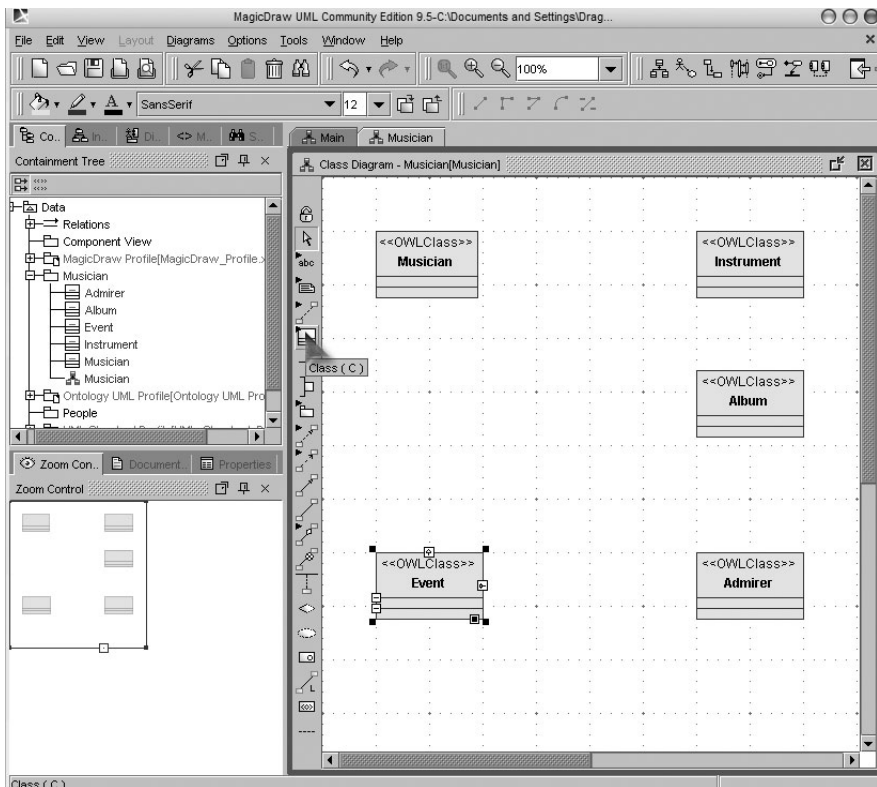


Fig. 11-9. Adding new classes to a diagram

² Figures in this book are printed in black and white, so you will see only different shades of gray.

11.1.5 Working with Ontology Properties

Now that we have placed the main classes on our Musician ontology diagram, we need to attach some properties to them. Without properties, they are just unrelated, isolated islands.

Owing to the significant difference between ontology properties and the UML elements most similar to them, namely attributes and associations, properties are represented by a stereotyped class and stereotyped associations that connect the property to its domains and ranges. When a property is simple (for example, a musician's name), it can be represented as a stereotyped attribute.

You can create properties in the same way as you create classes since they are represented in the same way – as stereotyped UML classes. Simple properties are represented as «OWLDataTypeProperty», while relationships between ontology classes are «OWLObjectProperty». Figure 11-10 shows some properties from the Musician ontology placed in the Musician diagram.

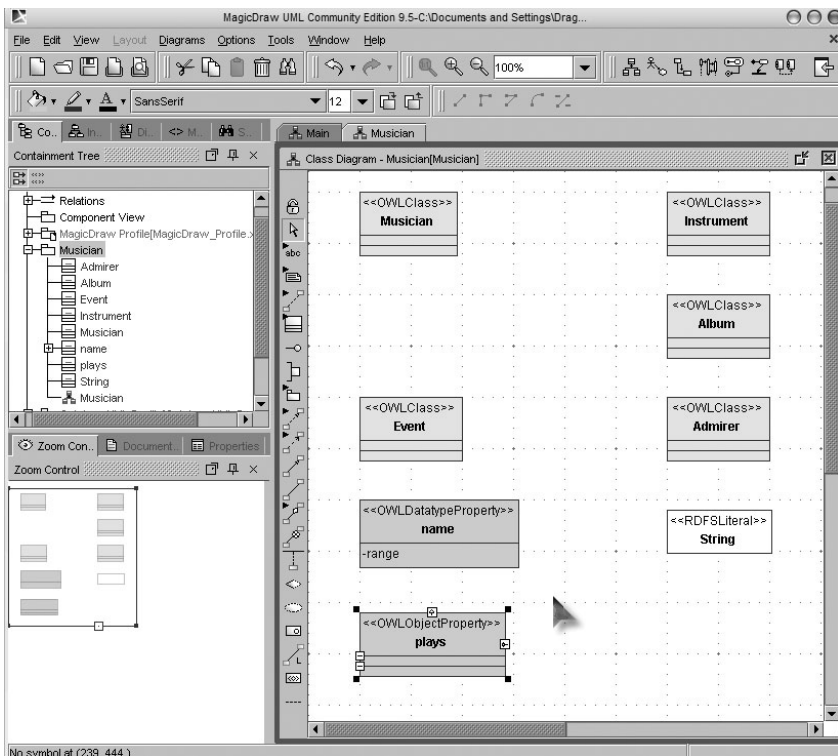


Fig. 11-10. Creating new properties

These properties are useful, but are not complete. Their main purpose is to connect entities, so we must add ranges and domains to them to make these connections.

If the property is a datatype property (a simple property in the object-oriented terminology), we can specify a range as an attribute of an «OWLDatatypeProperty» stereotyped «RDFSrange». Figure 11-11 shows the dialog that is shown upon opening the class specification of a name.

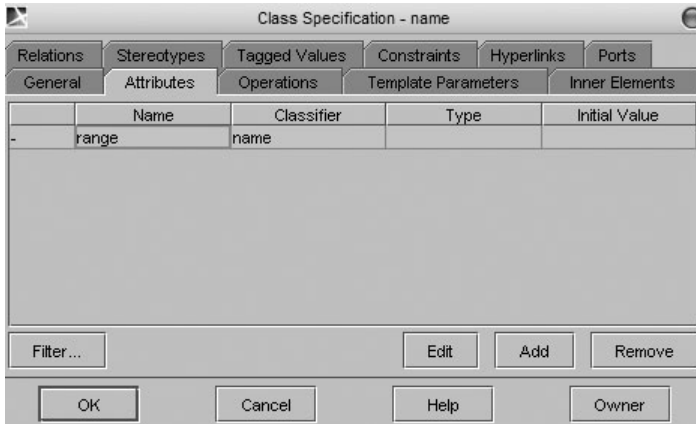


Fig. 11-11. Specification of «RDFSrange» of a name property

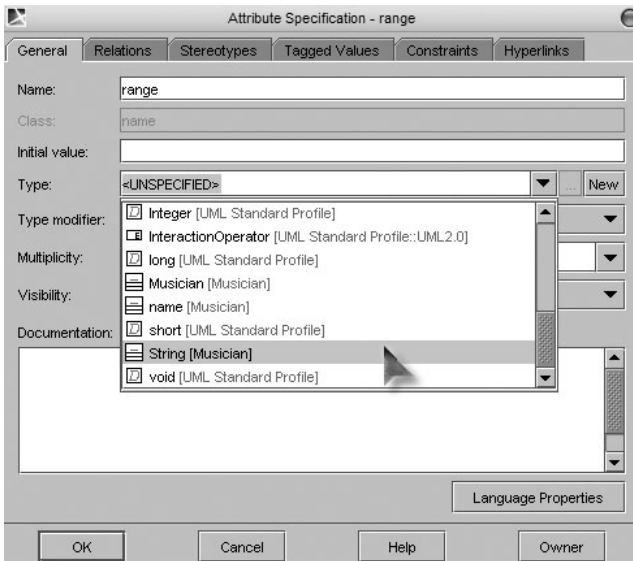


Fig. 11-12. Setting the range of a property

After the range has been created, a click on the Edit button will show the dialog with the attribute's details (Fig. 11-12). We need to select the type of the attribute, which in this case is String, an «RDFSLiteral». Of course, if this type has not been set, we should skip to the Stereotypes tab and select a stereotype of the attribute - «RDFSrange».

We have now set the range of the property, but we still have not connected it with its domains, the classes that it describes. To attach a simple property (DatatypeProperty) to a class, select its representation on a diagram and choose the Insert New Attribute option from a context menu. To make the attribute different from an ordinary attribute, set «RDFSdomain» as the attribute's stereotype (Fig. 11-13). We now have added a name attribute to both Musician and Instrument, set both attribute's stereotype to «RDFSdomain» and set the type to name.

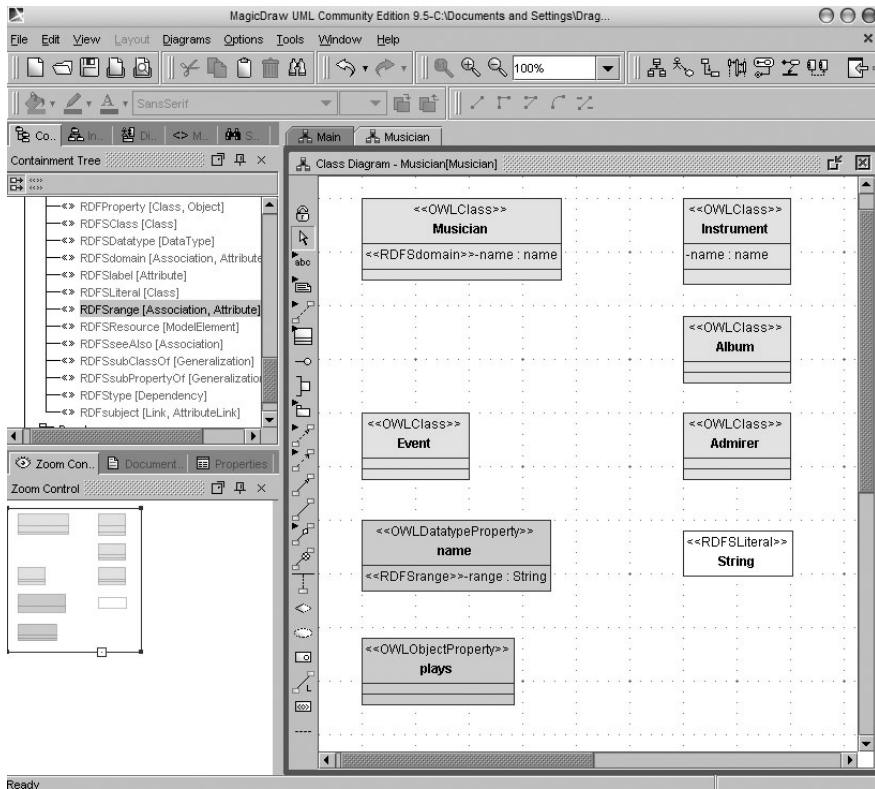


Fig. 11-13. Setting the domain of a datatype property

So, in both cases, the name is the same property (represented by a stereotyped UML class), whose domain and range are represented by different corresponding stereotyped UML attributes.

The majority of the important relationships will, however, be object properties. The example that we shall consider is the “plays” relationship. This is represented by «OWLObjectProperty», too, but range and domain are represented using relationships, not attributes. These associations also have the stereotypes «RDFSdomain» and «RDFSrange» stereotype.

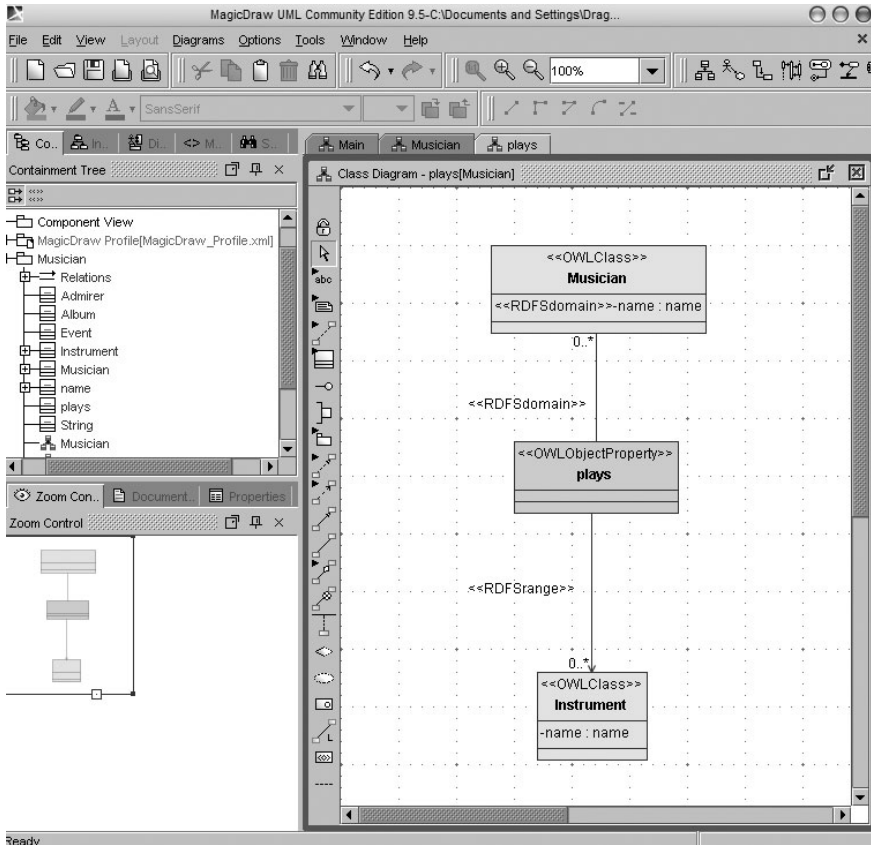


Fig. 11-14. Representing the domain and range of an object property

To avoid clutter in the existing diagram, we shall create a new class diagram for the purpose of representing the “plays” relationship (it is up to you how you arrange elements in your diagrams, of course). The new diagram is empty, so you should drag-and-drop the required existing elements from the containment tree (folder-like view on the left side).

Then, associate Musician and “play” with one association and “plays” and Instrument with another. Add the appropriate stereotypes and multiplicities to these associations and you have created a relationship (see Fig. 11-14).

11.1.6 Working with Individuals

One of the noticeable differences between ontology modeling and object-oriented modeling (i.e. UML) is that you will not see many objects in object-oriented diagrams. Objects are something that is left for runtime, which is not depicted in diagrams very much. However, ontologies care a lot more about specific instances of their concepts. So, it is very likely that you will need to draw some object diagrams when creating your ontologies.

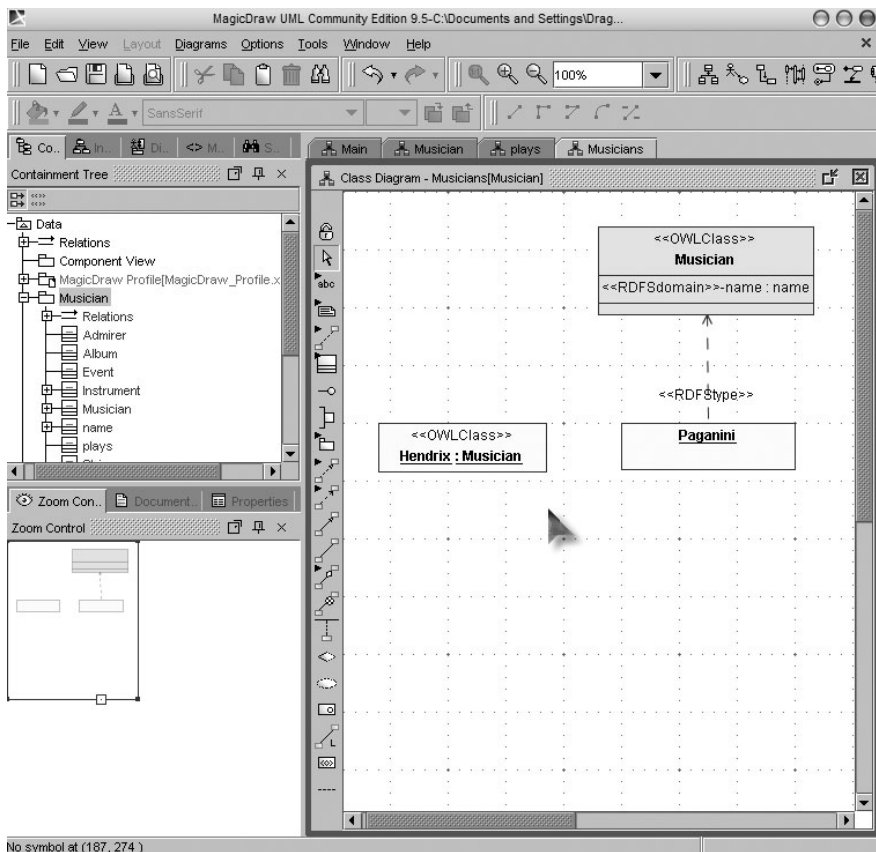


Fig. 11-15. Object diagram allowing to model individuals in MagicDraw

Object diagrams are, in fact, class diagrams that contain objects. While the UML specification allows this, some tools still do not support placing objects in class diagrams. Fortunately, MagicDraw does.

Placing an object in a diagram is as simple as placing a class. The object is symbolized by a box. The difference from the symbol for a class is that the object's name is underlined and the box does not have compartments for attributes and operations (although these compartments can be omitted in class symbols, too). So, we have created a new class diagram, named it "Artists", and placed a few objects in it (Fig. 11-15).

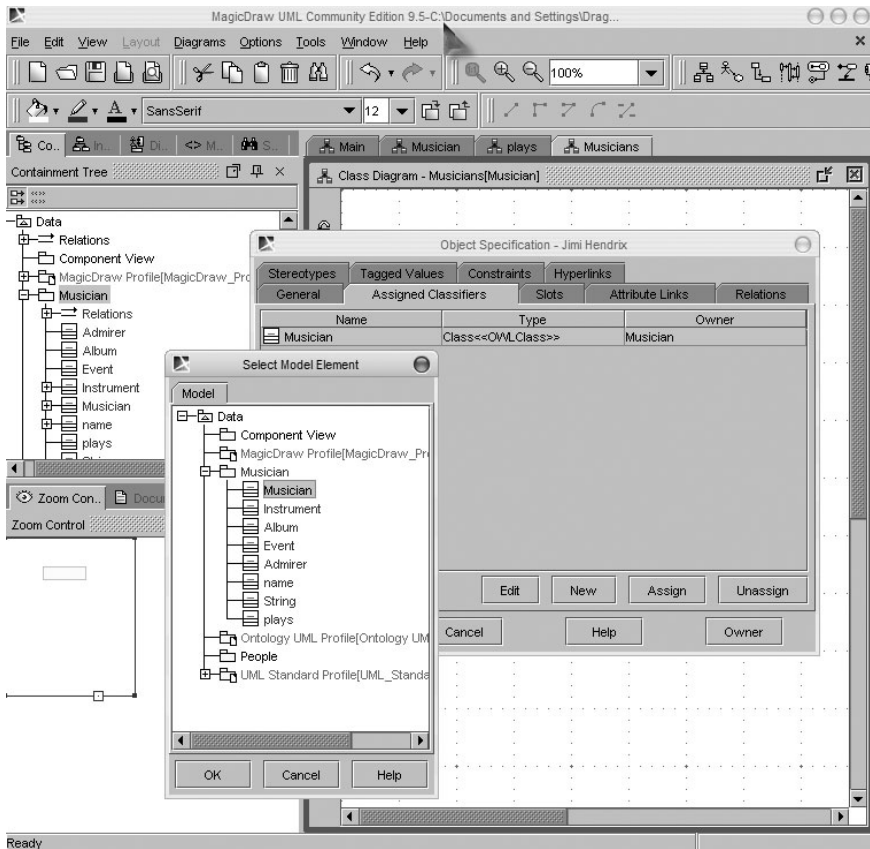


Fig. 11-16. Assigning a type to an individual

Most often, the type of an object is shown after its name (Hendrix:Musician) or through «RDFtype» dependency (Paganini).

To assign a classifier (type) to Hendrix in this diagram, go to its specification, choose the Assigned Classifiers tab and click on the Assign

button. Then navigate the tree that pops up and select Musician (Fig. 11-16). Objects have the same stereotypes as their classes, so if a class has a stereotype, it will be shown in the object's symbol (e.g. we have added the «OWLClass» stereotype to the Hendrix object).

To use the alternative approach, of an «RDFStype» dependency, just drag-and-drop the desired class onto same diagram where the object is and connect them with a dependency (shown by a dashed arrow symbol). Assign an «RDFStype» symbol to the dependency and you have assigned a type to an individual.

11.1.7 Working with Statements

What instrument did Jimi Hendrix play? And Niccolo Paganini? Jimi Hendrix was a famous rock guitar player in the 1960s. His playing style was so distinctive and advanced, that he is often considered the greatest guitar player in the history of rock music. More than a century ago, Niccolo Paganini's virtuosity on the violin gained him the status of the greatest violinist of all time. His technique was so technically perfect that it looked impossible. Some thrilled listeners even claimed that the Devil himself guided Paganini's hands.

To describe what instruments they played, we must add new individuals, Electric Guitar and Violin, to the diagram. These are objects whose type is Instrument. Then, we must create two statements: *Hendrix*>*plays*>*Electric Gitar* and *Paganini*>*plays*>*Violin*. A statement is represented by an object whose type is a class that represents property, which is associated with the subject and object of the statement. So, we must create two statements of type *plays* and connect them to the appropriate individuals using links with «RDFsubject» and «RDFobject» stereotypes. The procedure for doing this in MagicDraw is similar to the procedure applied for creating individuals, and the result is shown in Fig. 11-17.

Hopefully, this short tutorial will help you when you are at the beginning of your work with the Ontology UML Profile and MagicDraw. As you discover this tool, you will find that most UML tools are similar, and that you can quickly familiarize yourself with other tools, as well with other UML profiles. If you have any in-depth questions about advanced use of UML Profiles, you will probably find answers in the general literature, or in literature that deals with other UML profiles, or even in the literature about the tool you use. This is one case where we all can feel benefits from the standards.

