NeOn: Lifecycle Support for Networked Ontologies

Integrated Project (IST-2005-027595)

Priority: IST-2004-2.4.7 — "Semantic-based knowledge and content systems"

D1.1.2 Updated Version of the Networked Ontology Model

Deliverable Co-ordinator: **Peter Haase**

Deliverable Co-ordinating Institution: **Universität Karlsruhe (TH) (UKARL)**

Other Authors: **Saartje Brockmans (UKARL), Raul Palma (UPM), Jérôme Euzenat (INRIA), Mathieu d'Aquin (OU)**

Document Identifier:	NEON/2007/D1.1.2/1.0	Date due:	31.08.2007
Class Deliverable:	NEON EU-IST-2005-027595	Submission date:	31.08.2007
Project start date	March 1, 2006	Version:	1.0
Project duration:	4 years	State:	Final
		Distribution:	Public

NeOn Consortium

This document is part of the NeOn research project funded by the IST Programme of the Commission of the European Communities by the grant number IST-2005-027595. The following partners are involved in the project:

Open University (OU) – Coordinator Knowledge Media Institute – KMi Berrill Building, Walton Hall Milton Keynes, MK7 6AA United Kingdom Contact person: Martin Dzbor, Enrico Motta E-mail address: {m.dzbor, e.motta}@open.ac.uk	**Universität Karlsruhe – TH (UKARL)** Institut für Angewandte Informatik und Formale Beschreibungsverfahren – AIFB D-76128 Karlsruhe Germany Contact person: Peter Haase E-mail address: pha@aifb.uni-karlsruhe.de
Universidad Politécnica de Madrid (UPM) Campus de Montegancedo 28660 Boadilla del Monte Spain Contact person: Asunción Gómez Pérez E-mail address: asun@fi.ump.es	**Software AG (SAG)** Uhlandstrasse 12 64297 Darmstadt Germany Contact person: Walter Waterfeld E-mail address: walter.waterfeld@softwareag.com
Intelligent Software Components S.A. (ISOCO) Calle de Pedro de Valdivia 10 28006 Madrid Spain Contact person: Jesús Contreras E-mail adress: jcontreras@isoco.com	**Institut 'Jožef Stefan' (JSI)** Jamova 39 SL–1000 Ljubljana Slovenia Contact person: Marko Grobelnik E-mail address: marko.grobelnik@ijs.si
Institut National de Recherche en Informatique et en Automatique (INRIA) ZIRST – 665 avenue de l'Europe Montbonnot Saint Martin 38334 Saint-Ismier France Contact person: Jérôme Euzenat	**University of Sheffield (USFD)** Dept. of Computer Science Regent Court 211 Portobello street S14DP Sheffield United Kingdom Contact person: Hamish Cunningham
Universität Kolenz-Landau (UKO-LD) Universitätsstrasse 1 56070 Koblenz Germany Contact person: Steffen Staab E-mail address: staab@uni-koblenz.de	**Consiglio Nazionale delle Ricerche (CNR)** Institute of cognitive sciences and technologies Via S. Marino della Battaglia 44 – 00185 Roma-Lazio Italy Contact person: Aldo Gangemi E-mail address: aldo.gangemi@istc.cnr.it
Ontoprise GmbH. (ONTO) Amalienbadstr. 36 (Raumfabrik 29) 76227 Karlsruhe Germany Contact person: Jürgen Angele E-mail address: angele@ontoprise.de	**Food and Agriculture Organization of the United Nations (FAO)** Viale delle Terme di Caracalla 00100 Rome Italy Contact person: Marta Iglesias E-mail address: marta.iglesias@fao.org
Atos Origin S.A. (ATOS) Calle de Albarracín, 25 28037 Madrid Spain Contact person: Tomás Pariente Lobo E-mail address: tomas.parientelobo@atosorigin.com	**Laboratorios KIN, S.A. (KIN)** C/Ciudad de Granada, 123 08018 Barcelona Spain Contact person: Antonio López E-mail address: alopez@kin.es

Work package participants

The following partners have taken an active part in the work leading to the elaboration of this document, even if they might not have directly contributed writing parts of this document:

- UKARL
- UPM
- INRIA
- OU
- CNR

Change Log

Version	Date	Amended by	Changes
0.1	29.6.2007	Saartje Brockmans	Initial creation of document
0.2	03.08.2007	Saartje Brockmans	Metamodels added, MOF and partial motivation added
0.3	05.08.2007	Peter Haase	Overall structure
0.4	15.08.2007	Raul Palma	OMV
0.9	17.08.2007	Peter Haase	Final Write Up
1.0	03.10.2007	Peter Haase	Addressing Reviewer's comments

2006–2007 © Copyright lies with the respective authors and their institutions.

Executive Summary

In this deliverable we present an updated version of the NeOn networked ontology model for representing and managing relations between multiple networked ontologies. The deliverable supersedes the previous version of the initial version of the networked ontology model presented in D1.1.1.

To achieve flexibility and applicability, we define the networked ontology model using a metamodeling approach. The metamodeling features of Model Driven Architecture provide the means for the specification of modeling languages in a standardized, platform independent manner. In short, the Meta Object Facility (MOF) provides the language for creating metamodels, UML defines the language for creating models corresponding to specific metamodels. Defining the networked ontology model in terms of a MOF compliant metamodel yields a number of advantages, including (1) interoperability with software engineering approaches, (2) model transformations, to support the automated generation of APIs, exchange syntaxes, etc., (3) reuse of UML for modeling, and (4) independence from particularities of specific formalisms, enabling the ability to support currently competing formalisms (e.g. in the case of mapping languages), for which no standard exists yet.

The metamodel consists of individual modules for the individual aspects of networked ontologies. The main modules are: (1) a metamodel for the OWL ontology language, (2) a rule metamodel, (3) a metamodel for ontology mappings, and (4) a metamodel for modular ontologies. The metamodel is grounded by translations to specific logical formalisms that define the semantics of the networked ontology model.

With respect to the metamodel, the changes over the initial version of the networked ontology model are mainly:

- We provide a metamodel for the proposed OWL 1.1 language.
- In addition to the SWRL metamodel, we also provide a metamodel for F-Logic.
- For the generic mapping metamodel, we provide specialized metamodels for specific mapping languages.

Additionally, we provide an updated version of the Ontology Metadata Vocabulary OMV, our vocabulary to describe ontology metadata in a network of ontologies, Metadata here refers to data *about* the actual ontologies, as opposed to the content contained *in* an ontology, e.g. the provenance and authorship of an ontology, but also dependencies from other ontologies, etc. Metadata plays an important role in describing and managing the networked ontologies. Its purpose is to be able to clarify the relations between the available ontologies so that they are easy to search, to characterize and to maintain.

Contents

2006–2007 © Copyright lies with the respective authors and their institutions.

2006–2007 © Copyright lies with the respective authors and their institutions.

List of Tables

2006–2007 © Copyright lies with the respective authors and their institutions.

2006–2007 © Copyright lies with the respective authors and their institutions.

List of Figures

2006–2007 © Copyright lies with the respective authors and their institutions.

Part I

Foundations

2006–2007 © Copyright lies with the respective authors and their institutions.

Chapter 1

Introduction

1.1 The NeOn Big Picture

Next generation semantic applications will be characterized by a large number of ontologies, some of them constantly evolving. As the complexity of semantic applications increases, more and more knowledge will be embedded in applications, typically drawn from a wide variety of sources. This new generation of applications will thus likely rely on ontologies embedded in a network of already existing ontologies. Ontologies and metadata will have to be kept up to date when application environments and users' needs change. We argue that in this scenario it will become prohibitively expensive for people to directly adopt the current approach to semantic integration, where the expectation is to produce a single, globally consistent semantic model that serves the needs of application developers and fully integrates a number of pre-existing ontologies In contrast to the current model, future applications will very likely rely on networks of contextualized ontologies, which are usually locally, but not globally consistent.