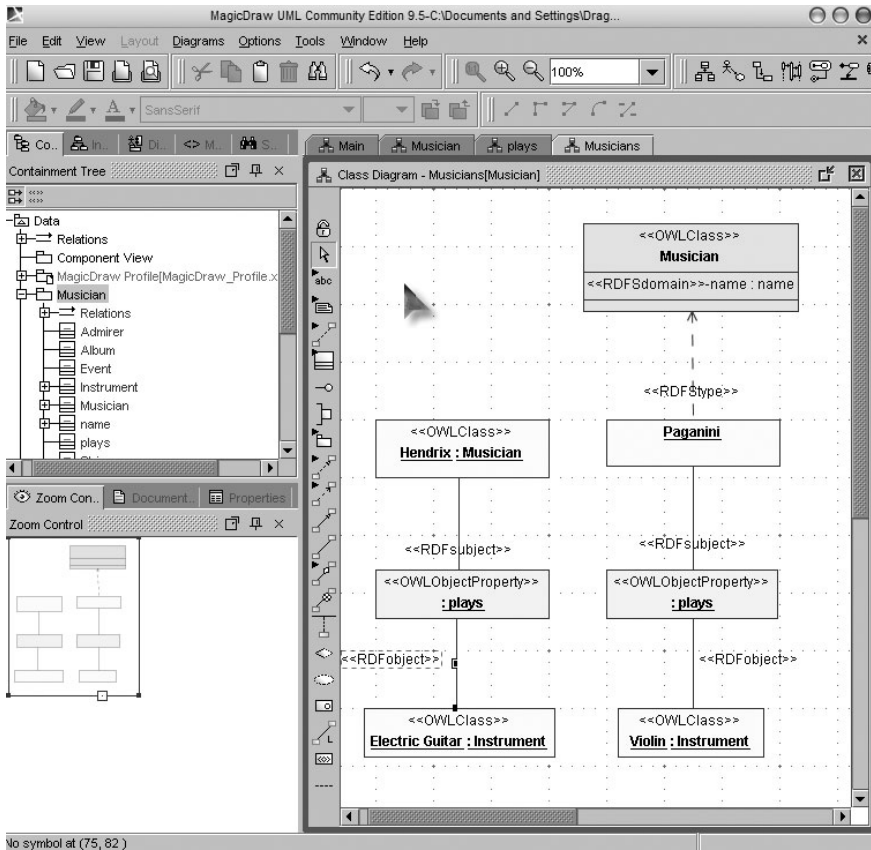


Fig. 11-17. Creating statements

11.2 Poseidon for UML

Another UML tool that can be easily adapted to ontology development is Poseidon for UML. Since Poseidon for UML (Poseidon, for short) is quite similar to MagicDraw with respect to the user interface, this section will just give a short overview of the features that have been used to develop ontologies using GOOD OLD AI Ontology UML Profile. We assume that the readers of this section have already read the section on MagicDraw.

Similarly to MagicDraw, the Poseidon core is based on the NetBeans Metadata Repository, which means that it correctly supports serialization to UML XMI for version 1.4 of the UML standard. The current version (4.0) also supports UML2. For developing ontologies in Poseidon with the

OUP, it is sufficient to use the Community Edition, which is free. One can download this software from <http://www.gentleware.com>, and then follow the installation procedure.³ Poseidon projects can be saved into two formats: UML XMI, and ZUML (Zipped UML), which is actually a compressed (zipped) archive containing the UML XMI project together with other Poseidon-specific files.

Poseidon is written on the Java platform, and so it can be run on any major operating system. Of course, you have to install the Java virtual machine if you have not done that yet. The organization of Poseidon's GUI is similar to that of most UML tools (see Fig. 11-18). In the upper left corner, there is a pane with a navigational tree that can provide various perspectives to the UML model under study: Class Centric, Diagram Centric, Inheritance Centric, Model Index, Package Centric, and State Centric. The default perspective is Package Centric, and in most cases this is the most suitable one for developing ontologies. A Poseidon UML project may contain many more packages, and each package may have a stereotype (e.g., the ontology stereotype). The lower left pane gives a Bird perspective of the UML diagram shown in the work area.

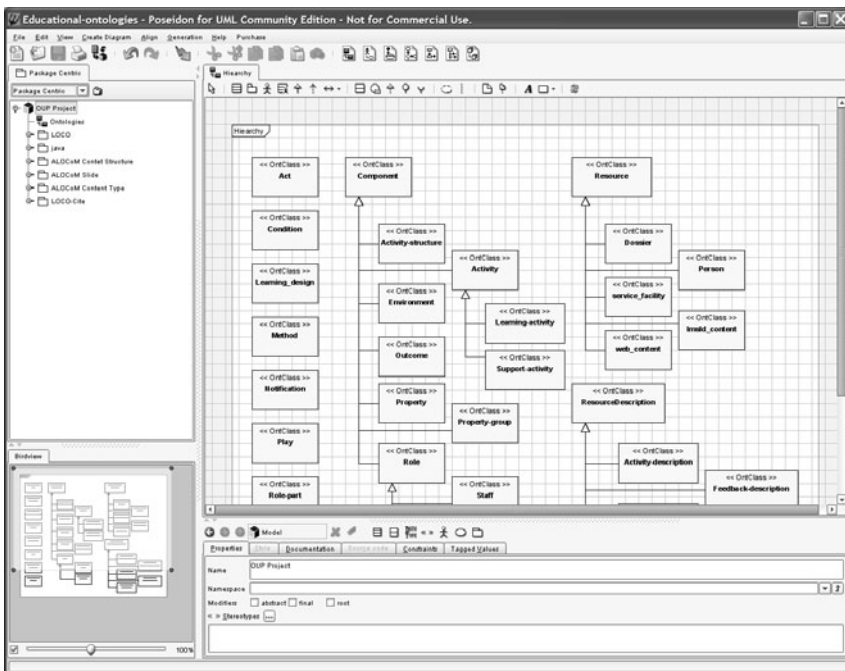


Fig. 11-18. The graphical user interface of Poseidon for UML

³ For any question related to this tool, please ask at <http://www.gentleware.com>

The most important part of Poseidon's GUI is the upper right pane, which is actually the working area, where you can draw diagrams of your ontologies. The working area shows the current UML diagram. You can open more than one UML diagram at the same time and all open diagrams are shown as separate tabs in the working area. Finally, the lower right pane has several tabs specifying features of a model element (e.g., a class, object, or association) selected in either the working area or the navigational tree, namely Properties, Style, Documentation, Source code, Constraints, and Tagged Values. The most important tab is Properties, which you use to specify various characteristics of model elements such as name, namespace, stereotype, and multiplicity.

The biggest difference between MagicDraw and Poseidon is that you cannot define a UML profile as a separate unit in Poseidon. In fact, you have always to create the necessary stereotypes in every new Poseidon model. In order to avoid such a rather annoying repetition of actions, we have developed an empty Poseidon project containing all OUP stereotypes, which is available on the Web page carrying the supplementary resources for this book (<http://www.modelingspaces.org>). After opening this project in Poseidon, you will have an empty package whose stereotype is "ontology", and also all of the stereotypes of the OUP.

11.2.1 Modeling Ontology Classes in Poseidon

To model ontology classes in Poseidon, you use UML class diagrams. You create diagrams in the navigational tree by selecting the target UML package (ontology) and selecting the menu option *Create Diagram > Class Diagram*. Otherwise, you can create a class diagram through the context menu (which you open with a right click). Note that the same class can be shown in many UML class diagrams, since the diagrams are notational features, whereas classes, associations, and other elements are parts of UML models. At the top of the current UML class diagram, there is a toolbar containing all model elements that can be represented in a UML class diagram. Drawing on UML class diagrams is done in a way similar to that for MagicDraw. The difference is that we always use the property pane (lower left pane) to define characteristics of model elements. For example, to define a stereotype for modeling the ontology class (OntClass), you select the stereotype button, and then the stereotype dialog is shown (Fig. 11-19).

You should add one of the available stereotypes (left) to the list of selected stereotypes (right). You should use the same procedure to define stereotypes for the other model elements that you use for modeling

ontologies (object, association, package, and dependency). All of the recommendations regarding the use of colors given in the section on MagicDraw are also applicable to Poseidon.

11.2.2 Modeling Ontology Individuals and Statements in Poseidon

To model individuals and statements in an ontology with Poseidon, you use UML object diagrams like as you do in MagicDraw. The notation and rules for drawing individuals and statements are the same as those described for MagicDraw, so we shall not give details about Poseidon in this section. However, there one important difference: you cannot draw UML objects in UML class diagrams in Poseidon, nor can you draw UML classes in UML object diagrams, even though it is allowed in the UML2 specification. Since it is very important for modeling ontologies to represent relations between UML classes and UML objects (e.g., the definition of enumerations as shown in Fig. 11-20), you should use UML deployment diagrams to represent this kind of relation.

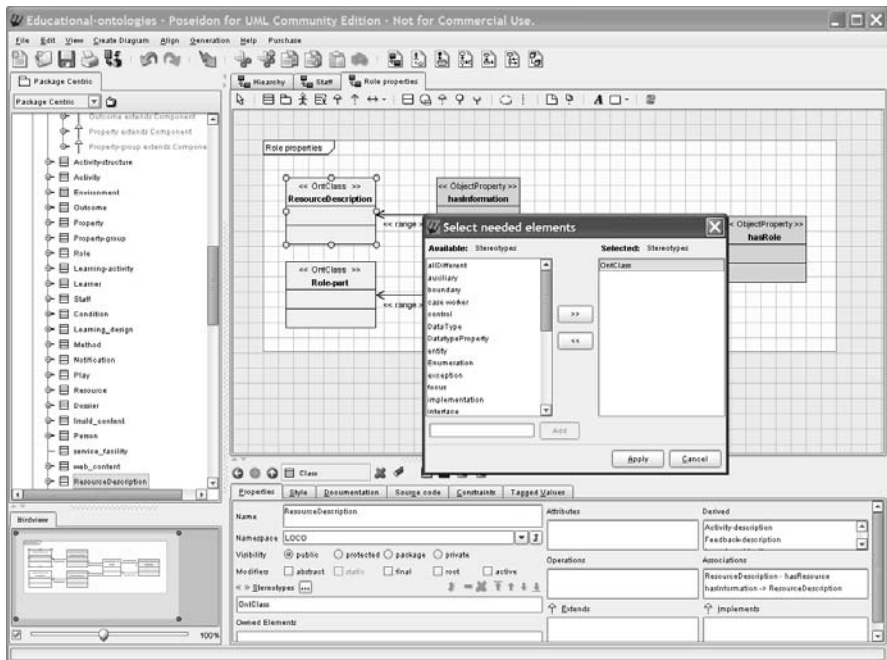


Fig. 11-19. Defining properties of UML model elements: the stereotype property

11.3 Sharing UML Models Between UML tools and Protégé Using the UML Back End

We have already described Protégé's UML back end, which allows the sharing of UML models between UML tools and Protégé by using the UML XMI format. In this section, we describe the practical steps that should be undertaken to exchange UML models and ontologies between UML tools and Protégé.⁴

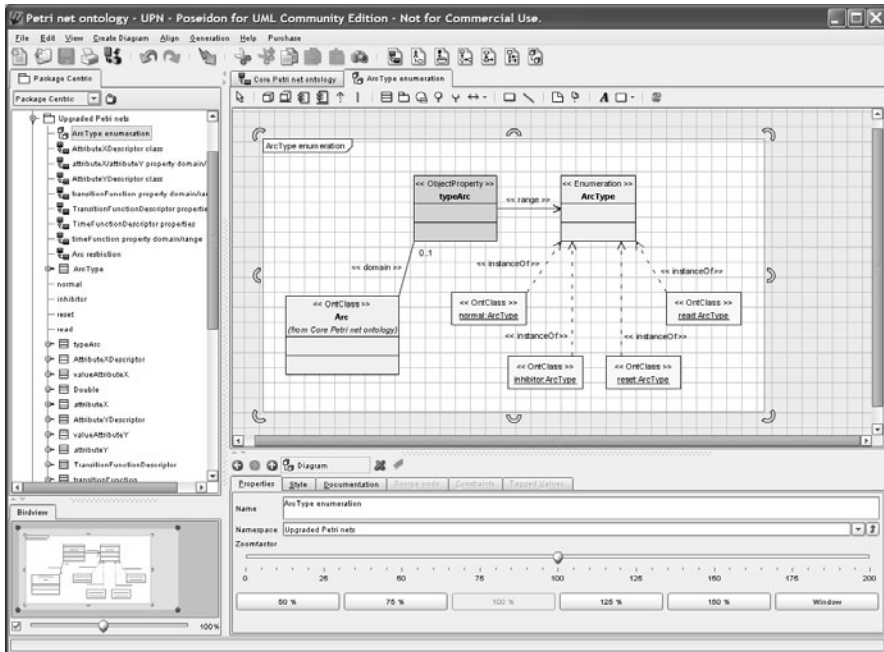


Fig. 11-20. UML deployment diagrams for representing relations between UML objects and UML classes in Poseidon

In order to import a UML model into Protégé, you take the following steps:

- Create an XMI file with your CASE tool (we give details of how to do this with selected tools below).

⁴ UML back end users' guide at <http://protege.stanford.edu/plugins/uml/use.html>.

- Choose *Project > Build New Project > UML 1.4 Class Diagrams*⁵ in Protégé, and click OK.
- In the subsequent dialog box, click on the + button near the *UML file name* text box.
- Select the XMI file and click OK so that the name of the XMI file appears as the UML file name.
- Click OK to start the import process.

In order to import a UML model developed in Poseidon, you should either save your project as an XMI file using the menu option *File > Export Project to XMI* or save your project as a ZUML file and then extract the XMI file using a zip archive extractor (e.g., WinZip). In Fig. 11-21, we show the Musician ontology in Poseidon, which we saved as an XMI file. By applying the steps listed above, we have imported this UML model into Protégé, and resulting Protégé ontology is shown in Fig. 11-22.

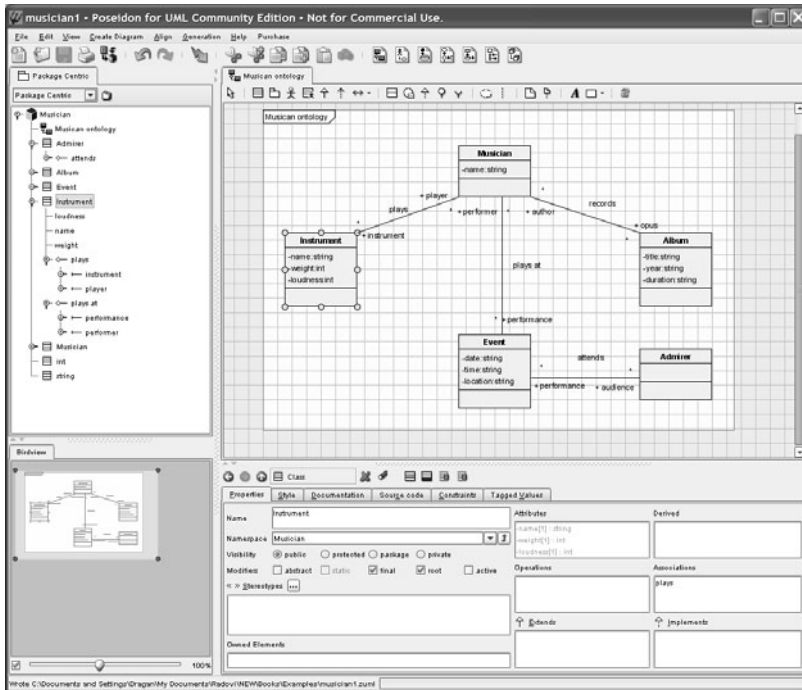


Fig. 11-21. The Musician ontology in Poseidon for UML, which we have imported into Protégé

⁵ Remember that we are loading UML models, not UML class diagrams. The wording used by Protégé is not correct.

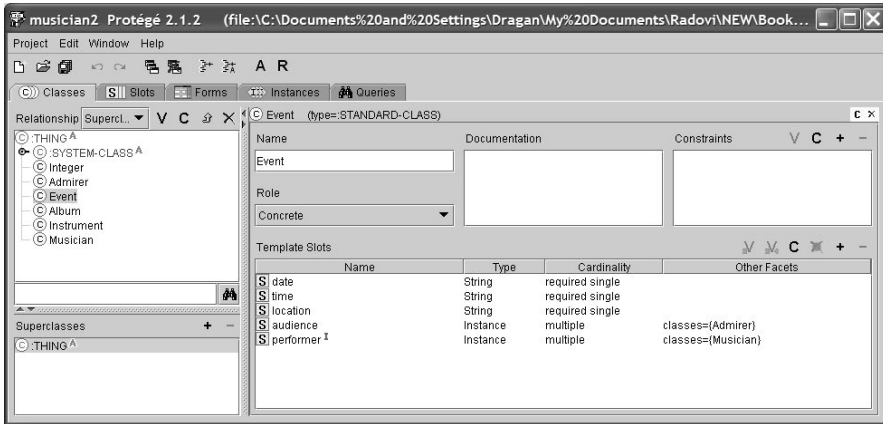


Fig. 11-22. The Musician ontology after importing the UML model shown in Fig. 11-21 into Protégé

However, in trying to perform this operation, you might face the problem that your XMI files generated in Poseidon cannot be imported into Protégé. When we analyzed this problem, we realized that only XMI files that do not contain UML diagrams can be imported into Protégé. This means that, before exporting Poseidon models into XMI, you should first delete all the diagrams.⁶ In our experiments we used Protégé 2.1.2, since the UML back end was compiled for this version, even though the current Protégé version is 3.2. We have used versions 2.0 and 3.2 of Poseidon successfully in the previous example.

To import a UML model developed in MagicDraw, you can save your project as either an XMI.ZIP file, which needs to be unzipped, or as an XMI file. Note that the extension of the XMI file generated by MagicDraw is .XML. Such an XMI file can be imported into Protégé using the above procedure. Unlike the case for Poseidon, the UML models generated by MagicDraw can be imported into Protégé even if they contain UML diagrams. In our experiments, we used MagicDraw Community Edition version 9.5.

Finally, if you want to import a UML model developed in IBM Rational Rose, you should first save your UML model in XMI version 1.0 using the XMI add-in (option *Tools > UML 1.3 XMI Addin > Export*), and then import it into Poseidon to obtain a valid UML 1.4 model. Such a UML model can be imported into Protégé.

⁶ Be careful – delete only the UML diagrams, not the UML model elements.

Once you have imported your UML model into Protégé, you can create an OWL ontology using the Protégé menu option *Project > Export to Format > OWL*.

In order to perform the opposite procedure of exporting a UML model from Protégé, you should take the following steps:

- Save your Protégé project in your usual format (e.g., OWL) before you begin the export.
- Choose the *Project > Convert Project to Format* menu item in Protégé.
- Select *UML 1.4 Class Diagrams*⁷ as the output format and click OK.
- Choose a new project name (take care not to overwrite the current project).
- Click OK to save the project into the specified XMI file.
- Since Protégé will reload the new project after storing the XMI file (this might modify your current knowledge base), you should reopen the original project if you want to continue working with it in the state that you had prior to the export procedure.

In our experiments with Poseidon 2.0 and 3.2 and MagicDraw 9.5, we could import Protégé ontologies exported as XMI files. Note that the newly created UML classes may not be assigned to any UML class diagram in your target UML tool. In some tools such as Poseidon, apparently the only option for seeing the new UML classes after import is to drag the imported classes from the explorer panel into the target model. A similar procedure must be applied in other UML tools, which can be a time-consuming and very hard task, especially when you are dealing with a big ontology.

Finally, we advise you to look for further details about the Protégé back end at its web site [Protégé UML, 2004]. Unfortunately, you must be prepared for many challenges when using this back end, as it is not being regularly maintained to keep up with new versions of Protégé and UML tools.

⁷ The wording is again wrong! You are converting a Protégé ontology into a UML model, not a UML class diagram.

12. An MDA Based Ontology Platform: AIR

In the past few years, software engineering has witnessed two major shifts: model-driven engineering has entered the mainstream and some leading development tools have become open and extensible¹. AI has always been a spring of new ideas that have been adopted in software engineering, but most of its gems have stayed buried in laboratories, available only to a limited number of AI practitioners. Should AI tools be integrated into mainstream tools and could it be done? We think that it is feasible, and that both communities can benefit from this integration. In fact, some efforts in this direction have already been made, both by major industrial standardization bodies such as the OMG, and by academic laboratories. In this chapter, we present an example of such an idea that should inspire readers to think about the integration of their work with the tools that are already in the mainstream.

12.1 Motivation

The basic problem of the existing environments for the development of intelligent systems is their narrow specialization. Most of them have been implemented to support only the functionalities initially envisioned— most often knowledge representation and reasoning. This is perfectly right from the intelligent systems point of view. However, real-world applications and their development are rarely clearly bordered in their scope; that's why these environments are not enough. It is, therefore, necessary to integrate applications that are used for intelligent systems development into mainstream software platforms. This topic is going to gain more and more attention with the development of the Semantic Web [Berners-Lee et al., 2001] and with increased integration of intelligent techniques in common information systems.

In order to design and develop a reliable, robust, well-architected, and easy-to-extend application or tool in any field, it is important to conform to

¹ Portions of this chapter have been reprinted with minor changes, with permission, from [Djurić et al., 2006a] ©2005 IEEE.

sound principles and rules of software engineering. Intelligent systems are no exception. It is especially important that AI development tools are designed closely following SE practices.

Keeping an eye on current developments and trends in software engineering can help design AI tools to remain stable over a longer period of time. Some trends in software engineering are general and span many, if not all fields and application domains. Others are more specific, but can be relevant to AI. For example, a general trend in SE is the use of tailored versions of UML to alleviate design of system specifics, and the large portion of this book was about that approach. Another such trend is that of plug-in architectures, which allow easy and modular extension of tools and systems by use of features targeting specific groups of users. Probably the best known such platform is Eclipse (<http://www.eclipse.org>), a base for top-quality Java and UML tools.

Another emerging SE trend that we have discussed, and which has intensive support from the OMG, is application development based on MDA [Miller & Mukerji, 2003]. MDA is a generally applicable idea, but is simultaneously of specific interest to AI developers since it has much in common with ontology modeling and development [Devedžić, 2002].

This chapter presents AIR [Djurić et al., 2006a], an integrated AI development environment based on MDA modeling concepts. Using the MDA philosophy in AIR makes it possible to employ mainstream software technologies that users are familiar with, and expand these technologies with new functionalities.

The development of AIR was motivated by the fact that although AI has always been a source of ideas that have been subsequently adopted in software engineering, most of these ideas have remained in laboratories. We believe that AI tools should be integrated with mainstream SE tools and thus become more widely known and used.

12.2 The Basic Idea

In order to integrate intelligent technologies with common information system technologies and take such systems out of the laboratory, we must develop appropriate tools. These tools must be easy to use and powerful enough to support creation of demanding applications. The best solutions for such demands are tools that employ mainstream software technologies that users are familiar with, and expand them with new functionalities. This is the idea that is at the root of AIR. AIR is an integrated software environment for developing of intelligent systems that:

- is based on open standards (OMG, W3C...),
- uses existing mainstream software tools and architectures, and
- is extendable and adaptable.

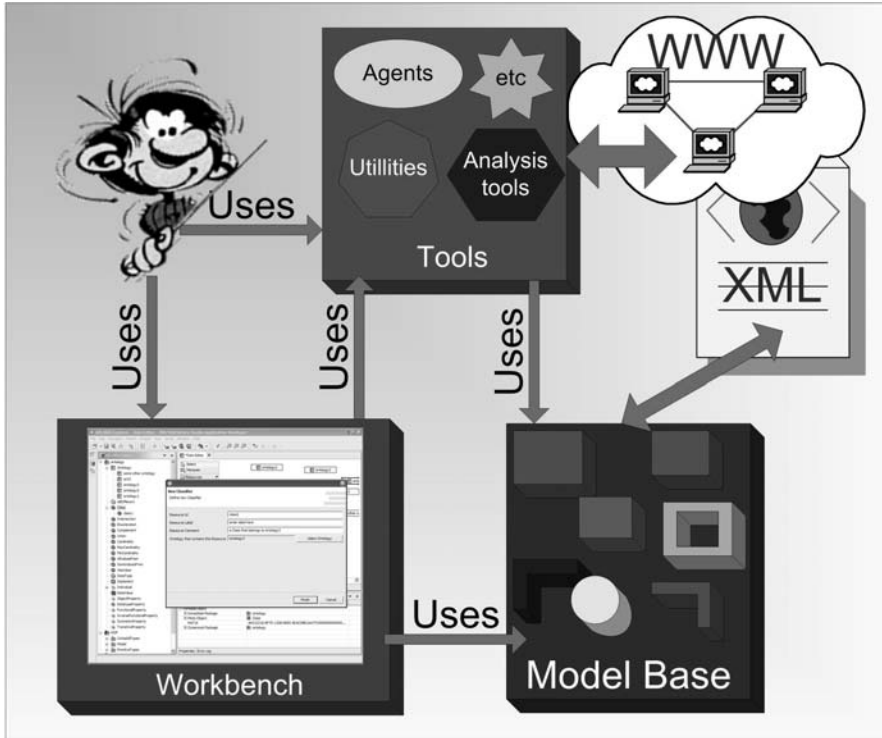


Fig. 12-1. Basic structure of AIR

We could itemize more characteristics, but these three are good examples of what AIR is intended to achieve. It is clear that today's tools must be built according to standards if they are to succeed, so compliance with OMG or W3C standards is a must whenever it is possible. In view of the fact that AIR is an academic project, it is illusory to expect that it can become serious environment if it does not use existing solutions as its base. Such an approach would lead it away from its scope and route it toward reinventing the wheel. Therefore, AIR should use any existing solution that fits into its puzzle. As it is intended to support work with new technologies that are still extremely changeable, AIR must support easy and seamless addition of new functionalities and the replacement of existing parts with improved versions. Many of the technologies that it

aims to support are still in early phase, which means that they are frequently exposed to changes. AIR must be able to follow those changes.

The basic structure of AIR is depicted in the block diagram shown in Fig. 12-1. The central part of AIR is a model base. Prrrrimarily, this includes models of intelligent systems domains, but it also can include models of any domain that is of interest. Currently, the model base is implemented as a MOF-based metadata repository [Dirckze, 2002; OMG MOF, 2002]. It contains MOF-based metamodels and models which are the core of AIR. AIR must have a mechanism that enables the exchange of contained data with other applications. This is achieved through the MOF XML Metadata Interchange (XMI) format, based on XML. Such a format also enables easier integration with the Web.

The other important part of AIR is an integrated development environment that provides a rich GUI for manipulation of models – the AIR workbench. This part of the AIR implementation is based on an Eclipse platform. The model base can be also reached from numerous tools that are not part of the GUI – agents, analysis tools, utilities, etc.

12.3 Metamodel – the Conceptual Building Block of AIR

AIR should be able to model a large number of domains. The real world consists of an infinite number of concepts and facts, which we are trying to describe using models. Models are described using metamodels, models of models [Seidewitz, 2003]. Modeling and metamodeling are well-known terms in software engineering, and standardization in this field recently started to gain more attention. AIR uses a four-layer MOF-based metamodeling architecture in accordance with the OMG's MDA standards. The basic metamodels that AIR uses and their place in this architecture are shown in Fig. 12-2.

In the center of this architecture is the Meta-Object Facility (MOF), a meta-metamodel used for the modeling of all metamodels. Beside the Unified Modeling Language (UML) and the Common Warehouse Metamodel (CWM), metamodels usual in such an architecture, the important metamodels are the Ontology Definition Metamodel (ODM) and the Rule Definition Metamodel (RDM). For more specific purposes, such as Petri nets, fuzzy logic, or neural nets, specialized metamodels can also be included. Such metamodels should be added only if the existing models, the ODM for instance, lack support for some of wanted features.

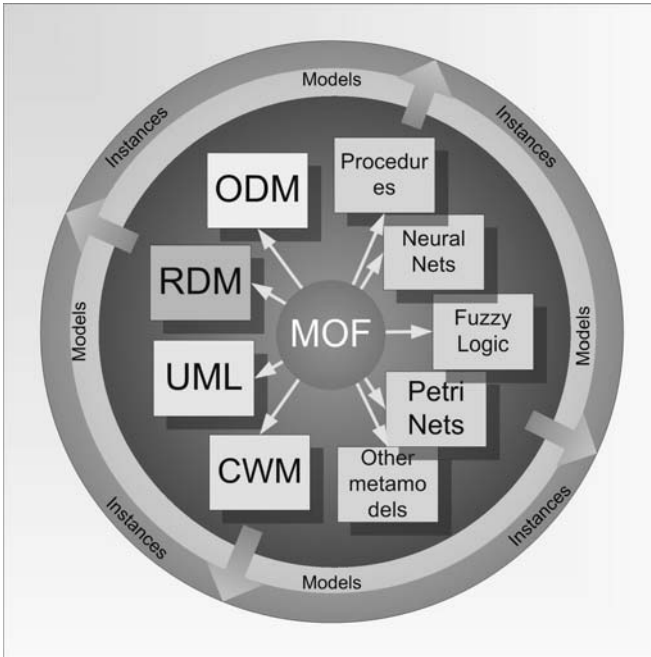


Fig. 12-2. AIR metamodelling

The basic building block of AIR is metamodel. A metamodel enables the domain of a problem to be described, e.g. it supports creation of models that describe certain specific problems in that domain. The place of metamodels and their accompanying elements in the MDA architecture are shown in Fig. 12-3.

A metamodel is described by the MOF, and it is located in the M2 (metamodel) layer. To utilize the widespread support for UML tools a corresponding UML profile can be added. This UML profile is an expansion of the UML metamodel in a standard way that enables UML to support new concepts. It is possible to add mappings to other UML profiles or metamodels, which enables several kinds of model to be used to model one problem, where each model is capable of capturing some specific details of the problem.

12.4 The AIR Metadata Repository

The heart of any well-designed application is its domain model. AIR stores its domain models in the model base, which is implemented as an MOF-based repository. The MOF specification also defines a framework for

implementing repositories that hold metadata (e.g., models) described by metamodels. Standard XML technology is used to transform metamodels into a metadata API, giving the framework an implementation. Figure 12-4 shows an overview of an MOF repository and its implementation in Java.

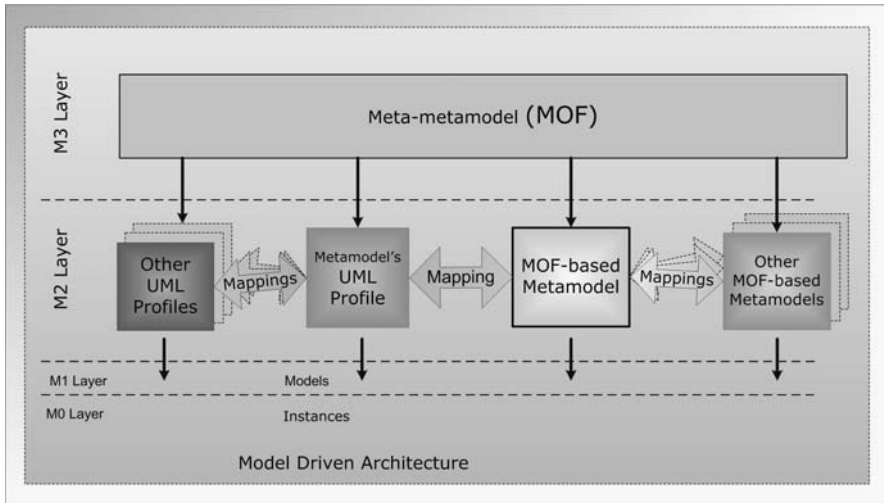


Fig. 12-3. A custom metamodel and UML profile

A conceptual view of an MOF-based repository is shown in the center of Fig. 12-4. This reflects the MDA four-layer MOF-based architecture [Miller & Mukerji, 2003]. The MOF metamodel is used as a meta-metamodel, which describes all other metamodels (including the MOF itself and UML). Custom metamodels can define mappings to UML, supported by UML profiles, which enables them to use support provided by UML tools.

The OMG has defined a standard format for platform-independent metadata interchange XML Metadata Interchange (XMI). This serializes MOF-based metamodels and models into plain text (XML), which enables such data to be exchanged in a standard way, and to be read by any platform-specific implementation.

The Java implementation of the repository is based on Java Metadata Interchange (JMI) [Dirckze, 2002], a Java metadata API. On the basis of any MOF-based metamodel (serialized to XMI), JMI-compliant metamodel-specific JMI interfaces can be generated. These interfaces can be used to access Java metadata repository, which is implemented by Java classes. All data from repository can be serialized into XMI and then exchanged with other repositories, regardless of their implementation. It is

only required that they support MOF-based metadata (i.e. that they can “understand” the MOF XMI format).

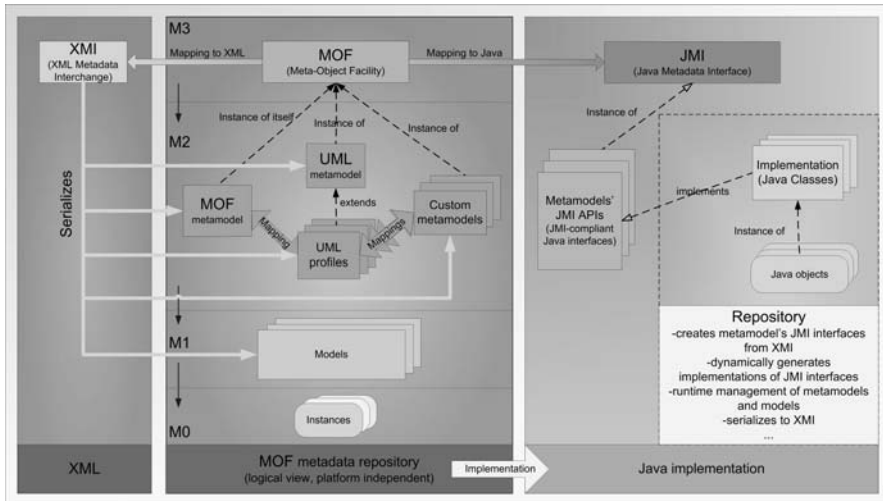


Fig. 12-4. Metadata repository structure (Java-based implementation)

The reference implementation for the JMI metadata repository is Unisys’ CIM (<http://www.unisys.com>), but it seems that it has not been updated recently. The other implementation is NetBeans MDR (<http://mdr.netbeans.org>), a part of the open source NetBeans project. It is used by AIR as a metadata repository due to its generic implementation of JMI interfaces and frequent improvements and development. The NetBeans MDR implements JMI interfaces in a generic way, so any metamodel can be loaded from XMI and instantly implemented using Java reflection.

The JMI specification, which is based on version 1 of the MOF is, however, a little awkward to implement, so there is a competing solution that has lately become more popular – the Eclipse Modeling Framework (EMF). EMF (<http://www.eclipse.org/emf>) is based on ECore – a metamodel similar to the MOF (see Chapter 4 for details), which is simpler and more implementation-friendly. ECore is not a standard specification, but its larger user base makes it an ad-hoc standard.

This multi-standard (JMI and Ecore) situation will hopefully be resolved with the specification of Java representation of version 2 of the MOF standard. MOF2 takes into account the difficulties of implementing the old MOF, and so it defines two separate specifications: CMOF (Complete MOF) and EMOF (Essential MOF), which has the ECore’s simplicity. A

part of AIR, the AIR Metadata Repository project (see <http://www.e-nspire.com> and <http://air-mds.dev.java.net>) is aimed at providing an EMOF-based repository for MOF2-based metamodels implemented using a lightweight Java approach. The lightweight approach is aimed at lowering the dependency on the specific hierarchies or interfaces of standard specifications (such as JMI or EMF) and to use plain Java objects (POJOs) and simple interfaces for implementing metamodels.

12.5 The AIR Workbench

Having a good architecture is an important thing, but a nice graphical user interface is the part that the user deals with directly. The AIR workbench provides various tools with a rich GUI that make the entire workbench user-friendly. This workbench is built on top of the Eclipse plug-in architecture and Eclipse IDE (www.eclipse.org), today's leading extensible platform [Gamma & Beck, 2003]. The main difference between Eclipse and other extensible IDEs is that Eclipse consists entirely of plug-ins that work on a tiny platform runtime, whereas other IDEs are monolithic tools with some extensions. Thus, Eclipse core plug-ins are of equal importance as any other plug-in, including the AIR plug-ins.

Figure 12-5 depicts the Eclipse-based AIR plug-in architecture. Although only the Eclipse Core is mandatory here, there is no reason not to utilize Eclipse UI (SWT, JFace, and Workbench), help and team support, so they are not discarded. Using the entire Eclipse IDE, AIR adds the plug-ins related to the MDR and intelligent systems – generic MDR support (AIR Framework, AIR NetBeans MDR, AIR MDR Core), support for specific metamodel (ODM, RDM, UML, CWM, etc.), and GUI-related support (AIR MDR Explorer). These plug-ins are inserted at extension points defined by plug-ins that are part of the Eclipse IDE. Being treated in exactly the same way as Eclipse native plug-ins, the AIR plug-ins also extend each other and allow future plug-ins to extend them.

A screenshot from the AIR Workbench is shown in Fig. 12-6. The Explorer shows MOF-based models and metamodels graphically and serves as a starting point for model manipulation. By selecting an element, the user can reach menus specific to that element and perform various actions. These actions range from usual ones (instantiating, deleting, viewing properties etc.) to more specific ones (opening various metamodel specific editors, starting transformations etc.). Owing to the underlying Eclipse architecture, these menus can be easily extended with new items that can initiate new actions.

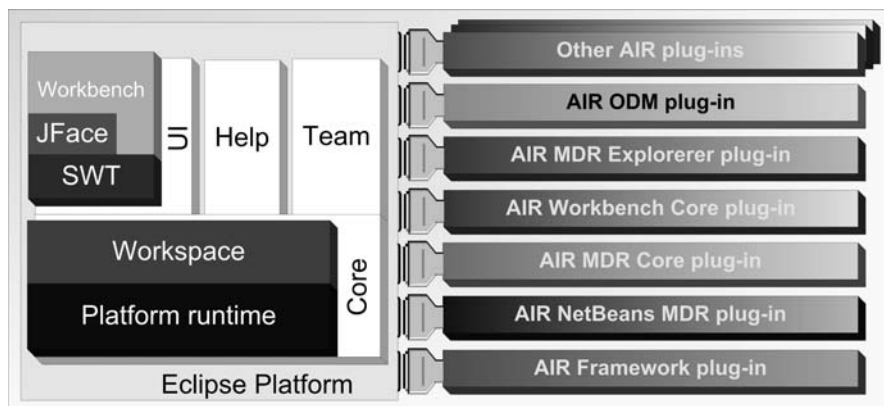


Fig. 12-5. Eclipse-based AIR plug-in architecture

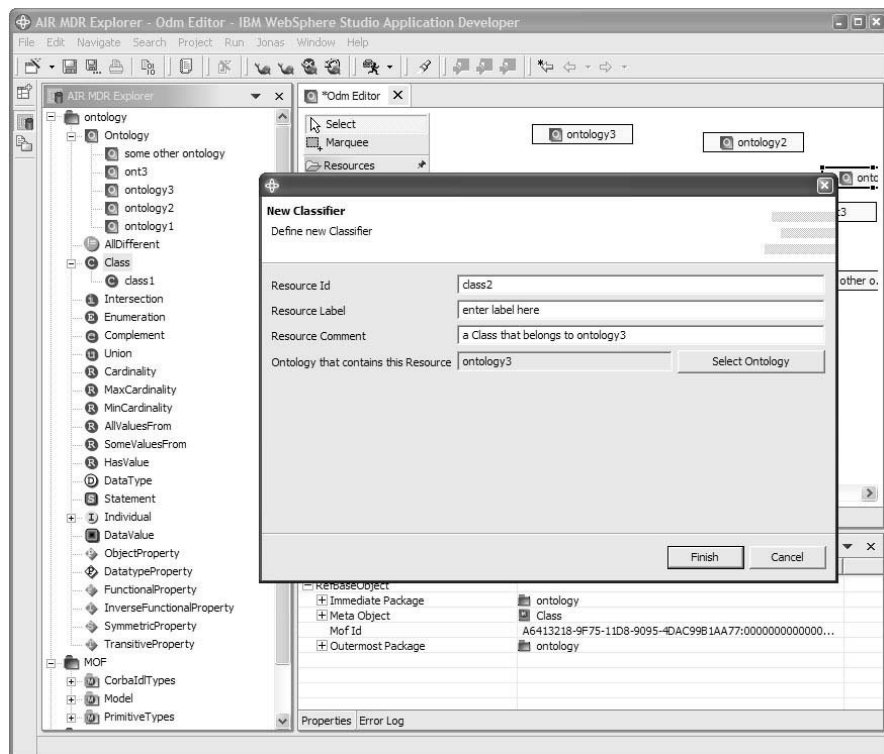


Fig. 12-6. An ontology in the AIR MDR Explorer

12.6 The Role of XML Technologies

The importance of XML technologies is well-known to the AI community, especially after the introduction of the Semantic Web [Decker et al., 2000]. The Semantic Web architecture itself is based on XML. The standard Semantic Web knowledge model (RDF), as well as the language for specifying simple ontology vocabularies (RDFS) are defined on the top of XML. These two standards are the basis for the current W3C Web Ontology Language (OWL) recommendation. Of course, ontology languages are not an isolated example of applying XML in AI. For example, a language for sharing rules on the Semantic Web (RuleML) is based on XML as well. Moreover, there are several AI development tools that define their own XML formats for sharing their knowledge bases (e.g. JessGUI tool [Jovanović et al., 2004] creates XML knowledge bases for Jess, a well-known expert system shell).

In the AIR framework, we use XMI for sharing metadata in MDA. In fact, XMI is not a specific XML format; it is rather a set of production rules that specify how one transforms a MOF-compliant model (i.e. metamodel and meta-metamodel) into a corresponding XML Schema and an XML document. Using this production principle we have a standard way of sharing MDA metadata by XML. Of course, there are a few standard XML Schemas for MOF-compliant models like the XMI schema for UML as well as the XML schema for MOF. However, it is necessary to define XML schemas for every new custom model or metamodel.

Knowing that these two different communities (AI and MDA) both employ XML, we could bridge the gap between them using XML. Since they use different XML formats we should define transformations between them. XSLT is coming as a natural solution to this problem. XSLT is a standard language for transforming XML documents into other documents, either XML or regular text. In Chapter 11 illustrates how we support model sharing between a UML tool (Poseidon for UML that uses UML XMI) and an ontology editor (Protégé, i.e. its OWL plug-in). Applying this XSLT principle we do not have to change (i.e. reprogram and recompile) an existing tool, but we just develop an auxiliary tool (i.e. an XSLT) that extends the existing functionalities. For the further discussion on mappings between ontology and MDA-based languages we refer readers to Chap. 11.

12.7 Possibilities

Bringing AI and SE close together results in well-engineered AI systems with a firm SE backbone. There are a number of possibilities for achieving such a synergy. At one extreme there are disciplined approaches with low "coupling", such as developing and using an API for building AI systems (like in OKBC) or merely using UML-based CASE tools in the process of designing AI systems. At the other extreme,, there are integrated AI development tools. In between, there are other opportunities to add more SE flavor to AI systems development. One can use a suite of tools instead of a complicated integrated tool, or extend the basic tool with a number of useful plug-ins, possibly with the idea of evolving the basic tool into a framework.

Using the MDA, UML, and MOF standards developed by the OMG is yet another possibility. True, it does take some time for AI developers to get used to it. In the long run, it does pay off as well. At its core are standard SE tools and XML technologies that many developers are familiar with. Due to MOF, it enables integration at the metamodeling level, which is related to ontological engineering. It also enables a smooth and gradual transition between the traditional and emerging modeling styles and paradigms.