This report is part of the work performed in WP 1 on "Dynamics of Networked Ontologies". The goal of this work package is to develop an integrated approach for the evolution process of networked ontologies and related metadata. For the individual phases of the process we will develop new methods that consider the complex relationships in a network of ontologies. These include dependencies, mappings, different versions and also take possible inconsistencies into account. Where possible, these methods will be supported by automatic or semi-automatic approaches based on natural language processing (NLP), data mining and machine learning techniques, and the corresponding tools will be developed to support the evolution process. Specific goals in this workpackage include support for:

1. representing, managing and interpreting dependencies between multiple networked ontologies
2. evolution of networked ontologies in exploiting various models of change propagation, which have different applicabilities depending on the model of coordination and control
3. maintaining partial/local consistency of a set of networked ontologies, which might not be globally consistent
4. evolving metadata along with changing ontologies and predicting future structural changes in ontologies.

In order to achieve the goals mentioned above it is crucial that a toolkit designed within NeOn offers a basic set of functionalities, which are also essential for the use cases. These are: (a) Integration of a large number of heterogeneous information sources (b) Maintenance of ontologies to reflect changes in the domain (c) Consensus forming in a decentralized scenario.

As shown in Figure 1.1, WP1 belongs to the central part of the research and development WPs in NeOn. The tasks of WP1 are heavily inter-related with other work packages. The most fundamental contribution of WP1, and in particular Task 1.1, is to provide the networked ontology model and its related dynamic aspects

to other work package partners. As a consequence, a major objective in the definition of the networked ontology model is the consideration of requirements from other work packages. We will closely collaborate with the case study partners to apply our technologies within the case studies.

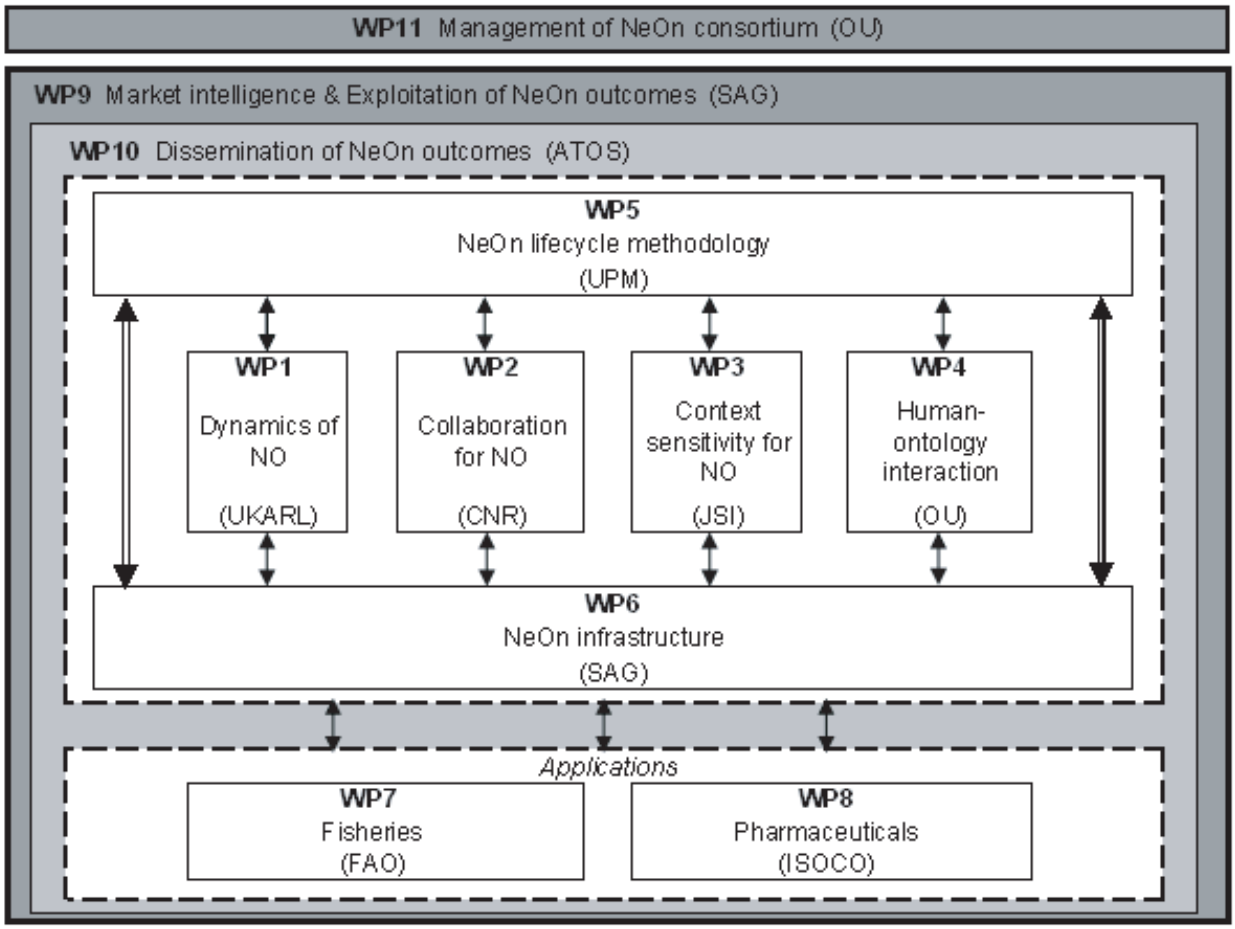


Figure 1.1: Relationships between different workpackages in NeOn

1.2 Networked Ontologies

The goal of this deliverable is to define and formalize the notion of networked ontologies in order to make it applicable throughout the NeOn project. Before formally defining the model, we will first informally discuss the notion of a networked ontology.

1.2.1 The current situation

The OWL ontology language is now well established as the standard ontology language for representing knowledge on the web. At the same time, rule languages, such as F-Logic, have shown their practical applicability in industrial environments. Often, ontology-based applications require features from both paradigms—the description logics and the rule paradigm—but their combination remains difficult. This is not only due to the semantic impedance mismatch [HPPSH05], but already because of the disjoint landscape in ontology engineering tools that typically support either the one or the other paradigm.

Additionally, to address distributed and networked scenarios as envisioned in NeOn, current ontology languages lack a number of features to explicitly express the relationships between ontologies and their el-

2006–2007 © Copyright lies with the respective authors and their institutions.

ements. These features include in particular formalisms for expressing *modular ontologies* and *mappings* (also called alignments) between ontologies that are heterogeneous on various respects.

Finally, in the current situation we lack the means to express information *about* ontologies and the relationships between them. Ontologies are distributed over the web, sometimes available directly, sometimes hidden within corporate networks (see Figure 1.2[1]). These ontologies are related to each others, but this remains difficult to assess:

- some are simple copies of other ones (it is hard to know which one is the master copy);
- some are versions of others (it is hard to know which one came first);
- some are used jointly with others (this information is hidden in applications);
- some are imported by others (this can be found through owl:imports).

Moreover, the ontologies have different characteristics:

- they are written in different languages and with different language variants;
- they use labels in different natural languages;
- they are built for different purposes.

In consequence, each application developer has to manually assemble ontologies, modify, import and export them, resulting in yet other disconnected, un-contextualized ontologies. Each time one of these ontologies is changed, at best, the developer will propagate the changes, at worst, they will not be taken into account at all.

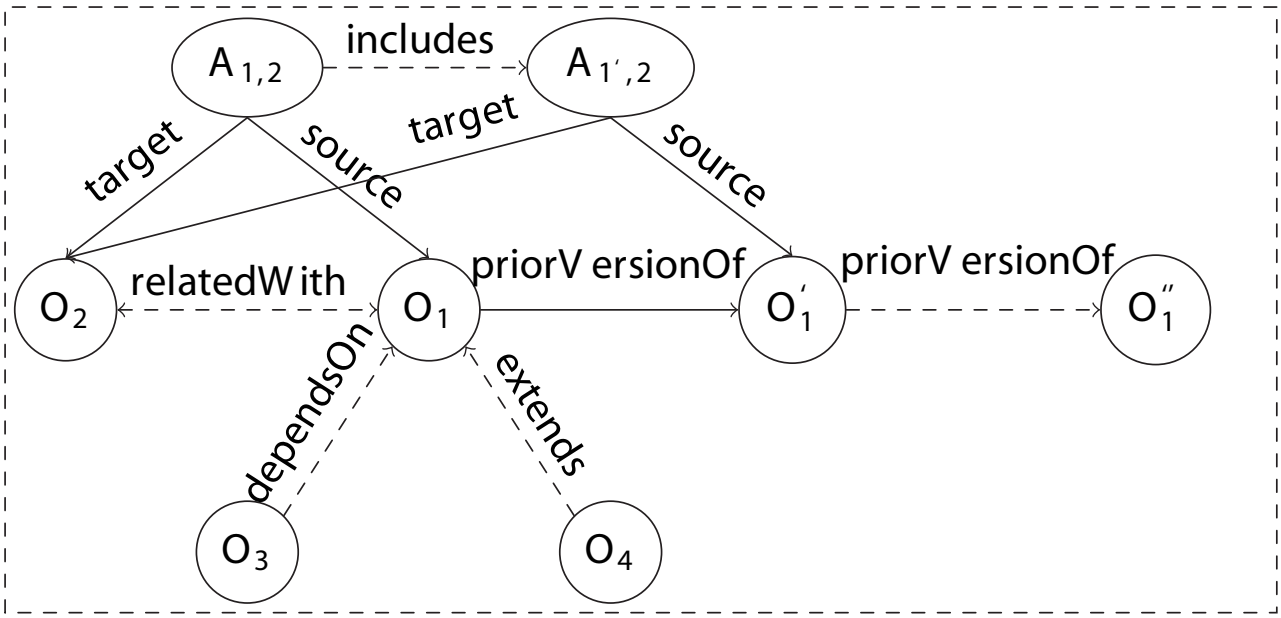


Figure 1.2: The current state of networked ontologies on the web. Dashed relations are in fact not explicitly recorded on the web but can be inferred from user actions.

This state of affairs constitutes a threat for ontology-based application development by incurring costs of sorting out the available material as well as lack of support for maintaining ontologies over their full life-cycle.

[1] In this figure, O denotes ontologies, A denotes mappings (alignments).

1.2.2 Objective and Definition of the NeOn networked ontology model

The purpose of the NeOn networked ontology model is not to be a completely new language for expressing ontologies on the web. Its purpose it to define an integrated model of an ontology language covering the relevant existing ontology languages and extending them with primitives to express the relationships in a network of ontologies. For that purpose, it will introduce – ontop of existing languages (e.g. OWL Web Ontology Language [Mv03], F-Logic) – a model of the interrelations between ontologies. Because the goal of NeOn is to non-exclusively account for existing formalisms, it will attempt to reuse these legacy formalism as much as possible and refrain from defining new languages when this is not necessary.

Subsequently, if we talk about networked ontologies and networks of ontologies, we refer to the following definition:

Definition 1 *A* Network of Ontologies *is a collection of ontologies related together via a variety of different relationships such as mapping, modularization, version, and dependency relationships. We call the elements of this collection* Networked Ontologies*.*

1.2.3 A metamodel of networked ontologies

In this deliverable, we describe the networked ontology model in terms of a metamodel. Metamodels are used for the specification of modeling languages in a standardized, platform independent manner. Furthermore, a grounding essentially specifies the bi-directional translation of the terms of the metamodel to those of the specific formalism.

The general idea of using a metamodel-based approach and UML profiles for this purpose is depicted in Figure 1.3: The metamodel for networked ontologies as well in MOF, i.e. it is defined in terms of the MOF meta-metamodel (explained in Detail in Chapter 3). The term metamodel is chosen, as a metamodel refers

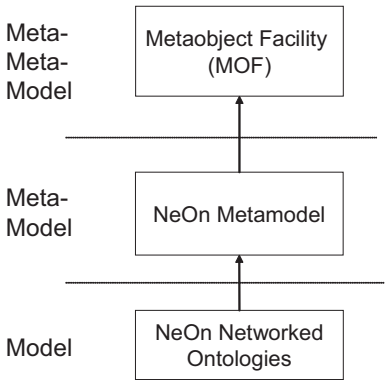


Figure 1.3: The NeOn networked ontology model as a MOF-metamodel

to model of a language, whereas the instances of the metamodel are referred to as models. In our case, models thus refer to the actual ontologies.

Semantics of the metamodel It should be noted that a metamodel is targeted mainly at a *structural specification* of a language, i.e. it is not intended to fully define the (logical) semantics of the language. The semantics is given by the bi-directional translation the metamodel to the underlying logical language. For example, the semantics of the OWL metamodel is given by the model theory of the underlying Description Logic. As such, the metamodel also does not aim at bridging the *semantic differences* between different ontology languages (OWL and F-Logic in particular). Yet, while the semantics of the languages is not directly known to a metamodeling-based system, the common representation within the same metamodeling framework allows translations between different languages based on so-called metamodel transformations. Here,

2006–2007 © Copyright lies with the respective authors and their institutions.

translations are made between logical patterns of the languages, as for example shown in [Bro07] for the translation from F-Logic to OWL.

1.2.4 Networked Ontology Metadata

Additionally, we provide a vocabulary to describe ontology metadata in a network of ontologies. Metadata here refers to data *about* the actual ontologies, as opposed to the content contained *in* an ontology, e.g. the provenance and authorship of an ontology, but also dependencies from other ontologies, etc. Metadata plays an important role in describing and managing the networked ontologies. Its purpose is to be able to clarify the relations between the available ontologies so that they are easy to search, to characterize and to maintain. It aims at making explicit the virtual and implicit network of ontologies.

A NeOn enabled tool will be able to take advantage of the NeOn networked ontology metadata for selecting the resources appropriate for users' needs. It will also enhance maintainability by updating the resources used by the applications when these resources evolve and tracking the conflicts that can occur in such a case. Moreover it should be able to export metadata descriptions according to the specification of the networked ontology model so that other tools can take advantage of them.

As presented in Figure 1.4, the metadata about networked ontologies can reside anywhere with regard to the ontologies: it can be embedded within actual ontology files, expressed in separate files along with the ontologies, expressed on completely different servers (like independent annotations) or reside on the local machine with the ontologies.

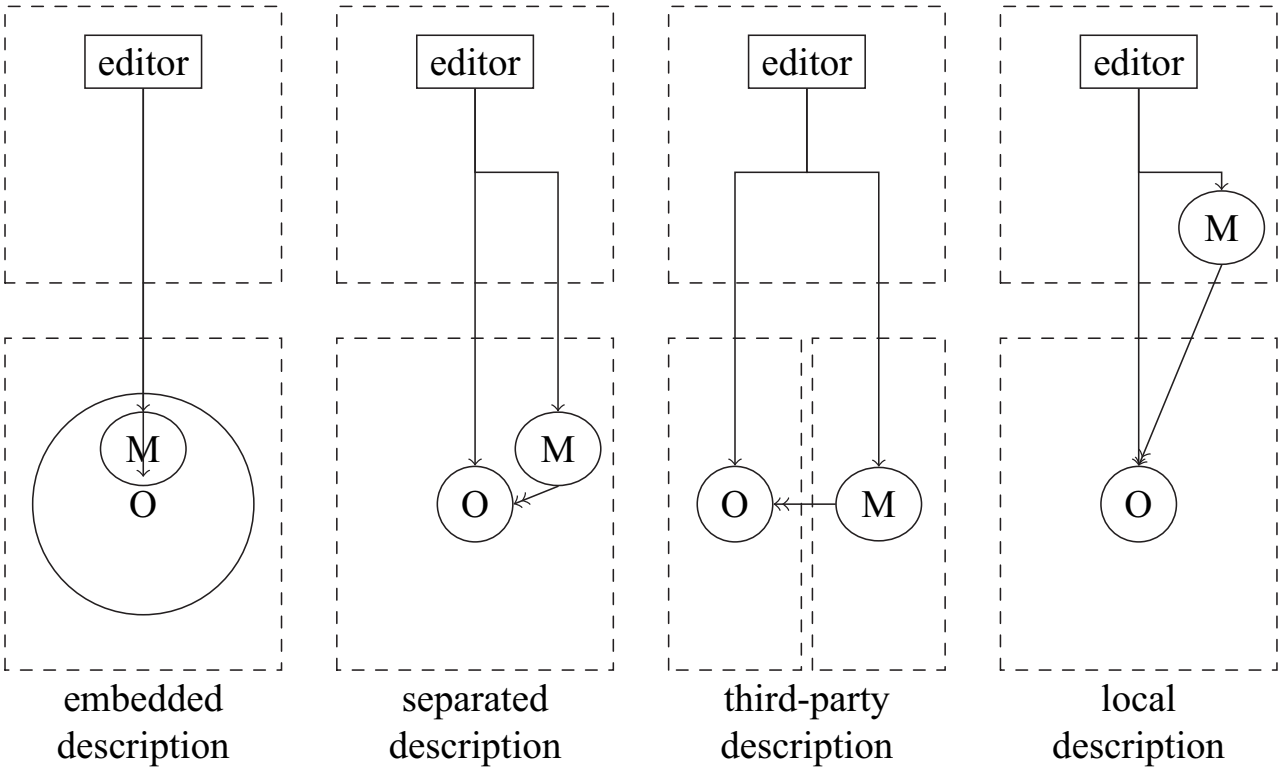


Figure 1.4: The localization of the NeOn networked ontology metadata (M) with regard to the actual ontologies (O). Rectangular boxes refer to machines