Integrating AI and SE tools and approaches is something that both sides can benefit from. Yet, academic tools are only a preview of what will be common in the future. Even the MDA has problems in attracting the majority of software developers and being well implemented by tools. Ontologies are also a new approach that is waiting for the spotlight of the mainstream community. One thing is sure: there are lots of similarities in AI (in this case, Knowledge engineering) and SE (in this case, the MDA) approaches and their lifecycle could be parallel. Building tools that benefit from the knowledge and experience of both communities is just a logical sequel.

13. Examples of Ontology

In the previous chapters we introduced the basic concepts of MOF-based languages for developing ontologies, such as the Ontology Definition Metamodel (ODM) and the Ontology UML Profile (OUP). We also discussed mappings between those languages and the OWL language. The purpose of this chapter is to illustrate the use of MOF-based languages for developing real-world ontologies. Here we discuss two different ontologies that we developed in different domains. The first example is a Petri net ontology that formalizes the representation of Petri nets, a well-known tool for modeling, simulation, and analysis of systems and processes. This Petri net ontology overcomes the syntactic constraints of the present XML-based standard for sharing Petri net models, namely Petri Net Markup Language. The second example covers issues in the e-learning domain. The case analyzed provides a foundation for ontologies for bridging between two very important fields in e-learning: learning objects and learning design. Whereas learning objects represent any kind of digital or non-digital content that can be used in technology-supported learning, learning design tries to describe the activities, methods, roles, objectives, and resources in a learning process. By describing both fields with ontologies, we enable the development of Semantic Web services such as searching for a suitable teaching method on basis of the specific content to be taught.

In both examples presented in this chapter we used the GOOD OLD AI Ontology UML Profile to develop the ontologies, because the XSLT that we described earlier, which is provided as supplementary material to this book, can transform such models into an OWL representation. Such OWL ontologies can be imported into the Protégé ontology editor and edited, as we have described earlier when describing the XSLT.

13.1 Petri Net Ontology

Petri nets are formal tool for the modeling, simulation, and analysis of various kinds of systems [Petri, 1962]. These may be distributed systems,

communication protocols, multiprocessor systems, Web services, agent systems, object-oriented systems, or adaptive hypermedia systems, to name but a few of the present uses of Petri net. A Petri net graph consists of two types of nodes: places and transitions. Figure 13-1 shows an example of a Petri net model. In this figure, places are represented by circles, and transitions by rectangles. These two types of nodes are connected by directed arcs, in such a way that only nodes of different types can be connected (i.e., a place and a transition). Each Petri net place may have zero or more tokens (e.g., Buffer' has two in Fig. 13-1). Tokens support the dynamic nature of Petri nets; they are a feature used to simulate systems. The simulation involves the firing of transitions. A Petri net transition (e.g., Put in buffer) can be fired if each of its input places (e.g., ID:p1) has one or more tokens. After the transition has been fired, the tokens are removed from the input places (e.g., ID:p1 and Buffer) of the transition, and tokens are generated in the output places (e.g., ID:p2 and Buffer').

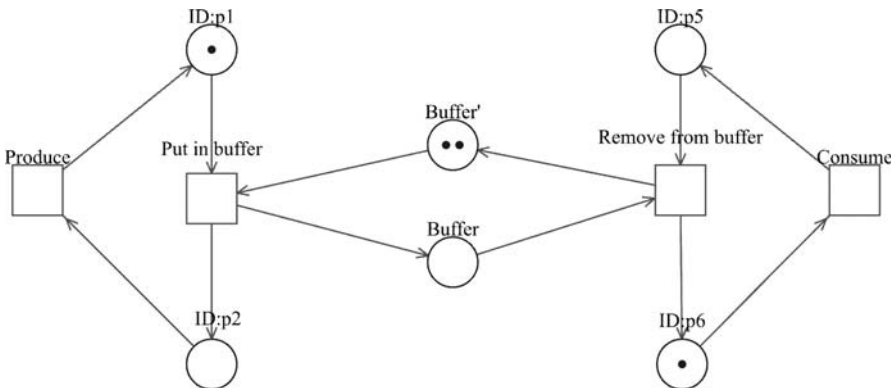


Fig. 13-1. An example of a Petri net model: the well-known synchronization problem of a consumer and producer

Note that Petri nets have evolved from their initial definition given in [Petri, 1962]. One of the most important characteristics of Petri nets is that we have specific extensions for each type of application. Such extensions are called Petri net dialects, kinds, or types. Some examples of Petri net dialects are: time Petri nets, colored Petri nets, object-oriented Petri nets, and upgraded Petri nets. The basic type of Petri nets is that of P/T nets, and they are basic type for other Petri net dialects [Reisig, 1985].

Currently, interoperability between Petri nets is possible at the level of the syntax for model sharing. Petri Net Markup Language (PNML) [Billington et al., 2003] is a result of a recent effort by the Petri net community aimed at providing XML-based model sharing. PNML is intended to be a

part of the future ISO/IEC high-level Petri net standard [Petri Net Standards, 2005]. However, this initiative is also syntactically oriented, i.e., it introduces some constraints that enable the validation of documents against their definition (e.g., validating whether an arc connects two nodes of different types, i.e., a place and a transition). In order to overcome these issues, we have developed a Petri net ontology using the GOOD OLD AI Ontology UML Profile that was later transformed into OWL.

13.1.1 Organization of the Petri Net Ontology

We defined our Petri net ontology using experience from previous formal description of Petri net (metamodel, ontologies, and syntax) [Gašević & Devedžić, 2004; Gašević & Devedžić, 2005]. These descriptions indicate very useful directions for selecting key concepts of Petri nets and specifying their mutual relations. PNML is of primary importance here – it is closely related to the Petri net ontology. We enhanced the usability of PNML by defining mappings to and from Semantic Web languages (RDF and OWL).

Having in mind the extensible nature of Petri nets and of many Petri net dialects, the Petri net ontology has been organized to have a common part that contains concepts common to all Petri net dialects. This common part may be specialized to a concrete Petri net dialect. In fact, this is the same principle that PNML is based on [Billington et al., 2003]. In Fig. 13-2, we show the common part of Petri net ontology, which we call the core Petri net ontology. The core Petri net ontology was extracted from the ontology sources that we analyzed.

We have introduced some concepts that do not really exist in Petri net models in order to obtain a more suitable concept hierarchy in the core ontology. We call these concepts “synthetic concepts.” An overview of these concepts is given in Table 13-1. The meanings of some of the Petri net concepts referred to in the table (e.g., module and structural element) are clarified in Sect. 13.1.2. In that subsection, we also describe how the Petri net ontology has been defined using UML and the Protégé ontology development tool.

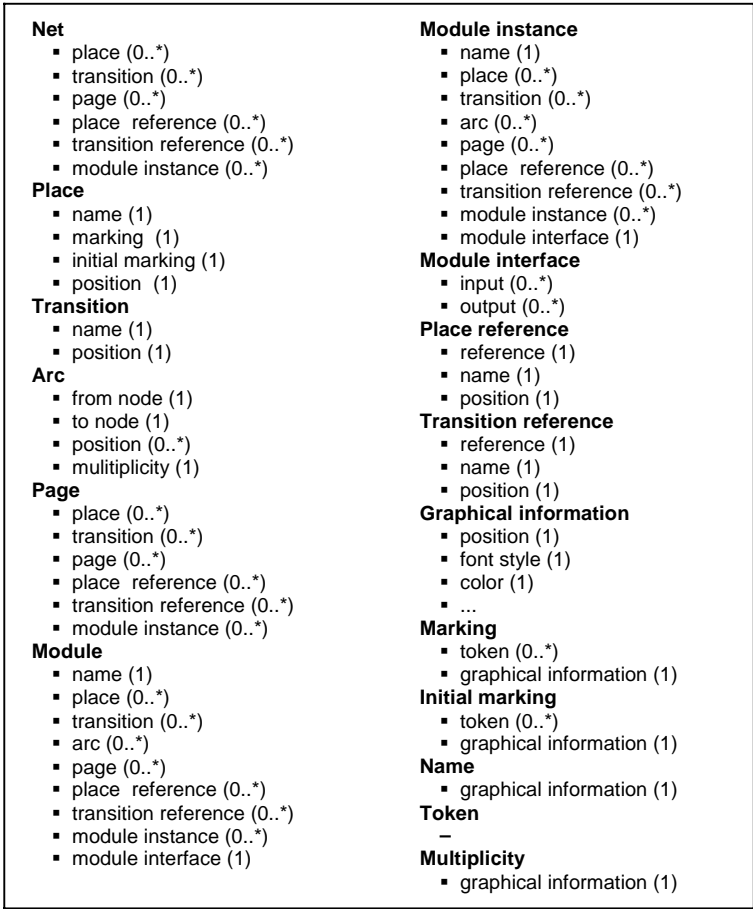


Fig. 13-2. Organization of the core Petri net ontology: key concepts, their mutual relations, and their cardinality

Table 13-1. Overview of the synthetic concepts in the core Petri net ontology, which are generalizations of the concepts in Fig. 13-2

Synthetic concept	Concepts generalized
Node reference	Place reference, transition reference
Node	Place, transition, node reference
Structural element	Page, module instance
Model element	Structural element, arc, node

For the development of the ontology we used the GOOD OLD AI Ontology UML Profile (OUP) given in [Djurić et al., 2005b]. Using this OUP, one can represent relations between the core concepts of the Petri net ontology and the specifics of a Petri net dialect. To do this, we suggest using the OUP's package mechanism. In the OUP, we attach `<<ontology>>` to a package. That means that the package is an ontology. Accordingly, we have put all core concepts of the Petri net ontology in an `<<ontology>>` package. If we wish to extend the Petri net ontology with concepts from a Petri net dialect, we only need to create a new `<<ontology>>`, related to the core `<<ontology>>` through the `<<imports>>` dependency. We illustrate this extension principle in Fig. 13-3.

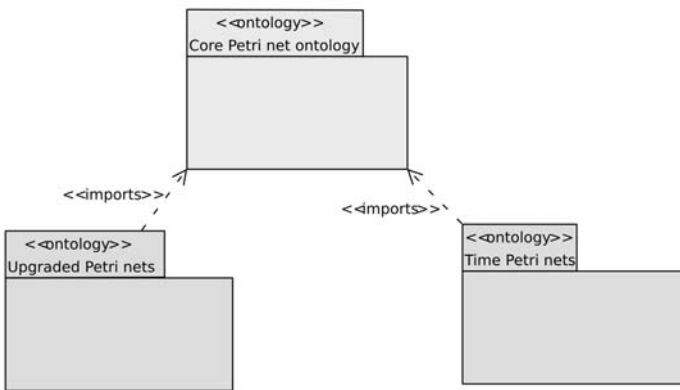


Fig. 13-3. Extension mechanism of the Petri net ontology: support for Petri net dialects

This example depicts how we can extend the core Petri net ontology (`<<ontology>> Petri net core`) with concepts from upgraded and time Petri nets (i.e., we attach new properties to the core classes to describe a Petri net dialect). An additional advantage of this approach is that we have the ability to merge concepts from a number of ontologies (i.e., `<<ontology>>` packages). As a result, we obtain one ontology definition, which can be expressed, for instance, in OWL (by applying an XSLT). Comparing our approach with the current PNML proposal for Petri Net Definition Types [Billington et al., 2003], one can see that our approach improves the maintainability of Petri net concepts, and better supports reusability of the concepts in the Petri net ontology. So far, we have defined extensions of the Petri net ontology for: P/T nets, time Petri nets, and upgraded Petri nets.

13.1.2 The Core Petri Net Ontology in the Ontology UML Profile

In the core Petri net ontology package, we have defined all of the concepts, properties, relations, and restrictions shown in Table 13-1 and Fig. 13-2. The core Petri net hierarchy is shown in Fig. 13-4. We have defined all relations among classes in the Petri net ontology using the `<<ObjectProperty>>` stereotype.

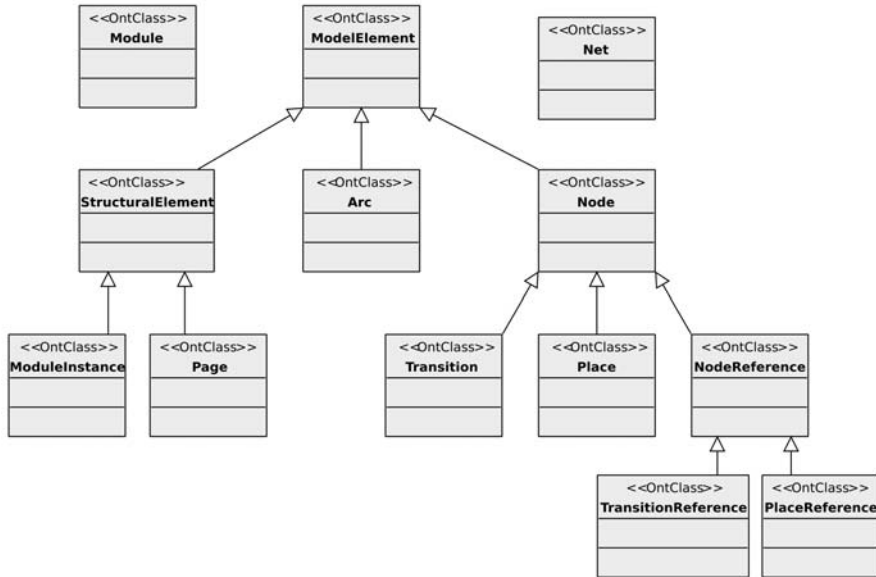


Fig. 13-4. Class hierarchy of the Petri net ontology, shown in the Ontology UML Profile

Figure 13-5 shows how the `<<ObjectProperty>>` *name* is defined as a graphical feature. In this case, the *name* property has as its range (through the `<<range>>` association) the *NameDescriptor* `<<OntClass>>`. However, this class is inherited from *GraphicalFeature*. *GraphicalFeature* was introduced to the Petri net ontology as the root class for all of the classes that constitute the range of a graphical feature. Similarly, we have defined other graphical features (e.g., marking). In addition, the *name* property has a domain *Net* and *Node*.

Figure 13-6 illustrates how we have related the *ModelElement* class to the *StructuralElement*, *Net*, and *Module* classes. In fact, we have defined the `<<ObjectProperty>>` stereotype *element*, whose range is the *ModelElement* class. On the other hand, the domain of this property is the following classes: *Module*, *StructuralElement*, and *Net*.

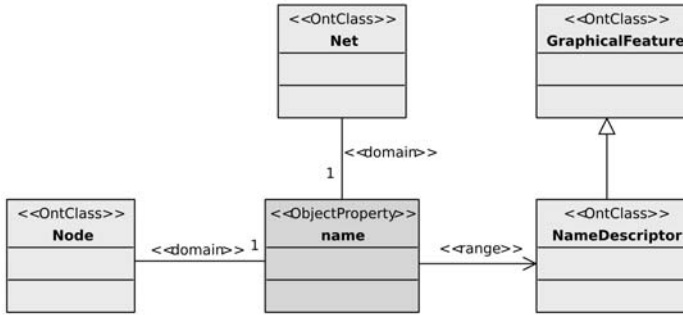


Fig. 13-5. An example of a graphical feature defined in the Ontology UML Profile: the *name* object property

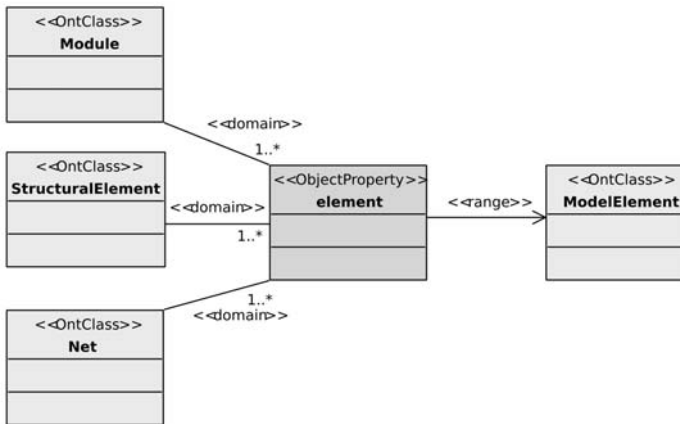


Fig. 13-6. The collections of elements of the Petri net model that contain the *StructuralElement*, *Module*, and *Net* classes, modeled by `<<ObjectProperty>>` stereotype *element*

Figure 13-4 shows that the *PlaceReference* and *TransitionReference* classes are inherited from the *NodeReference* class and inherit its *reference* object property. This property is defined to have its range in the *Node* and *NodeReference* classes. However, the *PlaceReference* and *TransitionReference* classes have an additional restriction on the *reference* property. That is to say, this may only take values from the *Place* and *PlaceReference* classes when it is a property of the *PlaceReference* class. Similarly, it may only take values from the *Transition* and *TransitionReference* classes when it is a property of the *TransitionReference* class. In Fig. 13-7, we show how this restriction is defined for the *PlaceReference* class. Using the `<<allValuesFrom>>` association stereotype, we have defined that the *PlaceReference* class to have a `<<Restriction>>`. The `<<allVal-`

uesFrom>> association stereotype means that a property with this restriction for a specific class must take values from a specific set. Using two dependency stereotypes, we have specified that this restriction is defined on the property *reference* (<<onProperty>> *reference*), and that it must take <<allValuesFrom>> the <<Union>> of the *Place* and *PlaceReference* classes.

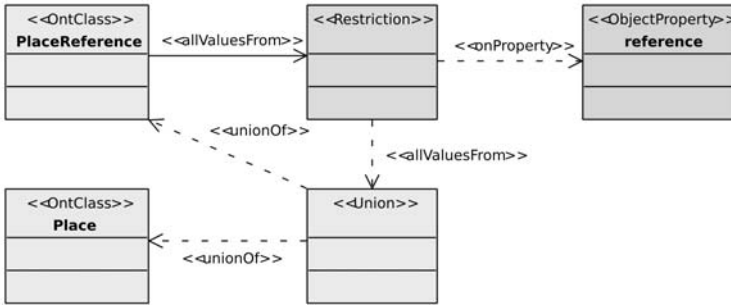


Fig. 13-7. The restriction that the *reference* property of the *PlaceReference* class must have all of its values from the union of the *Place* and *PlaceReference* classes

Figure 13-8 shows an excerpt from the Petri net ontology expressed in OWL. It was generated using an XSLT for transformation from the OUP ontology (i.e., XMI) to OWL. The figure illustrates part of the OWL *TransitionReference* restriction on the *reference* property. This restriction states that *TransitionReference*'s property *reference* must take all of its values from (*allValuesFrom*) the union of the classes *Transition* and *TransitionReference*.

```

<owl:Class rdf:ID="PlaceReference">
  <rdfs:subClassOf rdf:resource="#NodeReference"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#reference"/>
      <owl:allValuesFrom>
        <owl:Class>
          <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Place"/>
            <owl:Class rdf:about="#PlaceReference"/>
          </owl:unionOf>
        </owl:Class>
      </owl:allValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
  
```

Fig. 13-8. Part of the Petri net ontology in OWL: the *TransitionReference* class restriction

13.1.3 Example of an Extension: Upgraded Petri Nets

Here we illustrate an ontology that describes upgraded Petri nets (a Petri net dialect for modeling hardware) [Štrbac, 2002] in order to show how the Petri net ontology can be extended. The same procedure can be applied to describe other Petri net dialects (e.g., time Petri nets and colored Petri nets). Figure 13-9 shows the concepts that need to be introduced into the ontology in order to support upgraded Petri nets. Most of these concepts are ontology properties: *attribute X* and *attribute Y* are graphical features of the *Place* class; *function*, *function firing level*, and *time function* are features of the *Transition* class; and *typeArc* is a feature of the *Arc* class. The extension of the ontology for upgraded Petri nets also requires a restriction on the *Arc* class: an arc can only connect a place and a transition.

Core ontology concepts	PLACE	TRANSITION	ARC
Upgraded Petri net extensions	<ul style="list-style-type: none"> ▪ <i>attribute X</i> – graphical feature ▪ <i>attribute Y</i> – graphical feature 	<ul style="list-style-type: none"> ▪ <i>function</i> – property ▪ <i>function firing level</i> – property ▪ <i>time function</i> – property 	<ul style="list-style-type: none"> ▪ <i>an arc can only connect a place and a transition</i> – restriction ▪ <i>typeArc</i> – property

Fig. 13-9. The relation between the core Petri net concepts and their extensions for upgraded Petri nets

In terms of the OUP, this extension means that we have a new <<ontology>> package, which contains all concepts and restrictions specific to upgraded Petri nets. Figure 13-10 shows how we have attached the *typeArc* property to the *Arc* class. In fact, the domain of the *typeArc* property is *Arc*, whereas the enumeration *ArcType* is the range of the *typeArc*. The enumeration *ArcType* consists of four individuals: “normal,” “inhibitor,” “reset,” and “read.”

Having introduced all concepts and restrictions for upgraded Petri nets in the OUP model, we can generate its OWL equivalent using an XSLT. Figure 13-11 contains an excerpt from the OWL ontology generated for the *Arc* class. On the left side (Fig. 13-11a), we show the definition of the *Arc* class for the core Petri net ontology. On the right side (Fig. 13-11b), we show an excerpt from the corresponding definition in the ontology for upgraded Petri nets. It should be noted that Fig. 13-11b depicts only how the *typeArc* property is added in the OWL ontology. First, we added the *typeArc* property as a definition of a new object property, which has the *Arc* class as its domain and the *ArcType* class as its range. *ArcType* is an enumeration that consists of the individuals we have already mentioned.

The *Arc* class has only a cardinality restriction on the *typeArc* property. Note that the *Arc* class for upgraded Petri nets contains all of the definition of the *Arc* class in the core Petri net ontology (i.e., the definition shown in Fig. 13-11a). Then, following the same principle, the XSLT converter produced the other parts of the OWL ontology for upgraded Petri nets.

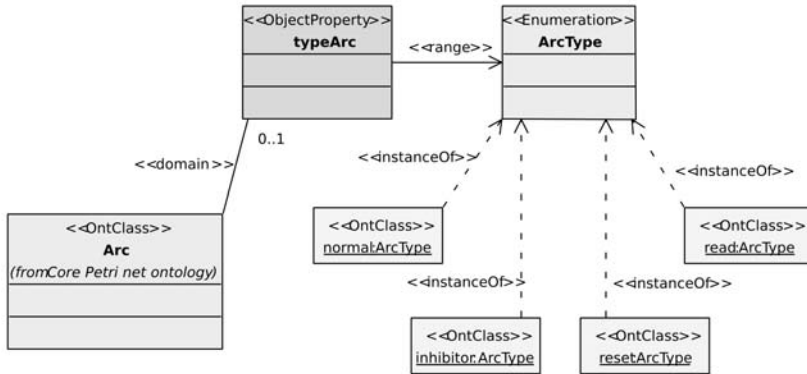


Fig. 13-10. An extension of the *Arc* class for upgraded Petri nets: the *typeArc* property with its range (the enumeration of the following values: “normal,” “inhibitor,” “reset,” and “read”)

<pre> <owl:Class rdf:ID="Arc"> <rdfs:subClassOf rdf:resource="#ModelElement"/> <rdfs:subClassOf> <owl:Restriction> <owl:onProperty rdf:resource="#multiplicity"/> <owl:maxCardinality rdf:datatype="#xsd:#nonNegativeInteger"> 1</owl:maxCardinality> </owl:Restriction> </rdfs:subClassOf> <rdfs:subClassOf> <owl:Restriction> <owl:onProperty rdf:resource="#fromNode"/> <owl:cardinality rdf:datatype="#xsd:#nonNegativeInteger"> 1</owl:cardinality> </owl:Restriction> </rdfs:subClassOf> <rdfs:subClassOf> <owl:Restriction> <owl:onProperty rdf:resource="#toNode"/> <owl:cardinality rdf:datatype= "#xsd:#nonNegativeInteger"> 1</owl:cardinality> </owl:Restriction> </rdfs:subClassOf> </owl:Class> </pre> <p style="text-align: center;">a)</p>	<pre> <owl:ObjectProperty rdf:ID="typeArc"> <rdfs:range rdf:resource="#ArcType"/> <rdfs:domain rdf:resource="#Arc"/> </owl:ObjectProperty> <owl:Class rdf:ID="ArcType"> <owl:oneOf rdf:parseType="Collection"> <ArcType rdf:about="#normal"/> <ArcType rdf:about="#inhibitor"/> <ArcType rdf:about="#reset"/> <ArcType rdf:about="#read"/> </owl:oneOf> </owl:Class> <ArcType rdf:ID="normal"/> <ArcType rdf:ID="inhibitor"/> <ArcType rdf:ID="reset"/> <ArcType rdf:ID="read"/> <owl:Class rdf:ID="Arc"> <!-- This extended Arc class has the same content as the Arc class in the Core Petri net ontology plus the following content: --> <rdfs:subClassOf> <owl:Restriction> <owl:onProperty rdf:resource="#typeArc"/> <owl:maxCardinality rdf:datatype="#xsd:#nonNegativeInteger"> 1</owl:maxCardinality> </owl:Restriction> </rdfs:subClassOf> <!--Restriction that an arc can only connect a place and transition--> </owl:Class> </pre> <p style="text-align: center;">b)</p>
--	--

Fig. 13-11. OWL definition of the *Arc* class: (a) for the core Petri net ontology; (b) for the ontology of upgraded Petri nets

Figure 13-12 shows how we have extended the *Place* class. We have added the *attributeX* and *attributeY* properties, so that each instance of the *Place* class may have instances of those properties. The ranges of *attributeX* and *attributeY* are *AttributeXDescriptor* and *AttributeYDescriptor* classes, respectively. Using the same principle, we have introduced two other datatype properties, namely *firingLevel* (Integer) and *functionName* (String).



Fig. 13-12. Attaching the *attributeX* and *attributeY* properties to the *Place* class

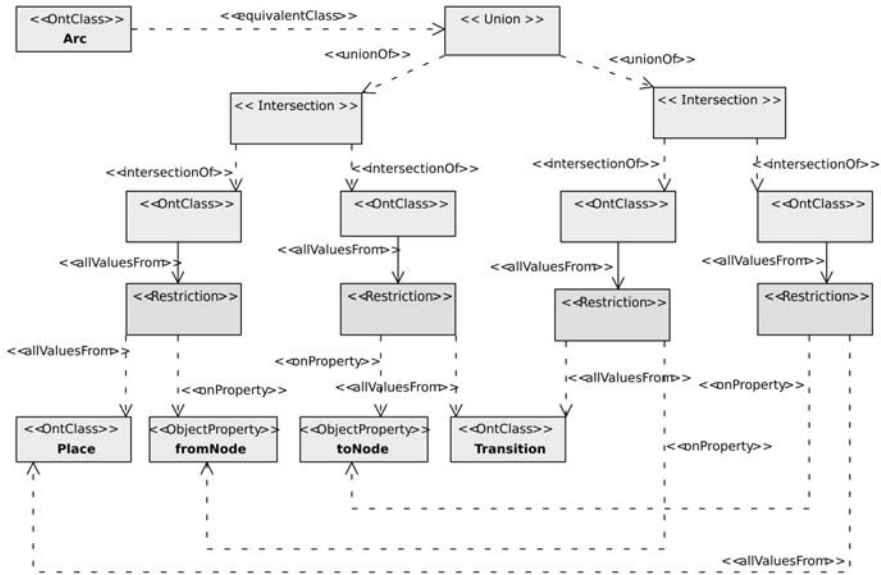


Fig. 13-13. An extension of the ontology regarding the *Arc* class that additionally restricts this class: an arc only may connect a transition and a place. This restriction is represented in the OUP that can be transformed to OWL using an XSLT

Figure 13-13 shows how we have put a restriction on a Petri net arc using the Ontology UML Profile. Note that this restriction is not part of the core Petri net ontology, since it is not a generally applicable rule for all Petri net dialects. However, most Petri net dialects have this restriction, and hence we have applied it here.

This restriction means that a Petri net arc (`<<OntClass>> Arc`) may connect only a *Place* and a *Transition*. This statement is expressed as a union (`<<Union>>`) of two intersections (`<<Intersection>>`). Our `<<OntClass>> Arc` is an equivalent class (`<<equivalentClass>>`) of this union. Since these two intersections are defined in a symmetric way, we shall describe only the left-hand one in Fig. 13-13. This intersection says that an *Arc* takes all its values from (`<<allValuesFrom>>` association) `<<OntClass>> Place` for the *fromNode* property and from `<<OntClass>> Transition` for the *toNode* property. The second (right) intersection specifies the opposite statement: *Arc*'s *toNode* property takes all its values from *Place*, and *Arc*'s *fromNode* property takes all its values from *Transition*.

13.2 Educational Ontologies¹

Specifying reusable chunks of learning content and defining an abstract way of describing designs for various units of learning (e.g., courses and lessons) are two of the most current research issues in the e-learning community. One field of research in this area is that of learning objects. Among the many important definitions of learning objects such as [Barrit et al., 1999; Richards, 2002; Wiley, 2002], we refer to a very broad definition [Duval, 2002]: A learning object is any entity, digital or nondigital, that can be used, reused, or referenced during technology-supported learning. As a result of research in that field we have the IEEE Learning Object Metadata (LOM) standard, which is a standardized set of metadata fields (e.g., title, description and category) that describe learning objects. However, in addition to this vague definition, learning objects suffer from a lack of ability to semantically express relations among different types of objects in the context of their use in an educational setting [Koper, 2001]. Accordingly, there is a second field of research referred to as learning design, which can be defined as the application of a pedagogical model to a specific learning objective, a specific target group, and a specific context

¹ Portions reprinted with minor changes, with permission, from [Knight et al., 2006].

or knowledge domain [Koper & Olivier, 2004]. As a response to these activities, there has been an initiative by IMS to define learning design-related recommendations [IMS LD IM, 2003].

Although the two aforementioned initiatives are interrelated, some questions still have to be answered, such as the following: How can we employ just some specific parts of a learning object, rather than the learning object as a whole in a specific learning design? How can we reuse the same learning design in different contexts with different learning objects? How can we personalize the content of the same learning object according to learners' models in the same learning design? How can we develop more extensive search and ranking services for learning objects and learning designs?

To address these problems, we have developed a set of ontologies to link learning designs and learning content.² The use cases that have driven our efforts involve enabling teachers to reuse learning designs not through complete automation of the process, but through better search services.

13.2.1 Conceptual Solution

We start from the premise that learning design offers tremendous potential for content repurposing. Starting with some educational content in the form of learning objects (including images, text, and animations) and some Web-based learning support services (chat, messaging, and multiple-choice tests), learning designs can choreograph the order in which the content will be presented, how it will be integrated into learning support services, how it will be sequenced, and how it will be assigned to learners in a lesson. Conceptually, this can be pictured as pulling learning objects from a repository and using the learning designs to integrate the learning objects into activities that involve the learners. In fact, one can regard a learning design as a kind of workflow chain definition specifying what activities learners will be involved in and what learning objects will be used in those activities. The IMS-LD specification [IMS LD IM, 2003] provides a capability to reference external learning objects through URI property elements and keep a clear separation between the learning design and the content being referenced.

When learning objects are incorporated into a learning design, there may be many possible learning objects to choose from. A course author

² We are very grateful to our colleagues Colin Knight and Griff Richards at Simon Fraser University, Canada, Jelena Jovanović at the University of Belgrade, Serbia and Montenegro, and Katrien Verbert at the Katholieke Universiteit Leuven, Belgium, for helping us write this section on educational ontologies.

will be able to automatically search through learning object repositories for suitable content. Ideally, the learning objects will contain metadata that will help the course author to identify the most suitable content for a specific purpose. However, this assumes that the learning object will have a single instructional context for which it can be useful. From the standpoint of learning object reuse, it would be advantageous for a learning object to have many different uses, so that expensive multimedia content elements could be reused in as many different learning objects as possible. For example, a learning object that contains pictures of the Acropolis could be used for both a grade 10 art course and a university-level history course.

The optimal way to facilitate the integration of learning objects into a learning design without compromising reusability is to treat the contexts for learning objects (learning object contexts, or LOCs) as distinct entities from the learning objects themselves, as shown in Fig. 13-14. The learning objects exist independently of any presupposed instructional context, meaning that they can be used in any situation in which a course author finds them useful. Within an extensive domain of many different instructional contexts, many different LOCs can be created and associated with learning objects in a many-to-many relationship. If a course author decides that a particular learning object is useful in a grade 7 biology course, a new context object is created associating that learning object with that specific context.

If we annotate the learning object with context information such as the prerequisites and competencies applicable to the learning object in a grade 7 biology course, we establish an implied ownership relation. In this case, the learning object can be owned by learning designs that target seventh-grade biology or an equivalent course. If we instead choose to include the information in the learning design, the learning design will be tied to a particular context, which reduces its reusability. Looking again at Fig. 13-14, we see the domain of instructional contexts. This shaded background represents all of the possible ways a given learning design could be used in practice. The learning objects remain outside this domain, so that they can be used by other learning designs in other contexts. In fact, a new LOC is created by associating any such learning object with a specific context.

An LOC would contain data that was specific to a single learning object in a particular instructional context. Learning objectives, competencies, and the evaluation would be stored in this object as opposed to the learning object, so that the learning object could be associated with multiple LOCs and various learning objectives, competencies, and evaluations. The LOC could also contain context-specific information about the subject domain ontology, since the specification of subject domain annotations will be dependent on the context.

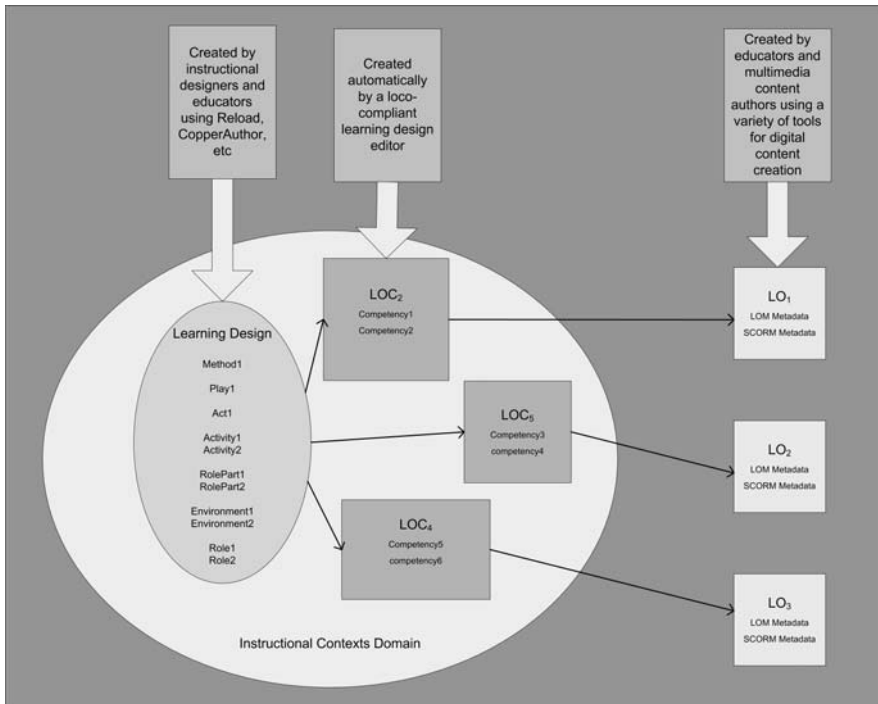


Fig. 13-14. Learning object contexts: a conceptual model

13.2.2 Mapping the Conceptual Model to Ontologies

In our efforts to provide an explicit specification (i.e., ontology) of the conceptual model depicted in Fig. 13-14 we have identified a need for the following three ontologies: (a) an ontology of learning object content, (b) an ontology of learning design, and (c) an ontology connecting those two ontologies. In the remainder of this section, we describe each of these ontologies in detail.

Learning Object Content Ontology

Having looked at several content models (e.g., Learnativity, the SCORM content aggregation model, CISCO RLO/RIO, and NETg), we decided to use the Abstract Learning Object Content Model (ALOCoM), the result of a recent EU ProLearn NoE project [Verbert et al., 2004], as a basis for an ontology that describes learning object content. The ALOCoM was designed to generalize all of the other content models mentioned, to provide

an ontology-based platform for integrating various content models, and to enable (semi-)automatic reuse of components of learning objects by explicitly defining their structure [Sheth et al., 2005].

On top of that model, we built an ontology called the ALOCoM ontology [Jovanović et al., 2005b]. This ontology is divided into two parts [Jovanović et al., 2005a]:

- the *ALoCoM Content Structure (CS)* ontology, which enables a formal representation of learning objects decomposed into components;
- the *ALoCoM Content Type (CT)* ontology, which defines the educational role of learning objects and their components.

In Fig. 13-15, we show the relation and organization of the ontologies representing the ALOCoM.

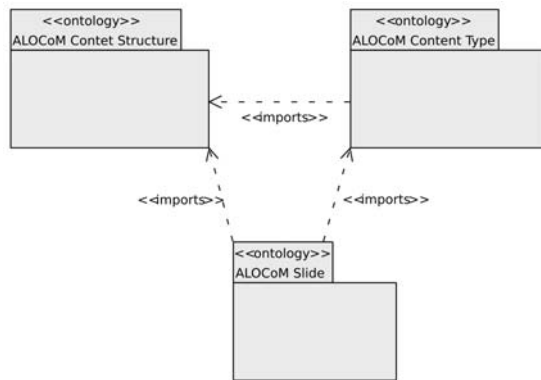


Fig. 13-15. Organization of the ALOCoM ontologies: the ALOCoM Content Structure ontology, the ALOCoM Content Type ontology, and an extension of these two ontologies (ALoCoM Slide)

The organization of the ALOCoM ontology was inspired by the IBM Darwin Information Typing Architecture (DITA), which contains a core part with general-purpose concepts (e.g., text, paragraph, list, ordered list, list item, and table) that can be extended depending on the specific needs of the author or user. Therefore, we organized the ALOCoM ontology as an extensible infrastructure consisting of a core part (containing the ALOCoM CS and ALOCoM CT packages, i.e., ontologies) with concepts common to all types of learning objects and an unlimited number of extensions, each extension supporting one specific type of learning object (e.g., the ALOCoM Slide ontology, which defines concepts specific to slide presentations).

domain. Accordingly, many of the DITA building blocks, such as *section*, *paragraph*, and *list*, are included in the core ontology as either direct or indirect subclasses of the *ContentObject* class. We did not include those DITA elements that are related to presentation, such as the *searchtitle* element, which is used when transforming a DITA element to XHTML to create a title element at the top of the resulting HTML file.

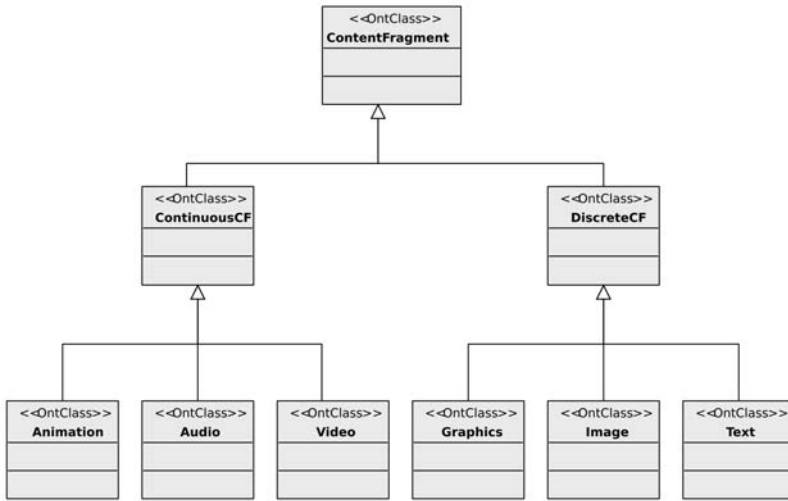


Fig. 13-17. The hierarchy of the *ContentFragment* class

The ALOCoM CT ontology is also rooted in the ALOCoM and has CF, CO, and LO as basic, abstract content types. However, these concepts are now regarded from perspective of educational/instructional roles they might have. Therefore, concepts such as *Definition*, *Example*, *Exercise*, and *Reference* have been introduced as subclasses of the *ContentObject* class (see Fig. 13-18), whereas concepts such as *Tutorial*, *Lesson*, and *Test* are some of the subclasses of the *LearningObject* class. The creation of this ontology was mostly inspired by a thorough examination of existing learning object content models [Verbert & Duval, 2004] by closely related work presented in [Ullrich, 2005]. The concepts defined in this ontology are used as values for the Learning Resource Type element of the IEEE LOM standard.

In Fig. 13-19a we show how we have extended the ALOCoM core concepts to support slide presentations, and Fig. 13-19b illustrates how we have extended the *hasPart* property in the *SlidePresentation* class, so that it may only take values that are instances of the Slide class.

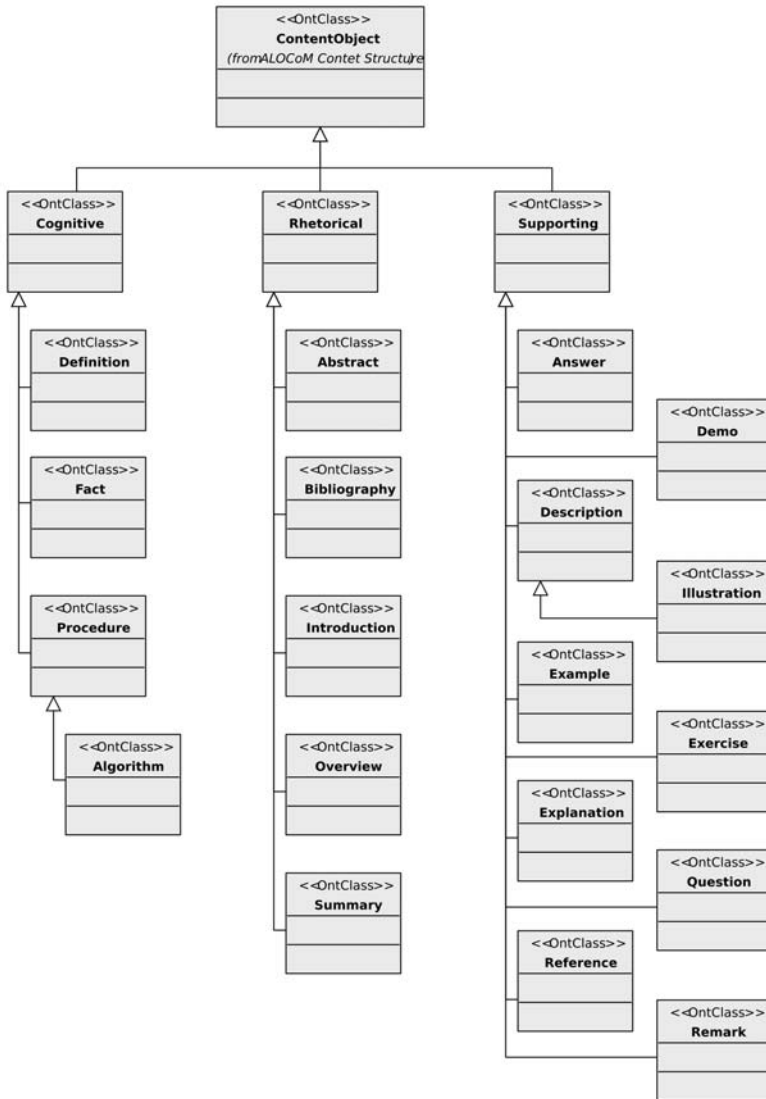


Fig. 13-18. Part of the ALOCoM Content Type ontology: the *ContentObject* class and its subclasses

LOCO – an Ontology Compatible with IMS-LD

The IMS-LD Information Model and its XML binding are a specification for learning design [IMS LD IM, 2003]. As many of the tools and editors for learning design will be developed around this specification, it is impor-

tant to maintain compatibility. We have used the IMS-LD Information Model as a blueprint for the creation of an IMS-LD-based ontology named the Learning Object Context Ontology (LOCO). To create the LOCO, we needed to make some changes to the IMS-LD Information Model [IMS LD IM, 2003] in order to conform to established good-practice recommendations for ontology design [Noy et al., 2001], and to resolve some ambiguities and inconsistencies in the information model. We have reported these inconsistencies in detail in [Knight et al., 2006].