1.2.5 Benefits

This NeOn networked ontology description will benefit all stakeholders and in particular, NeOn members:

- it will benefit the Semantic Web as a whole by providing more capabilities for processing ontologies and thereby make ontologies more understandable to machines;
- tool providers: by supporting this format considerably contribute ease the work of application development and maintenance;
- application users will have the guarantee of a better coherence and maintainability of their applications.

1.3 Overview of the Deliverable

In Chapter 3, we introduce the concept of metamodeling and its role for the definition of the networked ontology model. In the following chapters we introduce the specific modules of the metamodel: We start with the metamodel of OWL 1.1 in Chapter 4. We then present the metamodel for F-Logic and SWRL in Chapter 5, for mappings in Chapter 6, and for modularization in Chapter 7.

We then present OMV as a metadata ontology to capture information about networked ontologies in Chapter 9. Finally, in Chapter 10 we provide a summary and an outlook to subsequent deliverables.

2006–2007 © Copyright lies with the respective authors and their institutions.

Chapter 2

Ontology Languages

An ontology is a conceptual schema of a domain, representing the domain's data structure containing all the relevant entities and their relationships within that domain. Possibly, rules are defined on top of the ontology. Additionally, with the increasing use of ontologies, a new issue came up of aligning ontologies that describe the same domain.

OWL is the ontology language standardized by the World Wide Web Consortium and is very well adopted. F-Logic is another ontology language, which is well-known by quite some companies but, however, it was not intentionally built for ontologies but for deductive and object oriented databases. Next to the standard language OWL on which we focus our work, we incorporate F-Logic into our work.

Although OWL is very expressive, it is restricted to obtain decidability. This comes at the price that it can not express arbitrary axioms: the only axioms it can express are of a certain tree-structure. In contrast, decidable rule-based formalism such as function-free Horn rules are not so restricted in the use of axioms but lack some of the expressive power of OWL: they are restricted to universal quantification and lack negation in their basic form. To overcome the limitations of both approaches, several rule extensions for OWL have been heavily discussed [W3C05a] and the W3C initiated a working group for the definition of a Rule Interchange Format [W3C05b]. SWRL is one of the most prominent proposals for an extension of OWL with rules. DL-safe rules [MSS04] are a decidable subset of SWRL. As every DL-safe rule is also a SWRL rule, DL-safe rules are covered in our work. It should be noted that, although we consider SWRL as the most relevant rule extension for OWL in the context of our work, it is not the only rule language which has been proposed for ontologies. Other prominent alternatives for rule languages are mentioned in the W3C RIF charter, namely the Web Rule Language WRL [ABdB$^+$05] and the rules fragment of the Semantic Web Service Language SWSL [GKM05]. These languages differ in their semantics and consequently also in the way in which they model implicit knowledge for expressive reasoning support. From this perspective, it could be desirable to provide particular support tailored to these specific rule languages. From the perspective of conceptual modeling, however, different rule languages appear to be very similar to each other. This opens up the possibility to reuse our outcome for SWRL by augmenting it with some features and language primitives which are not present in SWRL.

Often when ontologies are written in the same language, still they are modeled differently and can not be aligned automatically. Mappings between the heterogenous ontologies have to be defined by the user. For such mappings, we concentrate on OWL ontology mappings since OWL is the most prominent ontology language currently available. Our work covers all existing OWL ontology mapping formalisms we are currently aware of.

This chapter introduces the ontology languages that we support in our work. Firstly, Section 2.1 describes OWL, after which Section 2.2 introduces SWRL. Subsequently, the language F-Logic is discussed in Section 2.3. Finally, OWL ontology mapping languages are addressed in Section 2.4.

2.1 Web Ontology Language (OWL)

The Web Ontology Language OWL [Mv03, PSHM07], based on the earlier language DAML+OIL [vHPSH01], was standardized by the W3C in February, 2004. Since, it has become the most-accepted ontology language in the semantic web community and is supported by a constantly increasing number and range of applications. At the moment of writing this deliverable, a new version of OWL, OWL 1.1 [PSHM07], based on experiences during the language's first years, is undergoing the standardization process. OWL allows to build, publish and share ontologies. By describing the concepts of a domain as well as their relationships formally, the meaning of documents becomes machine-understandable. The formal semantics of OWL is derived from Description Logics (DL, [BCM$^+$03]), an extensively researched knowledge representation formalism. Hence, most primitives offered by OWL can also be found in a Description Logic.

OWL is built on top of RDF [KC04] and extends the expressiveness of RDF and RDF Schema [BG04] by supporting additional vocabulary with a formal semantics.

The first version of OWL defined three sublanguages: OWL Lite, OWL DL and OWL Full. For these different language levels,

OWL Lite $<$ OWL DL $<$ OWL Full

holds in terms of expressivity.

OWL Full OWL Full allows expressions of higher order predicate logic, which results in the fact that OWL Full ontologies can be undecidable and hence OWL Full is impractical for applications that require complete reasoning procedures. Contrary to OWL Lite and OWL DL which impose restrictions on the use of the vocabulary, OWL Full supports the full syntactic freedom of RDF and can be seen as an extension of RDF. It has mainly been defined for compatibility with existing standards such as RDF.

OWL DL is equivalent to a decidable subset of first-order predicate logic. It closely corresponds to the $\mathcal{SHOIN}(D)$ description logic and all language features can be reduced[1] to the primitives of the $\mathcal{SHOIN}(D)$ logic. To allow the representation in this logic, OWL DL imposes several limitations on the set of supported language constructs. The limitations defined on OWL DL ease the development of tools and allow complete inference. For OWL DL, practical reasoning algorithms are known, and increasingly more tools support this or slightly less expressive languages.

OWL Lite is the smallest standardized subset of OWL Full, closely corresponding to the description logic known as $\mathcal{SHIF}(D)$. It is mainly to support the design of classification hierarchies and simple constraints. Several language elements that are supported by OWL DL, are not available in OWL Lite, specifically number restrictions are limited to arities 0 and 1. Furthermore, the oneOf class constructor is missing. Other constructors such as class complement, which are syntactically disallowed in OWL Lite, can nevertheless be represented via the combination of syntactically allowed constructors [Vol04].

It holds that every legal OWL Lite ontology is a valid OWL DL ontology, and every valid OWL DL ontology is a valid OWL Full ontology. For our work, OWL DL would be most useful as it is the OWL dialect which is mostly used in practice as it is decidable although still very expressive. The new version OWL 1.1 on which our work is built, is an extension of OWL DL. OWL 1.1 provides additional Description Logic expressive power, moving from the $\mathcal{SHOIN}(D)$ Description Logic that underlies OWL DL to the $\mathcal{SROIQ}$ Description Logic [HKS06]. [2] [3]

[1] Some language primitives are shortcuts for combinations of primitives in the logic.
[2] The work in this deliverable is based on the OWL 1.1 version available at the end of February, 2007.
[3] Throughout the rest of this deliverable, we are talking about OWL 1.1 whenever we refer to OWL.

2006–2007 © Copyright lies with the respective authors and their institutions.

OWL provides an elaborated set of constructs to define classes (concepts), properties and individuals (instances of one or more classes). We introduce the OWL 1.1 constructs while presenting our metamodel for OWL in Chapter 4.