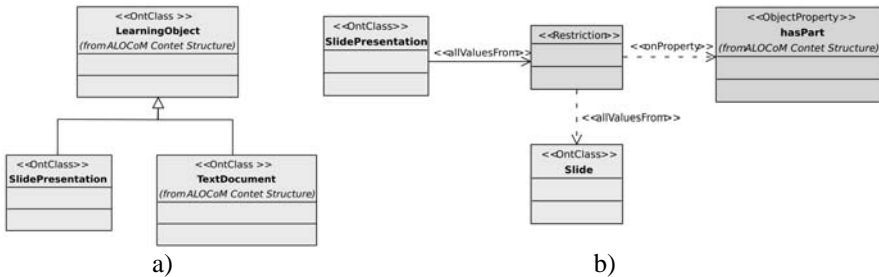


Fig. 13-19. Extension of the ALOCoM to support slide presentations (a), and the additional restriction on the *hasPart* property (b)

Figure 13-20 shows the LOCO's classes and their inheritance relationships, expressed in the GOOD OLD AI OUP. The main emphasis is on the *Learning_object* class, since our goal is to make a connection between learning content (e.g., represented in the ALOCoM ontology by the *LearningObject* class) and learning design (i.e., the LOCO). In the LOCO, the *Learning_object* class is a subclass of the *ResourceDescription* class. Accordingly, the *Learning_object* class inherits the following properties from the *ResourceDescription* class (see Fig. 13-21): *item*, *metadata*, *title*, and *hasResource*.

Let us describe the *hasResource* property in order to illustrate one of the class properties of the LOCO (see Fig. 13-22). Initially, the range of the *hasResource* property is the *Resource* class. However, according to the IMS-LD specification, we have to restrict this range, so that the range is a union of the *web_content* and *Imsl_d_content* classes (i.e., *hasResource* in the class *Learning_object* can take values that are instances of the *web_content* and *Imsl_d_content* classes). Additionally, the *Learning_object* class has the *hasLearning_object* property, which allows learning objects to aggregate other learning objects.

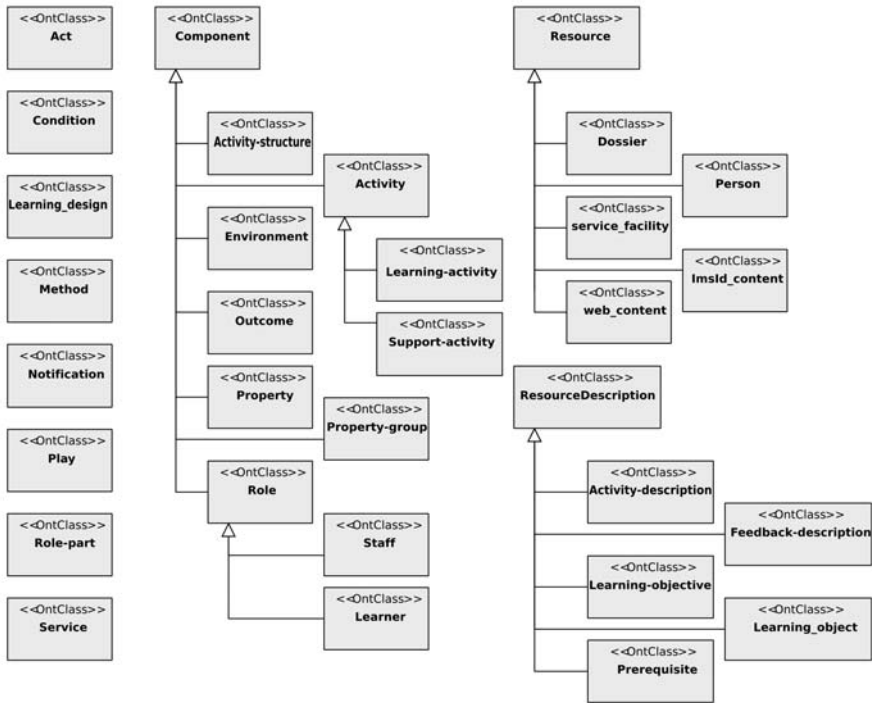


Fig. 13-20. Class organization of the LOCO

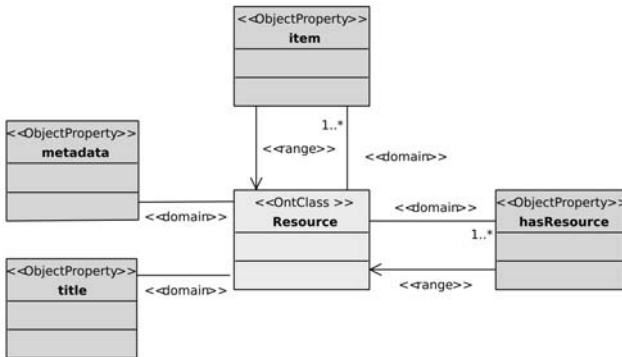


Fig. 13-21. The properties of the *Resource* class in the LOCO

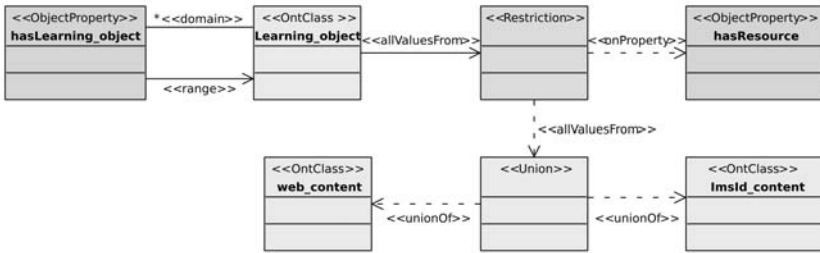


Fig. 13-22. The properties of and restriction on the *Learning_object* class in the LOCO

LOCO-Cite – an Ontology for Bridging the Learning Object Content and Learning Design Ontologies

The final step is to create an ontology that serves as a bridge linking the LOCO and the ALOCoM ontology in accordance with the conceptual model of learning object contexts shown in Fig. 13-14. Because this makes an explicit reference to a specific learning object, we have named this ontology LOCO-Cite. The LOCO and the ALOCoM ontology must be related to each other through LOCO-Cite, which links properties and classes across the boundaries of the individual ontologies to create a larger, unified ontology. Considering the constraints of the current versions of Protégé, which are not designed to work with multiple ontologies in the same view, we found the use of the OUP to be a suitable solution to this problem. Figure 13-23 shows how we organized the ontologies in order to connect the ALOCoM ontology and the LOCO, and thus the representation of learning objects and learning designs.

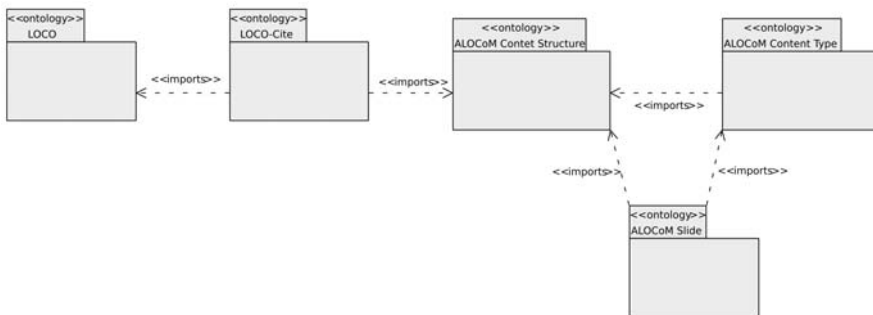


Fig. 13-23. Connecting the ALOCoM ontology and the LOCO through the LOCO-Cite ontology

Figure 13-24 indicates how the *LearningObjectContext* class in the LOCO-Cite ontology is linked to related concepts from both the LOCO (the *Learning_object* class in Fig. 13-20 and Fig. 13-22) and the ALOCoM Content Structure ontology (the *LearningObject* class in Fig. 13-16). First, we have defined a relation between the LOCO-Cite ontology and the ALOCoM ontology by saying that the *LearningObjectContext* class in LOCO-Cite is an *equivalentClass* to the *LearningObject* class in the ALOCoM ontology. Then, we have created a relation between the LOCO-Cite ontology and the LOCO through the *hasLearningObject* property of LOCO-Cite's *Learning_object* class, whose range is the *LearningObject* class in the ALOCoM Content Structure ontology.

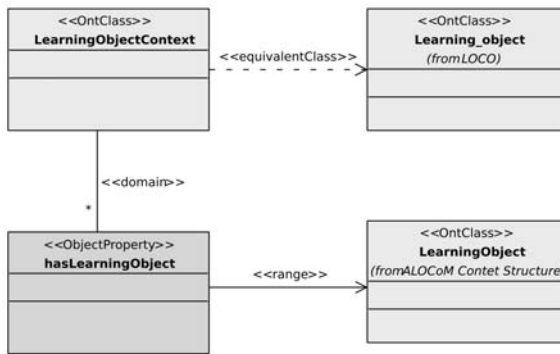


Fig. 13-24. Linking the LOCO and the ALOCoM ontology by introducing the *LearningObjectContext* class and the *hasLearningObject* into the LOCO-Cite ontology

All ontologies defined by the OUP (ALOCOM, LOCO, and LOCO-Cite in the present case) can be transformed into their OWL equivalents using the XSLT that we developed, as we have already illustrated for the Petri net ontology.

Having defined the above ontology-based framework for bridging learning objects and learning designs, we have provided an infrastructure for potential semantic services such as:

- employing descriptions of learning designs and learning objects by ontologies to search and reuse them, either as a whole or as disaggregated components;
- finding the most suitable teaching method stored in learning design repositories on the basis of specific competencies;

- personalizing learning objects according to the learners' profiles within a specific learning design by employing an ontology-based description of learning object content; and
- ranking learning designs returned by searches using various weight factors for the relationships defined in the proposed ontologies, users' reviews of both learning designs and learning objects, and ontology-defined competencies.

References

- Arnold, W.R. & Bowie, J.S. (1985), *Artificial Intelligence: A Personal Common-sense Journey*, Prentice Hall, Englewood Cliffs, NJ.
- Atkinson, C. & Kühne, T. (2002), "Profiles in a strict meta-modeling framework," *Science of Computer Programming*, vol. 44, no. 1, pp. 5–22.
- Atkinson, C. & Kühne, T. (2003), "Model-driven development: A metamodeling foundation," *IEEE Software*, vol. 20, no. 5, pp. 36–41.
- Baader, F., Calvanese, D., McGuinness, D., Nardi, D., & Patel-Schneider, P., eds. (2003), *The Description Logic Handbook*, Cambridge University Press, Cambridge, UK.
- Baclawski, K., Kokar, M., Kogut, P., Hart, L., Smith, J.E., Letkowski, J., & Emery, P. (2002a), "Extending the Unified Modeling Language for ontology development," *Software and Systems Modeling*, vol. 1, no. 2, pp. 142–156.
- Baclawski, K., Kokar, M., Smith, J.E., Wallace, E., Letkowski, J., Koethe, M.R., & Kogut, P. (2002b), *UOL: Unified Ontology Language: Assorted Papers Discussed at the DC Ontology SIG Meeting* [Online]. Available: <http://www.omg.org/cgi-bin/doc?ontology/2002-11-02> [Accessed: 2005, November 23].
- Bandwidth Market (2005), *Bandwidth Market, Glossary* [Online]. Available: <http://www.bandwidthmarket.com/resources/glossary/A7.html> [Accessed: 2005, September 9].
- Barr, A. & Feigenbaum, E.A., eds. (1981), *The Handbook of Artificial Intelligence* (3 volumes), William Kaufmann, Los Altos, CA.
- Barrit, C., Lewis, D., & Wieseler, W. (1999), *CISCO Systems Reusable Information Object Strategy Version 3.0* [Online]. Available: http://www.cisco.com/warp/public/779/ibs/solutions/learning/whitepapers/el_cisco_rio.pdf [Accessed: 2005, May 06].
- Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., & Stein, L.A. (2004), *OWL Web Ontology Language Reference* [Online]. Available: <http://www.w3.org/TR/owl-ref/> [Accessed: 2005, November 27].
- Berners-Lee, T. (1997–2004), *Design Issues: Architectural and Philosophical Points, W3C* [Online]. Available: <http://www.w3.org/DesignIssues/> [Accessed: 2005, October 11].
- Berners-Lee, T., Fischetti, M., & Dertouzos, T.M. (1999), *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*, Harper, San Francisco.

- Berners-Lee, T., Hendler, J., & Lassila, O. (2001), "The Semantic Web," *Scientific American*, vol. 284, no. 5, pp. 34–43.
- Bézivin, J. (2001), "From object composition to model transformation with the MDA," *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems*, Santa Barbara, CA, pp. 350–355.
- Bézivin, J. (2004), "In search of a basic principle for model driven engineering," *Upgrade*, vol. 5, no. 2, pp. 21–24.
- Bézivin, J., Dupé, G., Jouault, F., Pitette, G., & Rougui, J.E. (2003), "First experiments with the ATL model transformation language: Transforming XSLT into XQuery," *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture*, Anaheim, CA.
- Bézivin, J., Djurić, D., Devedžić, V., Favreau, J., Gašević, D., & Jouault, F. (2005), "An M3-neutral infrastructure for bridging model engineering and ontology engineering," *Proceedings of the 1st International Conference on Interoperability of Enterprise Software and Applications*, Geneva.
- Billington, J., Christensen, S., van Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., & Weber, M. (2003), "The Petri net markup language: Concepts, technology, and tools," *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*, Eindhoven, pp. 483–505.
- Bock, C. (2003), "UML without pictures," *IEEE Software*, vol. 20, no. 5, pp. 33–35.
- Boley, H., Tabet, S., & Wagner, G. (2001), "Design rationale of RuleML: A markup language for Semantic Web rules," *Proceedings of the 1st Semantic Web Working Symposium*, Stanford University, CA, pp. 381–401.
- Booch, G., Rumbaugh, J., & Jacobson, I. (1998), *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA.
- Brachman, R.J. (1979), "On the epistemological status of semantic networks," in *Associative Networks: Representations and Use of Knowledge by Computers*, ed. N.V. Findler, Academic Press, Burlington, MA, pp. 3–50.
- Brachman, R.J. & Schmoltze, J.G. (1985), "An overview of the KL-ONE knowledge representation system," *Cognitive Science*, vol. 9, no. 2, pp. 171–216.
- Brachman, R.J., McGuinness, D.L., Patel-Schneider, P.F., Resnick, L.A., & Borgida, A. (1991), "Living with CLASSIC: When and how to use a KL-ONE like language," in *Principles of Semantic Networks*, ed. J. Sowa, Morgan Kaufmann, San Francisco, CA, pp. 401–456.
- Brickley, D. & Guha, R.V. (2004), *RDF Vocabulary Description Language 1.0: RDF Schema, W3C Working Draft* [Online]. Available: <http://www.w3.org/TR/PR-rdf-schema> [Accessed: 2005, October 5].
- Brockmans, S., Volz, R., Eberhart, A., & Löffler, P. (2004), "Visual modeling of OWL DL ontologies using UML," *Proceedings of the 3rd International Semantic Web Conference*, Hiroshima, pp. 108–213.
- Carlson, D. (2001), *Modeling XML Applications with UML: Practical e-Business Applications*, Addison-Wesley, Reading, MA.

- Ceccaroni, L. & Kendall, E. (2003), "A graphical environment for ontology development," *Proceedings of the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems*, Melbourne, pp. 958–959.
- Chandrasekaran, B., Josephson, J.R., & Benjamins, V.R. (1999), "What are ontologies, and why do we need them?" *IEEE Intelligent Systems*, vol. 14, no. 1, pp. 20–26.
- Chaudhri, V.K., Farquhar, A., Fikes, R., Karp, P.D., & Rice, J.P. (1998), *Open Knowledge Base Connectivity 2.0.3* [Online]. Available: <http://www.ai.sri.com/~okbc/spec/okbc2/okbc2.html> [Accessed: 2005, September 23].
- Clark, A. (2001), *Mindware: An Introduction to the Philosophy of Cognitive Science*, Oxford University Press, New York.
- Clark, P. & Porter, B. (2004), *KM – the Knowledge Machine 2.0: Users Manual* [Online]. Available: <http://www.cs.utexas.edu.proxy.lib.sfu.ca/users/mfkb/km/userman.pdf> [Accessed: 2005, September 17].
- Conallen, J. (2002), *Building Web applications with UML*, 2nd Edition, Addison-Wesley, Reading, MA.
- Corcho, O., Fernández-López, M., & Gómez-Pérez, A. (2002), "Methodologies, tools and languages for building ontologies. Where is their meeting point?" *Data and Knowledge Engineering*, vol. 46, no. 1, pp. 41–64.
- CraneField, S. (2001a), "Networked knowledge representation and exchange using UML and RDF," *Journal of Digital Information*, vol. 1, no. 8, article no. 44, 2001-02-15.
- CraneField, S. (2001b), "UML and the Semantic Web," *Proceedings of the Semantic Web Working Symposium*, Stanford University, CA, pp. 113–130.
- DAML Ontology Library (2005), *DAML Ontology Library, DARPA* [Online]. Available: <http://www.daml.org/ontologies/> [Accessed: 2005, October 03].
- Davis, R., Shrobe, H., & Szolovits, P. (1993), "What is a knowledge representation?" *AI Magazine*, vol. 14, no. 1, pp. 17–33.
- DC Metadata Schema (2005), *Dublin Core Metadata Schema* [Online]. Available: <http://dublincore.org/documents/dcmi-terms/> [Accessed: 2005, July 11].
- Decker, S., Melnik, S., van Harmelen, F., Fensel, D., Klein, M., Broekstra, J., Erdmann, M., & Horrocks, I. (2000), "The Semantic Web: The roles of XML and RDF," *IEEE Internet Computing*, vol. 4, no. 5, pp. 63–74.
- Denny, M. (2002), *Ontology Building: A Survey of Editing Tools* [Online]. Available: <http://www.xml.com/pub/a/2002/11/06/ontologies.html> [Accessed: 2005, October 05].
- Denny, M. (2004), *Ontology Tools Survey, Revisited* [Online]. Available: <http://www.xml.com/pub/a/2004/07/14/onto.html> [Accessed: 2005, October 05].
- Devedžić, V. (1999), "Ontologies: Borrowing from software patterns," *ACM Intelligence*, vol. 10, no. 3, pp. 14–24.
- Devedžić, V. (2002), "Understanding ontological engineering," *Communications of the ACM*, vol. 45, no. 4, pp. 136–144.
- Devedžić, V. (2004a), "Education and the Semantic Web," *International Journal of Artificial Intelligence in Education*, vol. 14, pp. 39–65.

- Devedžić, V. (2004b), "Web intelligence and artificial intelligence in education," *Educational Technology & Society*, vol. 7, no. 4, pp. 29–39.
- deVos, A. & Rowbotham, C.T. (2001), "Knowledge representation for power system modelling," *Proceedings of the 22nd International Conference on Power Industry Computer Applications*, Sydney, pp. 50–56.
- deVos, A., Widergren, S.E., & Zhu, J. (2001), "XML for CIM Model Exchange," *Proceedings of the 22nd International Conference on Power Industry Computer Applications*, Sydney, pp. 31–37.
- Dirckze, R. (2002), *Java Metadata Interface (JMI) Specification Version 1.0* [Online]. Available: <http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html> [Accessed: 2005, November 27].
- Djurić, D. (2004), "MDA-based ontology infrastructure," *Computer Science and Information Systems*, vol. 1, no. 1, pp. 91–116.
- Djurić, D., Gašević, D., & Devedžić, V. (2005a), "Ontology modeling and MDA," *Journal of Object Technology*, vol. 4, no. 1, pp. 109–128.
- Djurić, D., Gašević, D., Devedžić, V., & Damjanović, V. (2005b), "A UML profile for OWL ontologies," in *Model Driven Architecture (European MDA Workshops: Foundations and Applications, MDFAFA 2003 and MDFAFA 2004, Twente, The Netherlands, June 26–27, 2003 and Linköping, Sweden, June 10–11, 2004. Revised Selected Papers)*, eds. U. Almann, M. Aksit, & A. Rensink, Lecture Notes in Computer Science, No. 3599, Springer, Berlin, Heidelberg, pp. 204–219.
- Djurić, D., Gašević, D., & Devedžić, V. (2005c), "MDA standards for ontology development" in *Ontologies in the Context of Information Systems*, eds. R. Sharman, R. Ramesh, & R. Kishore, Springer, Heidelberg, in press.
- Djurić, D., Devedžić, V., & Gašević, D. (2006a), "Adopting software engineering trends in AI," *IEEE Intelligent Systems*, in press.
- Djurić, D., Gašević, D., & Devedžić, V. (2006b), "The Tao of modeling spaces," *Journal of Object Technology*, in press.
- DOM (2005), *Document Object Model (DOM), W3C* [Online]. Available: <http://www.w3.org/DOM/> [Accessed: 2005, October 12].
- Domingue, J., Dzbór, M., & Motta, E. (2004), "Semantic layering with Magpie," in *Handbook on Ontologies*, eds. S. Staab & R. Studer, Springer, Berlin, Heidelberg, pp. 533–553.
- Duddy, K. (2002), "UML2 must enable a family of languages," *Communications of the ACM*, vol. 45, no. 11, pp. 73–75.
- DUET (2005), *DAML-UML Enhanced Tool (DUET) Documents* [Online]. Available: http://codip.grci.com/wwwlibrary/wwwlibrary/wwwlibrary/DUET_Docs/ [Accessed: 2005, November 23].
- Durkin, J. (1994), *Expert Systems: Design and Development*, Macmillan, New York.
- Duval, E. (2002), *IEEE Standard for Learning Object Metadata, IEEE Learning Technology Standardization Committee* [Online]. Available: http://ltsc.ieee.org/wg12/files/LOM_1484_12_1_v1_Final_Draft.pdf [Accessed: 2006, March 06].