The OWL 1.1 specifications provide the so-called functional-style syntax as a human-readable syntax. A syntax based on XML allows easy implementations of OWL 1.1. This XML exchange syntax is defined in the XML schema language [TBMM04]. For a full account of the available syntaxes, we refer the reader to [GMPS07]. Additionally, [GM07] provides a mapping from the OWL 1.1 functional-style syntax to the RDF exchange syntax [Bec04].

A formal semantics of OWL 1.1 is provided based on the principles for defining the semantics of description logics. The semantics is defined as a model-theoretic semantics [TV56] by interpreting the constructs of the functional-style syntax. The model-theoretic approach is an accepted paradigm for providing a formal account of meaning and entailment. To interpret OWL 1.1 ontologies serialized in the RDF syntax, they are translated into the functional-style syntax first. For a full account on the model-theoretic semantics of OWL, we refer to [GM06].

2.2 Semantic Web Rule Language (SWRL)

One of the most prominent proposals for an extension of OWL with rules is the Semantic Web Rule Language (SWRL, [HPSB$^+$04]). SWRL proposes to allow the use of Horn-like rules together with OWL axioms. For instance, asserting that the combination of the *hasParent* and *hasBrother* properties implies the *hasUncle* property, is not possible in OWL but in SWRL.

The SWRL specifications, submitted to W3C in May 2004, include a high-level abstract syntax for Horn-like rules extending the OWL abstract syntax. An extension of the model-theoretic semantics from OWL provides the formal meaning for rules written in this abstract syntax. Moreover, next to the abstract syntax, SWRL allows an XML syntax based on the OWL XML presentation syntax, as well as an RDF concrete syntax based on the OWL RDF/XMI exchange syntax. Both the different syntaxes and the model-theoretic semantics of SWRL are described in detail in [HPSB$^+$04].

SWRL has a high expressive power but at the price of decidability. Moreover, it becomes undecidable as rules can be used to simulate role value maps ([SS89]). To balance the expressive power against the execution speed and termination of the computation, suitable subsets of the language can allow efficient implementations. The original SWRL specifications are built on OWL 1.0. The minor changes to adapt to OWL 1.1 are incorporated in our work.

We present the SWRL language constructs while explaining the metamodel in Chapter 5.1.

2.3 Frame Logic (F-Logic)

Frame Logic (F-Logic), for which we refer to [KLW95] for a full account, is a deductive, object-oriented and frame-based formalism. Originally, the language was developed for deductive and object-oriented databases in 1995. Later on, however, it has been applied for the implementation of ontologies. F-Logic provides constructs for defining declarative rules which infer new information from the available information. Furthermore, queries can be asked that directly use parts of the ontology. From a syntactic point of view, F-Logic is a superset of first-order logic. Currently a new version of FLogic is defined as a community process taking current developments from the semantic web area into account. Since these new developments are work in progress we focus on the current implementations of FLogic as addressed in [Ont06, FHK97].

In contrast to OWL and SWRL, F-Logic does not have several syntactical representations defined. Again, we introduce the grammar and the object-model of F-Logic while presenting its metamodel in Chapter 5.2.

The F-Logic semantics is based on the fixpoint semantics of Datalog programs [Ull88]. The evaluation starts with an empty object base, and rules and facts are evaluated iteratively (note that queries are not part of an

F-Logic program). When the rule body is valid in the actual object base with certain variable bindings, these bindings are propagated into the rule head. In this way, new information from rule heads is deduced due to closure properties, and inserted into the object base. Until no new information can be obtained anymore, rules are continuously evaluated. However, this is only the abstract view on the semantics. An F-Logic progam is not necessarily executed in a bottom up fashion.
As with Datalog, the evaluation of a negation-free F-Logic program reaches a fixpoint which coincides with the unique minimal model of that program, which is defined as the smallest set of P- and F-atoms such that all closure properties and all facts and rules of the program are satisfied. As soon as a fixpoint is reached, the semantic of an F-Logic program is computed.

Additionally, F-Logic has a model-theoretic semantics. We refer to [KLW95] for a complete presentation of the F-Logic semantics.

2.4 OWL Ontology Mapping Languages

Since we believe OWL is the most important currently available ontology language, and since we chose OWL as the core of our work, we apply the logical restriction to focus on mappings between ontologies represented in OWL. In contrast to the area of ontology languages, where a de facto standard exists for representing and using ontologies, there is no agreement yet on the nature and the right formalism for defining mappings between ontologies.

OWL mapping formalisms are often based on non-standard extensions of the logics used to encode the ontologies. We focus on approaches that connect description logic based ontologies where mappings are specified in terms of logical axioms. This allows us to be more precise with respect to the nature and properties of mappings. At the same time, we cover all relevant mapping approaches [SSW05a] that have been developed that satisfy these requirements: C-OWL [BGvH$^+$03], OIS (Ontology Integration System, [CDL01]), DL for II (DL for Information Integration, [CDL02]), and DL-Safe Mappings [HM05].

[SU05] researched the nature of ontology mappings and identified some general aspects of the above mapping approaches. We want to take these general aspects as a basis for our work, since we want to make sure that all these approaches are covered. Additionally, we provide specific support for C-OWL and DL-Safe Mappings by defining sets of metamodel constraints as well as language mappings for these concrete formalisms. For specific details on C-OWL and DL-Safe Mappings, we refer to Chapter 6, where we present their aspects along with the introduction of the metamodel.

2006–2007 © Copyright lies with the respective authors and their institutions.

Part II

The NeOn Metamodel for Networked Ontologies

Chapter 3

Metamodeling for Ontologies

3.1 Metamodeling with MOF

The Model Driven Architecture [MKW04, Fra03] is an initiative of the standardization organization OMG (Object Management Group) to support model-driven engineering of software systems based on the idea that modeling is a better foundation for developing and maintaining systems. MDA utilizes modeling as technique to raise the abstraction level and to effectively manage the complexity of systems. Models increase productivity, under the assumption that it is cheaper to manipulate the model, which is an abstraction of the system, than the system itself.

Using the MDA methodology, system functionality may first be defined as a technology neutral model through an appropriate modeling language. Then, using automated tools, this can be translated to one or more technology specific models. One of the fundamental components of MDA that is used in our work, is the Meta Object Facility.

When describing a universe of discourse in the real world, objects are being *classified* according to their common features. Furthermore, when referring to the objects, some of their properties which are not relevant may be left out. This step of describing objects with only those original properties that are of interest, and leaving out the irrelevant properties, is called *abstraction*. In addition, next to classification and abstraction, the classes of objects are grouped based on common features, a process called *generalization*.

These three steps of classifying, abstracting and generalizing take place in many contexts, but they are crucial in the modeling area. When formalizing the description of some domain in a model, these three steps are typically combined, and therefore performed at the same time. That is to say, a modeler looks at the real objects to be modeled, and abstracts away any information which seems not to be required. Simultaneously, the objects are classified into types and these types are put into generalization-hierarchies.

To support the MDA initiative, it is essential that designed models are commonly understood by all involved parties. This requires the ability to specify the meaning of a model. This is where *metamodels* come into play. A metamodel specifies the constructs of a modeling language, using the constructs that can be used in the description of a language. The Meta Object Facility [Obj06, Fra03] provides such a set of metamodeling constructs to define MOF-based metamodels as models of modeling languages.

We first introduce MOF using its four-layer architecture in Section 3.1.1. Next, Section 3.1.2 starting on page 28 introduces the available MOF constructs.

3.1.1 The Four-Layer Architecture of MOF

Before explaining some more details of MOF, let us look at the relationship between a metamodel and an instance model that represents the objects in the real world. A standardized terminology of OMG eases the communication about the different layers we just introduced, namely the *information layer*, the *model layer* and the *metamodel layer*. These are shown in the three bottom layers of Figure 3.1.

2006–2007 © Copyright lies with the respective authors and their institutions.