- e-Learning Guru (2005), *e-Learning Guru, Glossary* [Online]. Available: <http://e-learningguru.com/gloss.htm> [Accessed: 2005, September 09].
- Evans, R. & Gazdar, G. (1996), "DATR: A language for lexical knowledge representation," *Computational Linguistics*, vol. 22, no. 2, pp. 167–216.
- Falkovych, K., Sabou, M., & Stuckenschmidt, H. (2003), "UML for the Semantic Web: Transformation-based approaches," in *Knowledge Transformation for the Semantic Web*, eds. B. Omelayenko & M. Klein, *Frontiers in Artificial Intelligence and Applications*, vol. 95, IOS, Amsterdam, pp. 92–106.
- Fensel, D. (2004), *Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce*, 2nd edition, Springer, Berlin, Heidelberg.
- Fensel, D. & Musen, M.A. (2001), "The Semantic Web: A brain for humankind," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 24–25.
- Fensel, D., van Harmelen, F., Horrocks, I., McGuinness, D.L., & Patel-Schneider, P.F. (2001), "OIL: An ontology infrastructure for the Semantic Web," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 38–45.
- Fernández-López, M., Gómez-Pérez, A., Sierra, J.P., & Sierra, A.P. (1999), "Building a chemical ontology using Methontology and the Ontology Design Environment," *IEEE Intelligent Systems*, vol. 14, no. 1, pp. 37–46.
- Fikes, R. (1998), *Multi-use Ontologies*, *Stanford University* [Online]. Available: <http://www.ksl.stanford.edu/people/fikes/cs222/1998/Ontologies/tsld001.htm> [Accessed: 2005, October 03].
- Fowler, M. (2003), *UML Aistilled, A Brief Guide to the Standard Object Modeling Language*, 3rd Edition, Addison-Wesley, Reading, MA.
- Frankel, S.D. (2003), *Model Driven Architecture: Applying MDA to Enterprise Computing*, Wiley, New York.
- Friedman-Hill, E. (2003), *Jess in Action*, Manning, Greenwich, UK.
- Gaines, B.R. (1991), "An interactive visual language for term subsumption languages," *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, Sydney, pp. 817–823.
- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.
- Gašević, D. & Devedžić, V. (2004), "Reusing Petri nets through the Semantic Web," *Proceedings of the 1st European Semantic Web Symposium*, Heraklion, Greece, pp. 284–298.
- Gašević, D. & Devedžić, V. (2005), "Interoperable Petri net models via ontology," *Journal of Web Engineering and Technology*, in press.
- Gašević, D., Djurić, D., Devedžić, V., & Damjanović, V. (2004a), "Converting UML to OWL ontologies," *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers & Posters*, NY, pp. 488–489.
- Gašević, D., Djurić, D., Devedžić, V., & Damjanović, V. (2004b), "UML for read-to-use OWL ontologies," *Proceedings of the 2nd IEEE International Conference on Intelligent Systems*, Vrana, Bulgaria, pp. 485–490.
- Gašević, D., Djurić, D., & Devedžić, V. (2005), "Bridging MDA and OWL ontologies," *Journal of Web Engineering*, vol. 4, no. 2, pp. 119–134.

- Genesereth, M.R. & Fikes, R.E. (1992), *Knowledge Interchange Format, Version 3.0, Reference Manual* [Online]. Available: <http://www-ksl.stanford.edu/knowledge-sharing/papers/kif.ps> [Accessed: 2005, October 16].
- Gómez-Pérez, A. & Corcho, O. (2002), "Ontology languages for the Semantic Web," *IEEE Intelligent Systems*, vol. 17, no. 1, pp. 54–60.
- Große, T., Doney, G., & Brodsky, S. (2002), *Mastering XMI: Java Programming with XMI, XML, and UML*, Wiley, New York.
- Gruber, T.R. (1992), *Ontolingua: A Mechanism to Support Portable Ontologies*, Knowledge Systems Laboratory, Stanford University, CA.
- Gruber, T.R. (1993), "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, no. 2, pp. 199–220.
- Guarino, N. (1995), "Formal ontology, conceptual analysis and knowledge representation," *International Journal of Human-Computer Studies*, vol. 43, no. 5/6, pp. 625–640.
- Haase, P., Broekstra, J., Eberhart, A., & Volz, R. (2004), "A comparison of RDF query languages," *Proceedings of the 3rd International Semantic Web Conference*, Hiroshima, pp. 502–517.
- Hagget, P. & Chorley, R.J. (1967), "Models, paradigms and new geography," in *Models in Geography*, Methuen, London.
- Hands Schuh, S. & Staab, S. (2002), "Authoring and annotation of Web pages in CREAM," *Proceedings of the 11th International World Wide Web Conference*, Honolulu, pp. 462–473.
- Hands Schuh, S. & Staab, S., eds. (2003a), *Annotation for the Semantic Web*, IOS, Amsterdam.
- Hands Schuh, S. & Staab, S. (2003b), "CREAM – creating metadata for the Semantic Web," *Computer Networks*, vol. 42, no. 5, pp. 579–598.
- Hands Schuh, S., Staab, S., & Volz, R. (2003a), "On deep annotation," *Proceedings of the 12th International World Wide Web Conference*, Budapest, pp. 431–438.
- Hands Schuh, S., Volz, R., & Staab, S. (2003b), "Annotation for the deep Web," *IEEE Intelligent Systems*, vol. 18, no. 5, pp. 42–48.
- Heflin, J. & Hendler, J. (2001), "A portrait of the Semantic Web in action," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 54–59.
- Hendler, J. (2001), "Agents and the Semantic Web," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 30–37.
- Hendler, J. & McGuinness, D. (2000), "The DARPA agent markup language," *IEEE Intelligent Systems*, vol. 15, no. 6, pp. 72–73.
- Hofstadter, D. (1994), *Fluid Concepts and Creative Analysis - Computer Models of the Fundamental Mechanisms of Thought*, Basic Books/Harper Collins, New York.
- Horrocks, I. & van Harmelen, F. (2002), *Reference Description of the DAML+OIL Ontology Markup Language* [Online]. Available: <http://www.daml.org/2001/03/reference> [Accessed: 2005, September 19].
- IMS LD IM (2003), *IMS Learning Design Information Model, Version 1.0 Final Specification, Revision 20* [Online]. Available: http://www.imsglobal.org/learningdesign/ldv1p0/imsld_infov1p0.html [Accessed: 2005, November 23].

- Jovanović, J., Gašević, D., & Devedžić, V. (2004), "A GUI for Jess," *Expert Systems with Applications*, vol. 26, no. 4, pp. 625–637.
- Jovanović, J., Gašević, D., & Devedžić, V. (2005a), "TANGRAM: An ontology-based learning environment for intelligent information systems," *Proceedings of the 10th World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education (E-Learn2005)*, Vancouver, pp. 2966–2971.
- Jovanović, J., Gašević, D., Verbert, K., & Duval, D. (2005b), "Ontology of learning object content structure," *Proceedings of the 12th International Conference on Artificial Intelligence in Education*, Amsterdam, pp. 322–329.
- Juerjens, J. (2003), *Secure Systems Development with UML*, Springer, Berlin, Heidelberg.
- Kalfoglou, Y. (2001), "Exploring ontologies," in *Handbook of Software Engineering and Knowledge Engineering*, Vol. 1, Fundamentals, ed. S.K. Chang, World Scientific, Singapore, pp. 863–887.
- Karp, R., Chaudhri, V., & Thomere, J. (1999), *XOL: An XML-Based Ontology Exchange Language*, Technical Report, SRI International [Online]. Available: <http://www.ai.sri.com/~pkarp/xol/xol.html> [Accessed: 2005, October 04].
- Kendall, E.F., Dutra, M.E., & McGuinness, D.L. (2002), "Towards a commercial ontology development environment," *Proceedings of the 1st International Semantic Web Conference (Posters and Demos)*, Chia, Sardinia.
- Klein, M. (2001), "Tutorial: The Semantic Web - XML, RDF, and relatives," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 26–28.
- Klein, M. & Visser, U. (2004), "Guest editors' introduction: Semantic Web Challenge 2003," *IEEE Intelligent Systems*, vol. 19, no. 3, pp. 31–33.
- Kleppe, A., Warmer, J., & Bast, W. (2003), *MDA Explained: The Model Driven Architecture: Practice and promise*, Addison-Wesley, Reading, MA.
- Knight, C., Gašević, D., & Richards, G. (2006), "An ontology-based framework for bridging learning design and learning content," *Educational Technology & Society*, vol. 9, no. 1, pp. 23–37.
- Knublauch, H. (2004), "Ontology-driven software development in the context of the Semantic Web: An example scenario with Protégé/Owl," *Proceedings of the International Workshop on the Model-Driven Semantic Web* (at the 8th International Conference on Enterprise Distributed Object Computing), Monterey, CA.
- Knublauch, H., Fergerson, R.W., Noy, N.F., & Musen, M.A. (2004), "The Protégé Owl plugin: An open development environment for Semantic Web applications," *Proceedings of the 3rd International Semantic Web Conference*, Hiroshima, pp. 229–243.
- Kobryn, C. (2001), "The road to UML 2.0: Fast track or detour," *Software Development Magazine*, April 2001, pp. 73–75.
- Kopena, J. & Regli, W.C. (2003), "DAMLJessKB: A tool for reasoning with the Semantic Web," *IEEE Intelligent Systems*, vol. 18, no. 3, pp. 74–77.
- Koper, R. (2001), *Modeling Units of Study from a Pedagogical perspective – The Pedagogical Metamodel Behind EML* [Online]. Available: <http://eml.ou.nl/introduction/docs/ped-metamodel.pdf> [Accessed: 2005, May 05].

- Koper, R. & Olivier, B. (2004), "Representing the learning design of units of learning," *Educational Technology & Society*, vol. 7, no. 3, pp. 97–111.
- Kremer, R. (1998), "Visual languages for knowledge representation," *Proceedings of the 11th Workshop on Knowledge Acquisition, Modeling and Management*, Banff, Canada.
- Kurtev, I. & van den Berg, K. (2003), "Model Driven Architecture based XML processing," *Proceedings of the ACM Symposium on Document Engineering*, Grenoble, France, pp. 246–248.
- Kurtev, I., Bézivin, J., & Aksit, M. (2002), "Technological spaces: An initial appraisal," *Proceedings of the Confederated International Conferences CoopIS, DOA, and ODBASE, Industrial Track*, Irvine, CA.
- Larman, C. (2001), *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, 2nd edition, Prentice-Hall, Upper Saddle River, NJ.
- Lassila, O. (1998), "Web metadata: A matter of semantics," *IEEE Internet Computing*, vol. 2, no. 4, pp. 30–37.
- Lenat, D.B. (1995), "CYC: A large-scale investment in knowledge infrastructure," *Communications of the ACM*, vol. 38, no. 11, pp. 33–38.
- Lenat, D.B. & Guha, R.V. (1990), *Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project*, Addison-Wesley, Reading, MA.
- Luke, S. & Heflin, J. (2000), *SHOE 1.01. Proposed Specification*, SHOE Project Technical Report, University of Maryland [Online]. Available: <http://www.cs.umd.edu/projects/plus/SHOE/spec.html> [Accessed: 2005, October 04].
- MacGregor, R. (1991), "Inside the LOOM classifier," *SIGART Bulletin*, vol. 2, no. 3, pp. 70–76.
- Maedche, A. & Staab, S. (2001), "Ontology learning for the Semantic Web," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 72–79.
- Manola, F. & Miller, E. (2004), *RDF Primer, W3C Recommendation* [Online]. Available: <http://www.w3.org/TR/REC-rdf-syntax/> [Accessed: 2005, October 04].
- Martin, D., Burstein, M., Hobbs, J., Lassila, O., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., & Sycara, K. (2004), *OWL-S: Semantic Markup for Web Services* [Online]. Available: <http://www.daml.org/services/owl-s/1.1/overview/> [Accessed: 2005, October 24].
- McBride, B. (2002), "Jena: A Semantic Web toolkit," *IEEE Internet Computing*, vol. 6, no. 6, pp. 55–59.
- McCarthy, J. (1999), *Concepts of Logical AI* [Online]. Available: <http://www-formal.stanford.edu/jmc/concepts-ai/concepts-ai.html> [Accessed: 2005, September 12].
- McGuinness, D.L. (2002), "Ontologies come of age," in *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*, eds. D. Fensel, J. Hendler, H. Lieberman, & W. Wahlster, MIT Press, Boston, MA, pp. 1–18.

- McIlraith, S.A., Son, T.C., & Zeng, H. (2001), "Semantic Web services," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46–53.
- Mellor, S.J., Clark, A.N., & Futagami, T. (2003a), "Guest editors' introduction: Model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 14–18.
- Mellor, S.J., Scott, K., Uhl, A., & Weise, D. (2003b), *MDA Distilled: Principles of Model-Driven Architecture*, Addison-Wesley, Reading, MA.
- Middleton, S.E., De Roure, D., & Shadbolt, N.R. (2004), "Ontology-based recommender systems," in *Handbook on Ontologies*, eds. S. Staab & R. Studer, Springer, Berlin, Heidelberg, pp. 477–498.
- Miller, J. & Mukerji, J. (2003), *MDA Guide Version 1.0*, OMG Document: omg/2003-05-01 [Online]. Available: http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf [Accessed: 2005, November 23].
- Minsky, M. (1985), *The Society of Mind*, Simon and Shuster, New York.
- Mitchell, F. (1998), *An Introduction to Knowledge Acquisition*, Dept. of Computer Science, University of Aberdeen, UK.
- Mizoguchi, R. & Kitamura, Y. (2001), "Knowledge systematization through ontology engineering – a key technology for successful intelligent systems," Invited paper, *Pacific-Asian Conference on Intelligent Systems*, Seoul.
- Myers, B.A. (1990), "Taxonomies of visual programming and program visualization," *Journal of Visual Languages and Computing*, vol. 1, no. 1, pp. 97–123.
- Naiburg, E.J. & Maksimchuk, R.A. (2001), *UML for Database Design*, Addison-Wesley, Reading, MA.
- Navigli, R., Velardi, P., & Gangemi, A. (2003), "Ontology learning and its application to automated terminology translation," *IEEE Intelligent Systems*, vol. 19, no. 1, pp. 22–31.
- Neches, R., Fikes, R.E., Finin, T., Gruber, T.R., Senator, T., & Swartout, W.R. (1991), "Enabling technology for knowledge sharing," *AI Magazine*, vol. 12, no. 3, pp. 36–56.
- Newell, A. (1982), "The knowledge level," *Artificial Intelligence*, vol. 18, no. 1, pp. 82–127.
- Niles, I. & Pease, A. (2001), "Towards a standard upper ontology," *Proceedings of the 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*, Ogunquit, ME, pp. 81–88.
- Noy, N.F. & McGuinness, D.L. (2001), *Ontology Development 101: A Guide to Creating Your First Ontology*, Knowledge Systems Laboratory, Stanford University, CA.
- Noy, N.F., Ferguson, R.W., & Musen, M.A. (2000), "The knowledge model of Protégé-2000: Combining interoperability and flexibility," *Proceedings of the 12th International Conference on Knowledge Engineering and Knowledge Management. Methods, Models, and Tools*, Juan-les-Pins, France, pp. 17–32.
- Noy, N.F., Sintek, M., Decker, S., Crubézy, M., Ferguson, R.W., & Musen, M.A. (2001), "Creating Semantic Web contents with Protégé-2000," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 60–71.
- ODM DSTC (2003), *Ontology Definition Metamodel*, DSTC Initial Submission, OMG Document ad/2003-08-01 [Online]. Available: <http://www.omg.org/cgi-bin/doc?ad/03-08-01> [Accessed: 2005, November 23].

- ODM Gentleware (2003), *Ontology Definition Metamodel*, Gentleware Initial Submission, OMG Document ad/03-08-09 [Online]. Available: <http://www.omg.org/cgi-bin/doc?ad/03-08-09> [Accessed: 2005, November 23].
- ODM IBM (2003), *Ontology Definition Metamodel*, IBM Initial Submission ad/03-07-02, OMG Document ad/03-07-02 [Online]. Available: <http://www.omg.org/cgi-bin/doc?ad/03-07-02> [Accessed: 2005, November 23].
- ODM Sandpiper&KSL (2003), *Ontology Definition Metamodel*, Sandpiper Software Inc and KSL Initial Submission, OMG Document ad/03-08-06 [Online]. Available: <http://www.omg.org/cgi-bin/doc?ad/03-08-06> [Accessed: 2005, November 23].
- OMG CORBA (2002), *UML Profile for CORBA, v 1.0*, OMG Document formal/02-04-01 [Online]. Available: <http://www.omg.org/cgi-bin/doc?formal/02-04-01> [Accessed: 2005, November 23].
- OMG DAF (2005), *Utility Management System Data Access Facility, v2.0.1*, OMG Document formal/05-06-01 [Online]. Available: <http://www.omg.org/cgi-bin/doc?formal/2005-06-03> [Accessed: 2005, November 23].
- OMG EAI (2004), *UML Profile for Enterprise Application Integration (EAI)*, OMG Document formal/04-03-26 [Online]. Available: <http://www.omg.org/cgi-bin/doc?formal/2004-03-26> [Accessed: 2005, October 23].
- OMG MOF (2002), *OMG Meta Object Facility (MOF) Specification v1.4*, OMG Document formal/02-04-03 [Online]. Available: <http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf> [Accessed: 2005, November 23].
- OMG MOF2 (2003), *OMG Meta Object Facility (MOF) 2.0 Core Specification Version 2.0: Final Adopted Specification*, OMG Document ptc/03-10-04 [Online]. Available: <http://www.omg.org/cgi-bin/apps/doc?ptc/03-10-04.pdf> [Accessed: 2005, November 23].
- OMG MOF-to-Text (2004), *OMG MOF Model to Text Transformation Language – Request For Proposal*, OMG Document ad/04-04-07 [Online]. Available: <http://www.omg.org/cgibin/apps/doc?ad/04-04-07.pdf> [Accessed: 2005, November 23].
- OMG ODM (2004), *Ontology Definition Metamodel – Preliminary Revised Submission to OMG RFP ad/2003-03-40, vol. 1* [Online]. Available: <http://codip.grci.com/odm/draft> [Accessed: 2005, November 23].
- OMG ODM RFP (2003), *OMG Ontology Definition Metamodel Request for Proposal*, OMG Document: ad/2003-03-40 [Online]. Available: <http://www.omg.org/cgi-bin/doc?ad/2003-03-40> [Accessed: 2005, November 23].
- OMG ODM RFP (2005), *Ontology Definition Metamodel – Third Revised Submission to OMG RFP ad/2003-03-40*, OMG Document ad/05-08-01 [Online]. Available: <http://www.omg.org/cgi-bin/apps/doc?ad/05-08-01.pdf> [Accessed: 2005, November 23].
- OMG QVT (2003), *OMG MOF Query/Views/Transformations First Revised Submission*, OMG Document ad/03-08-03 [Online]. Available: <http://www.omg.org/cgi-bin/apps/doc?ad/03-08-03.pdf> [Accessed: 2005, November 23].

- OMG UML (2003a), *OMG Unified Modeling Language (UML) Specification: Infrastructure, Version 2.0, Final Adopted Specification*, OMG Document ptc-03-09-15 [Online]. Available: <http://www.omg.org/cgi-bin/doc?ptc/2003-09-15> [Accessed: 2005, November 23].
- OMG UML (2003b), *OMG Unified Modeling Language: Superstructure, Version 2.0, Final Adopted Specification*, OMG Document ptc/03-08-02 [Online]. Available: <http://www.omg.org/cgi-bin/apps/doc?ptc/03-08-02.zip> [Accessed: 2005, November 23].
- OMG XMI (2002), *OMG XMI Specification, v1.2*, OMG Document formal/02-01-01 [Online]. Available: <http://www.omg.org/cgi-bin/doc?formal/2002-01-01> [Accessed: 2005, November 23].
- OWL Ontology Library (2005), *OWL Ontology Library, Stanford University* [Online]. Available: <http://protege.stanford.edu/plugins/owl/owl-library/> [Accessed: 2005, October 03].
- Payne, T. & Lassila, O. (2004), "Semantic Web services," *IEEE Intelligent Systems*, vol. 19, no. 4, pp. 14–15.
- Petri, C.A. (1962), *Kommunikation mit Automaten*, Schriften des IIM Nr. 3, Institut für Instrumentelle Mathematik, Bon.
- Petri Net Standards (2005), *Petri Net Standards* [Online]. Available: <http://www.informatik.uni-hamburg.de/TGI/PetriNets/standardisation/> [Accessed: 2005, November 23].
- Poole, J. (2002), *Common Warehouse Metamodel: An Introduction to the Standard for Data Warehouse Integration*, Wiley, New York.
- Preece, A. & Decker, S. (2002), "Intelligent Web services," *IEEE Intelligent Systems*, vol. 17, no. 1, pp. 15–17.
- Protégé (2005), *Protégé, Stanford University* [Online]. Available: <http://protege.stanford.edu/> [Accessed: 2005, November 27].
- Protégé Ontologies Library (2005), *ProtegeOntologiesLibrary* [Online]. Available: <http://protege.cim3.net/cgi-bin/wiki.pl?ProtegeOntologiesLibrary> [Accessed: 2005, October 03].
- Protégé UML (2004), *Protégé UML Backend: Store Protégé Ontologies in UML* [Online]. Available: <http://protege.stanford.edu/plugins/uml> [Accessed: 2005, November 23].
- Protégé XMI (2004), *Protégé XMI Backend: Store Protégé Ontologies in MOF Metamodels* [Online]. Available: <http://protege.stanford.edu/plugins/xmi/> [Accessed: 2005, November 23].
- Pulman, S.G. (1996), "Controlled language for knowledge representation," *Proceedings of the 1st International Workshop on Controlled Language Applications*, Katholieke Universiteit Leuven, Belgium, pp. 233–242.
- Rapaport, W.J. (2000), "Cognitive science" in *Encyclopedia of Computer Science*, 4th edition, eds. A. Ralston, E.D. Reilly, & D. Hemmendinger, Grove's Dictionaries, New York, pp. 227–233.
- Reisig, W. (1985), *Petri Nets: An Introduction*, Springer-Verlag, Berlin, Germany.
- Richards, G. (2002), "Editorial: The challenges of the learning object paradigm," *Canadian Journal of Learning and Technology*, vol. 28, no. 3, pp. 3–10.