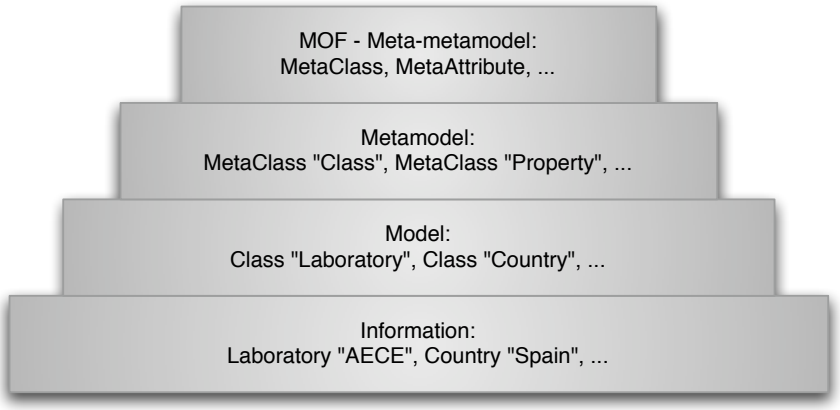


Figure 3.1: OMG's four-layer metamodel hierarchy

The undermost layer of this hierarchy encompasses the raw information to be described. For example, the figure contains information about a pharmaceutical laboratory called AECE and about the country Spain, in which the laboratory is located. One layer above, the model layer contains the definition of the required structures, like the classes used for classifying information. Thus in the example, the classes Laboratory and Country are defined. If these structures are combined, they describe the model for the given domain. The metamodel on the third layer defines the terms in which the model is expressed. In our example, we would state that models are expressed with classes and properties by instantiating the respective metaclasses. Important to mention is that the architecture allows models to span more than one of the lower levels. Ontologies do not have a clear separation between model and information. Therefore, in our work, the two bottom layers of Figure 3.1 are combined into one layer.

Finally, the figure also presents the fourth layer, the meta-metamodel layer called the MOF layer. It is the meta-metadata defining the modeling constructs that can be used to define and manipulate a set of interoperable metamodels on the level below. The MOF layer contains only the simplest set of concepts for models and metamodels, and captures the structure and semantics of arbitrary metamodels. Note that the top MOF layer is "hard wired" in the sense that it is fixed, while the other layers are flexible and allow to express various metamodels such as the existing UML metamodel or our metamodel for ontologies, rules and ontology mappings.

The model-driven framework supports any kind of metadata and allows new kinds to be added as required. Clearly, the MOF layer plays a crucial role in this four-layer metamodel hierarchy of OMG. It allows to define, manipulate and integrate metamodels and models in a platform-independent manner.

3.1.2 Available Constructs in MOF

The set of constructs provided by MOF is a simple set of concepts, though powerful enough for capturing the static structure of a model. The five MOF-concepts are represented as metaconcepts in Figure 3.1 and can define the abstract syntax of a language:

- types (classes, primitive types, and enumerations),
- generalization,
- attributes,
- associations, and
- operations.

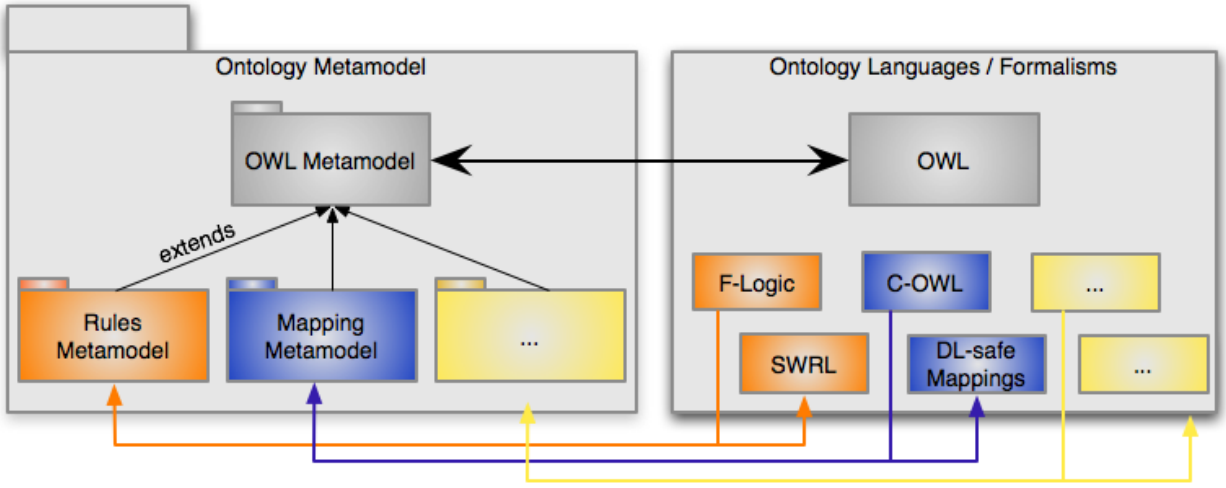


Figure 3.2: Modules of the Networked Ontology Metamodel and possible groundings in ontology languages

Ecore [BGS+03] has emerged as a quasi standard for the definition of metamodels and we provide our metamodels in the Ecore-format. Also, throughout the years, OMG has striven for a unification of MOF and UML ([Gro05], [Obj06]). This makes UML also suitable as a graphical notation for MOF metamodels. Moreover, when UML is used to build MOF metamodels, such specifications are not merely UML diagrams. Instead, MOF borrows object-oriented class-modeling constructs from UML and presents them for describing the abstract syntax of modeling constructs. Thus, MOF metamodels can look like UML class models, and one can use UML class-modeling tools to display them. We present our metamodels also in UML so that they are easily understandable for the reader.

3.1.3 MOF vs. UML

It should be noted that a MOF specification is not merely a UML diagram. Instead, MOF borrows OO class-modeling constructs from UML and presents them as the common means for describing the abstract syntax of modeling constructs. Thus, MOF metamodels look like UML class models, and one can use UML class-modeling tools to create them.

3.2 A Networked Ontology Metamodel

The ontology metamodel as well as a UML profile are grounded in MOF in that they are defined in terms of the MOF meta-metamodel. The UML profile mechanism is an extension mechanism to tailor UML to specific application areas. UML profiles define a visual notation for optimally supporting the specification of networked ontology models. This visual syntax is based on the metamodel. Mappings in both directions between the metamodel and the profile have to be established.

However, the OWL ontology metamodel is just one part of the networked ontology metamodel. The metamodel consists of several modules. The core module, i.e. the OWL metamodel, is extended by different modules that provide additional features, e.g. rules or mappings. In many application scenarios, only particular aspects of the networked ontology model are needed. In these cases, only the relevant modules need to be supported and used.

While the OWL ontology metamodel has a direct grounding in the OWL ontology language, the extensions have a generic character in that they are formalism independent and allow a grounding in different formalisms.

2006–2007 © Copyright lies with the respective authors and their institutions.

3.2.1 Design Considerations

Modularization The metamodel consists of several modules. The core module, i.e. the OWL metamodel, is extended by different modules that provide additional features, e.g. modularization, mappings, etc. In many application scenarios, only particular aspects of the networked ontology model are needed. In these cases, only the relevant modules need to be supported and used.

Compatibility with standards In terms of the metamodel, two aspects of standards are relevant: The first aspect relates to the fact that we are using a standard formalism—namely the Meta Object Facility—to describe the metamodel. The second aspect relates to the metamodel of networked ontologies itself: A major design goal is compatibility with existing ontology languages. With the Web Ontology Language OWL we have a standard for representing ontologies, therefore we provide a metamodel of OWL directly, with a one-to-one translation. For the other aspects of networked ontologies (mappings, rules, ...) no such standards exist yet. In favor of general applicability we therefore provide generic metamodels for these extensions that allow translations to different formalisms, as shown in Figure 3.2.

3.2.2 Advantages of the Networked Ontology Metamodel

Defining the ontology model in terms of a MOF compliant metamodel yields a number of advantages:

1. Interoperability with software engineering approaches In order for semantic technologies to be widely adopted by users and to succeed in real-life applications, they must be well integrated with mainstream software trends. This includes in particular interoperability with existing software tools and applications to put them closer to ordinary developers. MDA is a solid basis for establishing such interoperability. With the ontology model defined in MOF, we can utilize MDA's support in modeling tools, model management and interoperability with other MOF-defined metamodels.

2. Standardized model transformations MOF specifications are independent of particular target platforms (e.g. programming languages, concrete exchange syntaxes, etc.). Industry standardized mappings of the MOF to specific target languages or formats can be used by MOF-based generators to automatically transform the metamodel's abstract syntax into concrete representations based on XML Schema [TBMM04], Java, etc. For example, using the MOF-Java mapping, it is possible to automatically generate a Java API for a MOF metamodel.

3. Custom model transformations As MOF-based metamodels are all built using the same standardized constructs, mappings between them can be defined in a straightforward way. Based on these mappings between the metamodels, automatic transformations between models of the concerning languages can be performed.

4. Reuse of UML for modeling With respect to interoperability with other metamodels, UML is of particular importance. UML is a well established formalism for visual modeling and has been proposed as a visual notation for knowledge representation languages as well [HEC+04, BKK+01, BKK+02, CP99, Kre98]. While UML itself lacks specific features of knowledge representation languages, the extension mechanisms – UML profiles – allow to tailor the visual notation to specific needs.

5. Independence from particularities of specific formalisms The metamodeling approach of MDA and MOF allows to define a model in an abstract form independent from the particularities of specific logical formalisms. This enables to be compatible with currently competing formalisms (e.g. in the case of rule- or mapping languages), for which no standard exists yet. Language mappings define the relationship with

particular formalisms and provide the semantics for the ontology model. Furthermore, the extensibility capabilities of MOF allow to add new modules to the metamodel if required in the future.

3.2.3 Semantics of the Metamodel

The MOF does not completely define the semantics of the language specified by a metamodel. The semantic features of the MOF are limited to describing constraints on the structure of models, using the Object Constraint Language OCL. Obviously, this is not sufficient . and not intended . to capture the complete semantics of arbitrary languages. The actual semantics of the language specified by the metamodel is defined by so called language mappings, also called groundings. They define the relationship with particular (logical) formalisms and provide the semantics for the metamodel. For the networked ontology model this means the following:

- Our metamodel for OWL DL ontologies has a one-to-one mapping to the abstract syntax of OWL DL and thereby to its formal semantics.
- The rule metamodel currently has a translation to SWRL and its semantics. However, SWRL is not generally accepted as a standard for representing rules. Other proposals for rule languages exist, which differ in their semantics. To support such languages (such as F-Logic) in the future, it would either be possible to define new language mappings from our metamodel to these languages, or to define different rule metamodels, each of which is tailored to a specific rules language.
- For the mapping metamodel, we deliberately abstained from subscribing to a particular mapping language, but instead constructed the metamodel to be able to cover all relevant approaches to mapping languages. We believe that many of them are useful in different contexts and that a formalism-independent metamodel is appropriate. The work on groundings in specific mapping languages is work in progress and will be reported in subsequent deliverables.
- Also for the modularization metamodel, we did not subscribe to a specific language for modular ontologies. Different language mappings would allow to support various languages for modular ontologies. As part of a subsequent deliverables (D1.1.3) we will define the appropriate semantics for modular ontologies in NeOn.

2006–2007 © Copyright lies with the respective authors and their institutions.

Chapter 4

The OWL Metamodel

This chapter introduces the core of our networked ontology model, which is a MOF-based metamodel for OWL 1.1 ontologies.
We provide the metamodel for OWL 1.1 using the core modeling features provided by MOF. To state the metamodel more precisely, we augment it with OCL constraints, which specify invariants that have to be fulfilled by all models that instantiate the metamodel.

A metamodel for an ontology language can be derived from the modeling primitives offered by the language. Our metamodel for OWL ontologies has a one-to-one mapping to the functional-style syntax of OWL 1.1 and thereby to its formal semantics.[1]

Along with the explanation of the various OWL constructs[2], we introduce and discuss the corresponding metaclasses, their properties and their constraints. To simplify the understanding of the metamodel, we add accompanying UML diagrams to our discussion.
This chapter presents the MOF-based metamodel for OWL 1.1 in eight sections: Section 4.1 starts with ontologies and annotations, after which Section 4.2 presents entities and data ranges. Next, Section 4.3 demonstrates class descriptions and Section 4.4 presents OWL axioms. Then, Section 4.5 gives class axioms, after which Section 4.6 presents object property axioms. Finally, Section 4.7 presents data property axioms and Section 4.8 presents facts.

4.1 Ontologies and Annotations

An OWL ontology is defined by a set of axioms, of which OWL 1.1 provides six different types. Additionally, an ontology has an ontology URI which defines it uniquely, a (possibly empty) set of imported ontologies, and a set of annotations (see Example 1). With the import primitive, another ontology and so its axioms can be imported into an ontology. An annotation is some arbitrary information on the ontology, like the creator, version info and so forth, and consists of a URI to define the annotation type, and a constant which defines the annotation's value. Additionally, also the ontology's elements, the axioms, can be annotated.

Example 1 *The following example illustrates the definition of an ontology and involves the specification of an imported ontology and an annotation:*
Ontology(PharmaceuticalDomain Import(Medication) Comment("An ontology about the pharmaceutical domain."))

[1] Where the language specifications define constraints on character strings like for instance URIs, this is not included as OCL constraints in the metamodel. Although it might be somehow possible to enforce such restrictions by defining OCL constraints, it would be cumbersome and thus it is preferable that tools that implement the metamodel support these constraints.
[2] Remember, however, that the language itself is not part of our contribution.

Figure 4.1 shows the metamodel presentation for ontologies, its axioms and annotations as metaclasses. The association *ontologyAxiom* connects the ontology to its axioms, whereas the associations *ontologyAnnotation* and *axiomAnnotation* connect ontologies respectively axioms to their annotations. The class *Ontology* has an attribute to identify the ontology, *URI*, whereas the attribute *URI* of the class *Annotation* specifies the type of annotation. Although it would be possible and also correct to define a metaclass *URI* to represent all URIs, a URI is only a simple value and thus it is more suited to represent it as an attribute of the concerning metaclasses instead of as a first-class object in the form of a metaclass. Finally, an association *importedOntology* from the class *Ontology* to itself links an ontology to its imported ontologies.

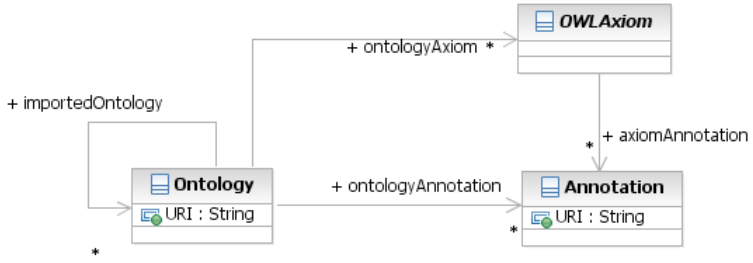


Figure 4.1: OWL metamodel: ontologies

Annotations can be self-defined or can be one of two specificly defined annotation types: label and comment. The URI that defines the type of an annotation is not applicable to a label or comment as their type is already defined through the construct itself. The OWL 1.1 specifications specify the value of an annotation as a constant, which is composed of a string and a datatype URI in case of a typed constant, or a string and a language tag in case of an untyped constant.

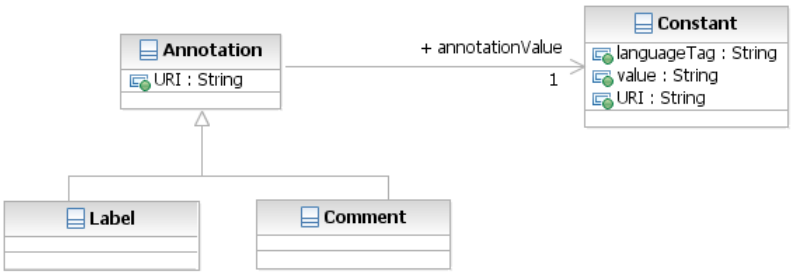


Figure 4.2: OWL metamodel: annotations

Figure 4.2 demonstrates how the metamodel specifies the two specific types of annotations as subclasses of the metaclass *Annotation*. The type of a self-defined annotation is defined through its *URI*, whereas the type of a label or comment is defined through the instantiation of the metaclass. OCL constraints define the restriction that the attribute *URI* is empty for *Label* and *Comment* but mandatory for all other annotations:

1. When an annotation is not a label or a comment, a URI must be defined:
   context Annotation inv:
   not(self.oclIsTypeOf(Label) or self.oclIsTypeOf(Comment))
   implies self.URI = 1