- RuleML (2005), *RuleML – The Rule Markup Initiative* [Online]. Available: <http://www.ruleml.org/> [Accessed: 2005, September 20].
- Rumbaugh, J., Jacobson, I., & Booch, G. (1998), *The Unified Modeling Language Reference Manual*, Addison-Wesley, Reading, MA.
- Rumbaugh, J., Jacobson, I., & Booch, G. (2004), *The Unified Modeling Language Reference Manual*, 2nd edition, Addison-Wesley, Reading, MA.
- Russell, S. & Norvig, P. (2002), *Artificial Intelligence: A Modern Approach*, 2nd edition, Prentice-Hall, Englewood Cliffs, NJ.
- Schreiber, G., Wielinga, B., de Hoog, R., Akkermans, H., & van de Velde, W. (1994), “CommonKADS: A comprehensive methodology for KBS development,” *IEEE Expert*, vol. 9, no. 6, pp. 28–37.
- Scott Cost, R., Finin, T., Joshi, A., Peng, Y., Nicholas, C., Soboroff, I., Chen, H., Kagal, L., Perich, F., Zou, Y., & Tolia, S. (2002), “ITtalks: A case study in the Semantic Web and DAML+OIL,” *IEEE Intelligent Systems*, vol. 17, no. 1, pp. 40–47.
- Seidewitz, E. (2003), “What models mean,” *IEEE Software*, vol. 20, no. 5, pp. 26–32.
- Selic, B. (2003), “The pragmatics of model-driven development,” *IEEE Software*, vol. 20, no. 5, pp. 19–25.
- Sheth, A., Ramakrishnan, C., & Thomas, C. (2005), “Semantics for the Semantic Web: The implicit, the formal and the powerful,” *International Journal on Semantic Web & Information Systems*, vol. 1, no. 1, pp. 1–18.
- Sigel, J. (2001), *Developing in OMG’s Model-Driven Architecture, Revision 2.6*, Object Management Group White Paper [Online]. Available: <ftp://ftp.omg.org/pub/docs/-omg/01-12-01.pdf> [Accessed: 2005, November 23].
- Sirin, E., Parsia, B., & Hendler, J. (2004), “Filtering and selecting Semantic Web services with interactive composition techniques,” *IEEE Intelligent Systems*, vol. 19, no. 4, pp. 42–49.
- Smith, M.K., Welty, C., & McGuinness, D.L. (2004), *OWL Web Ontology Language Guide, W3C* [Online]. Available: <http://www.w3.org/TR/owl-guide/> [Accessed: 2005, October 1].
- Sowa, J.F. (2000), *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Brooks Cole, Pacific Grove, CA.
- SPARQL (2005), *SPARQL Query Language for RDF, W3C* [Online]. Available: <http://www.w3.org/TR/rdf-sparql-query/> [Accessed: 2005, October 11].
- Staab, S. & Studer, R., eds. (2004), *Handbook on Ontologies*, Springer, Berlin, Heidelberg.
- Stanford Encyclopedia of Philosophy (2005), *Stanford Encyclopedia of Philosophy* [Online]. Available: <http://plato.stanford.edu/entries/cognitive-science/> [Accessed: 2005, September 10].
- Štrbac, P. (2002), *An Approach to Modeling Communication Protocols by Using Upgraded Petri Nets*, Military Academy, Belgrade, Serbia and Montenegro.
- Stuckenschmidt, H. & van Harmelen, F. (2005), *Information Sharing on the Semantic Web*, Springer, Berlin, Heidelberg.

- SUO WG (2005), *Standard Upper Ontology Working Group (SUO WG)*, *IEEE P1600.1* [Online]. Available: <http://suo.ieee.org.proxy.lib.sfu.ca/index.html> [Accessed: 2005, October 9].
- SW Activity (2005), *Semantic Web Activity Statement*, W3C [Online]. Available: <http://www.w3.org/2001/sw/Activity> [Accessed: 2004, October 11].
- Swartout, W.R. & Tate, A. (1999), "Guest editors' introduction: Ontologies," *IEEE Intelligent Systems*, vol. 14, no. 1, pp. 18–19.
- Tallis, M., Goldman, N.M., & Balzer, R.M. (2002), "The Briefing Associate: easing authors into the Semantic Web," *IEEE Intelligent Systems*, vol. 17, no. 1, pp. 26–32.
- Uhl, A. & Ambler, S.W. (2003), "Point/Counterpoint," *IEEE Software*, vol. 20, no. 5, pp. 70–73.
- Ullrich, C. (2005), "The learning-resource-type is dead, long live the learning-resource-type!" *Learning Objects and Learning Designs*, vol. 1, no. 1, pp. 7–15.
- van der Vet, P.E. & Mars, N.J.I. (1998), "Bottom-up construction of ontologies," *IEEE Transactions on Knowledge and Data Engineering*, vol. 10, no. 4, pp. 513–526.
- Verbert, K. & Duval, E. (2004), "Towards a global architecture for learning objects: A comparative analysis of learning object content models," *In Proceedings of the 16th World ED-MEDIA 2004 Conference*, Lugano, Switzerland, pp. 202–209.
- Verbert, K., Klerkx, J., Meire, M., Najjar, J., & Duval, E. (2004), "Towards a global component architecture for learning objects: An ontology based approach," *Proceedings of OTM 2004 Workshop on Ontologies, Semantics and E-learning*, Agia Napa, Cyprus, pp. 713–722.
- Vianu, V. (1997), "Rule-based languages," *Annals of Mathematics and Artificial Intelligence*, vol. 19, no. 1–2, pp. 215–259.
- Vinoski, S. (2002), "Web services interaction models, part 1: Current practice," *IEEE Internet Computing*, vol. 6, no. 3, pp. 90–92.
- Wagner, G., Tabet, S., & Boley, H. (2003), "MOF-RuleML: the abstract syntax of RuleML as a MOF model, Integrate 2003," *OMG Meeting, Boston* [Online]. Available: <http://www.omg.org/docs/br/03-10-02.pdf> [Accessed: 2005, September 20].
- Welty, C.A. (1996), *An Integrated Representation for Software Development and Discovery*, Rensselaer Polytechnic Institute, New York.
- Wiley, D.A. (2002), "Connecting learning objects to instructional design theory: A definition, a metaphor, and a taxonomy," in *The Instructional Use of Learning Objects*, ed. D.A. Wiley, Agency for Instructional Technology and Association for Educational Communications of Technology, Bloomington, IN, pp. 1–35.
- WordNet (2005), *WordNet*, Princeton University [Online]. Available: <http://wordnet.princeton.edu.proxy.lib.sfu.ca/w3wn.html> [Accessed: 2005, October 5].

- Wuwongse, V., Anutariya, C., Akama, K., & Nantajeewarawat, E. (2002), "XML declarative description: A language for the Semantic Web," *IEEE Intelligent Systems*, vol. 17, no. 1, pp. 54–65.
- XML (2005), *Extensible Markup Language (XML) 1.0*, W3C, 3rd edition [Online]. Available: <http://www.w3.org/TR/2004/REC-xml-20040204/> [Accessed: 2005, October 12].
- XML Schema (2005), *XML Schema*, W3C [Online]. Available: <http://www.w3.org/XML/Schema> [Accessed: 2005, October 12].
- XPetal (2002), *The XPetal Tool* [Online]. Available: <http://www.langdale.com.au/styler/xpetal/> [Accessed: 2005, November 23].
- Zahniser, R.A. (1993), "Design by walking around," *Communications of The ACM*, vol. 36, no. 10, pp. 115–123.
- Zhong, N., Liu, J., & Yao, Y., eds. (2003), *Web Intelligence*, Springer, Berlin, Heidelberg.

Index

A

- ABox, 25
- abstraction, 131
 - context-dependent, 131
- AI. *See* artificial intelligence
- AIFB OWL DL metamodel, 159
- AIR, 255
 - basic idea, 256
 - building blocks, 258
 - metadata repository, 259
 - structure, 261
 - model base, 259
 - plug-ins, 262
 - role of XML, 264
 - workbench, 262
- ALoCoM Content Structure ontology, 282
- ALoCoM Content Type ontology, 282
- analogies, 11
- artificial intelligence, 3
 - integration with software engineering, 256
 - tools, 255
- artificial neural network, 9

B

- Briefing Associate, 70, 106

C

- class diagram, 230
- CLASSIC, 34
- CMOF, 115, 261
- cognitive science, 4, 7

- Common Warehouse Metamodel, 118
- CommonKADS, 38
- commonsense knowledge, 13
- Complete MOF, 115, 261
- concept map languages, 34
- concept maps, 18, 34
- concepts, 10, 25
- conceptual framework, 52
- Conceptual Graphs, 34
- conceptual modeling spaces, 134
- concrete modeling spaces, 134
- connectionist paradigm, 9
- containment tree, 231
- content theory, 52
- controlled languages, 36
- CREAM framework, 97, 98
- CYC, 105

D

- DAML+OIL, 59, 87, 88, 89, 90, 91
- DAML-ONT, 87
- DATR, 35
- declarative (model-theoretic) languages, 30
- declarative knowledge, 12
- degree of belief, 5
- derivation rules, 31
- description logics, 25
- design patterns, 68
- diagrams tree, 231
- DUET (DAML UML Enhanced Tool), 164

E

Eclipse, 256
Eclipse Modeling Framework, 183, 229, 261
EMF. *See* Eclipse Modeling Framework
EMOF, 115, 261
EMOF-based repository, 262
Essential MOF, 115, 261
extended UML metamodel for ontologies, 150, 153
extensible stylesheet language transformation (XSLT), 32

F

first-order logic, 23
first-order predicate calculus, 23
formal logic, 10
Foxtrot, 71, 72
frame-based representation languages, 27
frames, 19
fuzzy facts, 16
fuzzy rules, 17
fuzzy set, 16

G

general modeling architecture, 130
glossary, 51
GOOD OLD AI Ontology Definition Metamodel, 160
GOOD OLD AI Ontology UML Profile, 160, 207, 267

H

heuristic knowledge, 13
higher-order logic, 23

I

images, 11
implementation level, 41
individuals, 25, 201

inexact and uncertain knowledge, 5, 13
inheritance tree, 231
integrity rules, 31
intelligent agents, 79, 102
intelligent Web services, 101, 102
interoperability, 53

J

Java Metadata Interchange (JMI), 260

K

KIF, 24, 59
knowledge, 3
knowledge acquisition, 4, 37
knowledge engineering, 36
knowledge engineers, 37
Knowledge Interchange Format (KIF), 24, 59
knowledge level, 41
knowledge representation, 3
knowledge representation languages, 5, 19
knowledge representation techniques, 5, 14
knowledge retrieval, 4
knowledge sharing and reuse, 53, 54
knowledge storing, 4
KRS, 34

L

LOCO (ontology), 286
LOCO-Cite (ontology), 288
logical level, 42

M

MagicDraw, 230
creating new ontology, 234
individuals, 244
ontology classes, 237
ontology properties, 240
statements, 246
Magpie, 69

- mappings
 - MDA-based languages and ontologies, 212
 - ontology and MOF modeling spaces, 212
 - transformations, 214, 217
 - OWL and Ontology UML Profile, 217
 - MDA. *See* Model Driven Architecture
 - membership function, 16
 - mental representations, 5
 - metadata, 72
 - metadata language, 73
 - metaknowledge, 13
 - metalanguage, 35
 - metametamodel, 110, 113, 130
 - Meta-Object Facility, 110, 114
 - metamodel, 72, 73, 110, 113, 115, 117, 130
 - AIFB OWL DL metamodel, 159
 - Common Warehouse Metamodel, 118
 - explicit, 130
 - GOOD OLD AI Ontology Definition Metamodel, 160
 - implicit, 129
 - Ontology Definition Metamodel. *See* Ontology Definition Metamodel
 - Protégé MOF-compatible metamodel, 161, 163
 - self-defined, 131
 - super-metamodel, 131
 - metamodeling, 72
 - Meta-Object Facility, 110, 114, 131, 137
 - Complete MOF, 115, 261
 - Essential MOF, 115, 261
 - Methontology, 67
 - Methontology framework, 66
 - model, 109, 110
 - characteristics, 109
 - definitions, 128
 - Model Driven Architecture, 109, 111, 112, 173
 - four-layer architecture, 112
 - Model Driven Development, 109
 - model transformations, 138
 - modeling architecture, informal, 129
 - modeling spaces, 127
 - conceptual modeling spaces, 134
 - concrete modeling spaces, 134
 - duality, 132
 - essentials, 131
 - MOF modeling space, 137
 - orthogonal modeling spaces, 136
 - parallel spaces, 135
 - RDF(S) modeling space, 137
 - transformations, 138
 - modeling the real world, 128
 - modus ponens, 10
 - MOF. *See* Meta-Object Facility
 - MOF modeling space, 137
 - MOF XML Metadata Interchange (XMI), 125, 258
- N**
- natural language, 35, 48
 - NetBeans MDR, 229, 247, 261
 - neural connections, 11
 - notation for knowledge representation, 5
- O**
- object–attribute–value triplet, 15, 92
 - ODM. *See* Ontology Definition Metamodel
 - OIL, 59, 87, 89
 - Core OIL, 87
 - Heavy OIL, 87
 - Instance OIL, 87
 - Standard OIL, 87
 - OKBC, 39
 - ontological analysis, 53
 - ontological engineering, 58
 - ontological knowledge, 14
 - ontological level, 76
 - ontology, 45, 46, 47, 48, 173
 - ALOCoM Content Structure, 282
 - ALOCoM Content Type, 282

- LOCO, 286
- LOCO-Cite, 288
- Petri net ontology, 267
 - extension for upgraded Petri nets, 275
- ontology classes, 201
- Ontology Definition Metamodel, 119, 172, 181
 - AIFB OWL DL metamodel, 159
 - GOOD OLD AI Ontology Definition Metamodel, 160
 - joint submission, 183
 - metamodels, 181
 - official proposals, 176
 - DSTC, 181
 - Gentleware, 181
 - IBM, 181
 - Sandpiper Software Inc and KSL, 181
 - overview, 174
- OWL metamodel
 - Ontology, 195
 - owl:AllDifferent, 195
 - owl:DatatypeProperty, 193
 - owl:differentFrom, 195
 - owl:distinctMembers, 195
 - owl:FunctionalProperty, 194
 - owl:InverseFunctionalProperty, 194
 - owl:ObjectProperty, 193
 - owl:Ontology, 195
 - owl:Restriction, 194
 - owl:SymmetricProperty, 194
 - owl:Thing, 191
 - owl:TransitiveProperty, 194
 - OWLAllDifferent, 195
 - OWLallValuesFrom, 195
 - OWLAnnotationProperty, 197
 - OWLbackwardCompatibleWith, 195
 - OWLcardinality, 195
 - OWLClass, 191
 - OWLcomplementOf, 192
 - OWLDataRange, 195
 - OWLDatatypeProperty, 193
 - OWLdifferentFrom, 195
 - OWLdisjointWith, 192
 - OWLdistinctMembers, 195
 - OWLequivalentClass, 192
 - OWLhasValue, 195
 - OWLIncompatibleWith, 195
 - OWLIntersectionOf, 192
 - OWLmaxCardinality, 195
 - OWLminCardinality, 195
 - OWLObjectProperty, 193
 - OWLoneOf, 192, 195
 - OWLOntology, 195
 - OWLRestriction, 193, 194
 - OWLsomeValuesFrom, 195
 - OWLUnionOf, 192
- Property, 194
- RDFS metamodel
 - Ontology, 190
 - rdf:object, 186
 - rdf:predicate, 186
 - rdf:Property, 187
 - rdf:Statement, 186
 - rdf:subject, 186
 - rdf:type, 185
 - RDFAlt, 188
 - RDFBag, 188
 - RDFfirst, 189
 - RDFList, 189
 - RDFobject, 186
 - RDFpredicate, 186
 - RDFProperty, 185, 187
 - rdfs:Class, 185
 - rdfs:Container, 188
 - rdfs:domain, 187
 - rdfs:range, 187
 - rdfs:Resource, 184
 - localName, 185
 - namespace, 185
 - uri, 185
 - rdfs:subClassOf, 186
 - rdfs:subPropertyOf, 188
 - RDFSClass, 185
 - RDFSContainer, 188
 - RDFSdomain, 188
 - RDFSeq, 188
 - RDFSisDefinedBy, 189
 - RDFSLiteral, 185
 - RDFSmember, 188
 - RDFSrange, 188
 - RDFSResource, 185
 - RDFSrest, 189
 - RDFSseeAlso, 189
 - RDFSsubClassOf, 185
 - RDFSsubPropertyOf, 188
 - RDFStatement, 185, 186

- RDFsubject, 186
 - RDFtype, 192
 - RDFvalue, 189
 - RDFXMLLiteral, 185
 - Request for Proposals, 174, 181
 - ontology development, 173
 - ontology development
 - environments, 59, 60
 - ontology development methodology, 65
 - ontology development tools, 58
 - ontology learning, 63, 64
 - ontology markup languages, 59
 - ontology modeling, 229
 - ontology representation languages, 52, 59
 - Ontology UML Profile, 119, 174, 201
 - AIFB Ontology UML Profile, 159
 - ALOCoM Content Structure, 282
 - ALOCoM Content Type, 282
 - design rationale, 178
 - GOOD OLD AI Ontology UML Profile, 160, 207, 267
 - LOCO, 286
 - LOCO-Cite, 288
 - Petri net ontology, 267
 - extension for upgraded Petri nets, 275
 - stereotypes
 - «domain», 205
 - «equivalentClass», 202
 - «OntClass», 204
 - «OWLAllDifferent», 201
 - «OWLClass», 201, 202
 - «OWLDatatypeProperty», 205, 206
 - «OWLObjectProperty», 204, 205
 - «OWLRestriction», 201
 - «range», 205
 - «RDFdomain», 206
 - «RDFobject», 206
 - «RDFProperty», 204
 - «RDFrange», 206
 - «RDFSClass», 201
 - «RDFSProperty», 203
 - «RDFsubject», 206
 - ontology-learning tools, 63
 - Open Knowledge Base
 - Connectivity, 39
 - orthogonal modeling spaces, 136
 - OUP. *See* Ontology UML Profile
 - OWL, 50, 51, 60, 81, 90, 92, 181, 190
 - ODM metamodel, 190
 - ODM metamodel class hierarchy, 191
 - OWL DL, 91, 191
 - OWL Full, 92, 191
 - OWL Lite, 91, 191
 - OWL-S, 61, 103, 104
- P**
- parallel modeling spaces, 135
 - Petri net ontology, 267
 - extension for upgraded Petri nets, 275
 - POJO, 262
 - Poseidon for UML, 247
 - ontology classes, 249
 - ontology individuals, 250
 - ontology statements, 250
 - procedural knowledge, 12
 - production rules, 31
 - production system languages, 30
 - propositional logic, 22
 - Protégé, 61, 161
 - MOF-compatible metamodel, 161, 162, 163
 - UML back end, 161, 162, 251
 - UML tools and Protégé, 251
 - XMI back end, 161
- Q**
- Quickstep, 71, 72
- R**
- RDF, 59, 81, 84, 85, 89, 92, 93, 97, 99
 - RDF Schema, 59, 81, 84, 85, 87, 88, 89, 92, 137, 181

- ODM metamodel, 184
- ODM metamodel class hierarchy, 184
- RDF(S). *See* RDF Schema
- RDF(S) modeling space, 137
- RDFS. *See* RDF Schema
- reaction rules, 31
- reasoning, 4
- resolution principle, 10
- roles of knowledge representation, 6
- rule-based representation languages, 29
- RuleML, 31, 32
- rules, 10, 17

S

- semantic annotation, 96
- semantic interoperability, 95
- semantic markup, 96
- semantic networks, 18, 34, 49, 85
- Semantic Web, 79, 80, 81, 95, 96, 102, 104
- Semantic Web languages, 59, 81
- Semantic Web layer-cake, 81, 82
- Semantic Web services, 100, 102, 106
- service-oriented architecture, 100
- sharing UML models between UML tools and Protégé, 251
- SPARQL, 92, 93, 94
- Standard Upper Ontology, 74, 75
- stripping pattern, 84, 86
- structural knowledge, 13
- SUO, 74, 75
- super-metamodel, 131
- symbolic paradigm, 9

T

- taxonomy, 52
- TBox, 25
- technical spaces, 139
- thesaurus, 51
- transformation rules, 31
- types of human knowledge, 4, 11

U

- UML. *See* Unified Modeling Language
- UML profiles, 120, 232
 - AIFB Ontology UML Profile, 159
 - GOOD OLD AI Ontology UML Profile, 160, 207, 267
 - Ontology UML Profile, 119, 174, 201, 267
 - UML profile for CORBA, 121
 - UML profile for database design, 123
 - UML profile for modeling Web applications, 122
 - UML profile for modeling XML Schema, 122
- UML tools, 229
 - constraints, 229
 - MagicDraw, 230
 - ontology modeling, 229
 - Poseidon for UML, 247
 - problems, 229
- UML XML Metadata Interchange (XMI), 124, 125, 229
- uncertain facts, 15
- uncertain rules, 17
- Unified Modeling Language, 34, 49, 61, 73, 117
 - relations to ontology languages, 151, 152, 170
 - transformations to and from ontology languages, 156
- Unified Ontology Language, 155
- Unisys' CIM, 261

V

- visual languages for knowledge representation, 32
- Visual Ontology Modeler, 167
- vocabulary, 46, 51

W

W3C, 79, 80
Web Ontology Language. *See* OWL
Web services, 79, 100, 101
Web-based ontology languages, 59,
81
WordNet, 64

X

XMI. *See* XML Metadata
Interchange
XML, 81, 82, 97
XML Metadata Interchange, 34, 61,
123, 124
MOF XMI, 125, 258
UML XMI, 124, 125, 229
XML Schema, 81, 83, 89
Xpetal, 165
XSLT, 32