2. A label does not have a URI:
   context Label inv:
   self.URI = 0

3. A comment does not have a URI:

2006–2007 © Copyright lies with the respective authors and their institutions.

context Comment inv:
self.URI = 0

An association *annotationValue* connects an *Annotation* to its value, so the text that forms the annotated remark, represented by the metaclass *Constant*. *Constant* has three attributes for specifying the different parts a constant is composed of. The attribute *value* is applicable for both typed and untyped constants, whereas *languageTag* is only applicable to untyped constants and *URI* only to typed constants. A constraint defines the applicability of the attributes *languageTag* and *URI* [3]:

1. A constant has either a language tag (untyped constant) or a URI (typed constant), but can not have both:
   context Constant inv:
   (self.languageTag = 1 implies self.URI = 0) and
   (self.URI = 1 implies self.languageTag = 0)

4.2 Entities and Data Ranges

Entities are the fundamental building blocks of OWL 1.1 ontologies. OWL 1.1 has five entity types: data types, OWL classes[4] (see Example 2), individuals, object properties and data properties. A datatype is the simplest type of data range. The second entity, a class, is a simple axiomatic class description classifying a set of instances. These class instances are called individuals and are also classified as OWL entities. At last, an object property connects an individual (belonging to a class) to another individual, whereas a data property connects an individual to a data value (belonging to a data range).

Example 2 *The following example illustrates the definition of a class in OWL:*
OWLClass(Medication)

The OWL specifications highlight entities as the main building blocks of an OWL ontology and its axioms. Hence, the metamodel defines them as first-class objects in the form of metaclasses. Figure 4.3 presents an abstract metaclass *OWLEntity* which is defined as supertype of all types of entities. The five specific types of entities are specified as subtypes of *OWLEntity*: *Datatype*, *OWLClass*[5], *ObjectProperty*, *DataProperty* and *Individual*. An attribute *URI* of the abstract metaclass *OWLEntity* is inherited by all subclasses to identify the entity.

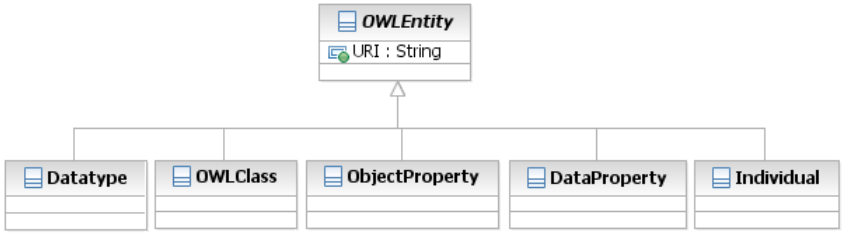


Figure 4.3: OWL metamodel: entities

Just like ontologies and axioms, also entities can be annotated (see Example 3). OWL categorizes such entity annotations as axioms. Hence an entity can be involved in two types of annotation: an annotation of the entity itself, or an annotation of such entity annotation as an axiom.

[3] Note that the metamodel specifies the attribute *value* as mandatory for any constant.
[4] OWL provides two classes with predefined URI and semantics: owl:Thing defines the set of all objects (top concept), whereas owl: Nothing defines the empty set of objects (bottom concept).
[5] Note that a simple class, in contrast to the other entities, has the prefix 'OWL' in conformity with the OWL 1.1 specifications.

Example 3 *The following example illustrates the annotation of an entity: EntityAnnotation(OWLClass(Medication) Annotation(CreationDate "Created in March 2007."))*

Figure 4.4 demonstrates the metamodel representation of entity annotations by the metaclass *EntityAnnotation* which is a subclass of the metaclass *OWLAxiom*, as OWL specifies an entity annotation as an axiom. The association *entityAnnotation* connects *EntityAnnotation* to the annotation. As the number of annotations

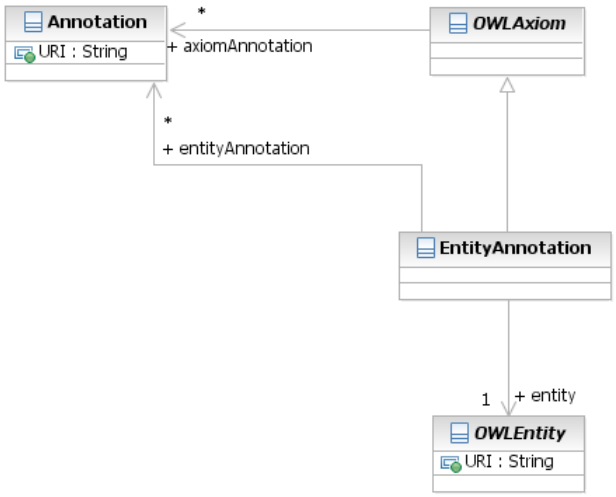


Figure 4.4: OWL metamodel: entity annotations

on entities is unrestricted, the association carries the multiplicity 'zero to many'. The other association from *EntityAnnotation*, *entity*, specifies the entity on which the annotation is defined. Naturally, the multiplicity of this association is 'exactly one'.

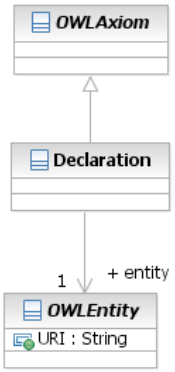


Figure 4.5: OWL metamodel: declarations

Additionally, OWL provides axioms to declare entities (see Example 4). An entity is declared in an ontology if the ontology, or one of its imported ontologies, contains a declaration axiom for the entity. Entity declarations are important for checking the structural consistency of an ontology: if each entity occuring in an axiom of the ontology is declared in the ontology, the ontology is structurally consistent. Figure 4.5 presents a metaclass *Declaration* as a subclass of the metaclass *OWLAxiom*, linked to the entity via the association *entity*.

Example 4 *The following example illustrates how a class definition is declared: Declaration(OWLClass(Medication))*

2006–2007 © Copyright lies with the respective authors and their institutions.

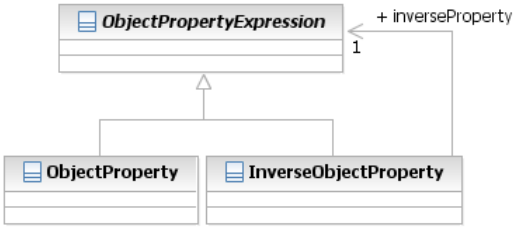


Figure 4.6: OWL metamodel: object property expressions

OWL distinguishes two types of property expressions: object property expressions and data property expressions, represented by the respective metaclasses *ObjectProperty* and *DataProperty*. An object property can possibly be defined as the inverse of an existing object property (see Example 5). The metamodel has an abstract superclass *ObjectPropertyExpression* for both the usual object property and the object property defined as an inverse of another. Figure 4.6 gives its subclasses *ObjectProperty* and *InverseObjectProperty*, representing normal respectively inverse object properties. An inverse object property can be defined

Figure 4.7: OWL metamodel: data property expressions

based on any, normal or inverse, object property, hence the association *inverseProperty* from the metaclass *InverseObjectProperty* is connected to the superclass *ObjectPropertyExpression*.

Example 5 *The following example illustrates the definition of an inverse object property: InverseObjectProperty(MadeFromIngredient)*

Figure 4.7 shows that, although inverse data properties can not be defined, the entity *DataProperty* is also generalized into a supertype in the metamodel, in symmetry with object properties.

Figure 4.8 gives the metamodel representations for the various data range constructs in OWL. To define a range over data values, OWL provides four constructs, which are generalized in the metamodel into an abstract superclass *DataRange*. The basic and simplest data range is the datatype, defined by a URI.

The metaclass *DataComplementOf* defines a data range as the complement of an existing one, connected through the association *dataRange*. The third data range construct, represented in the metamodel by the metaclass *DatatypeRestriction*, applies a facet on an existing data range, connected through the association *dataRange*. The facet is specified through the association *datatypeFacet*, whereas the value of the facet is specified as a constant. A constraint defines the possible types of facets:

1. The facet of a datatype restriction may only have specific values:
   context DatatypeRestriction inv:
   self.datatypeFacet = 'length'
   or self.datatypeFacet = 'minLength'
   or self.datatypeFacet = 'maxLength'

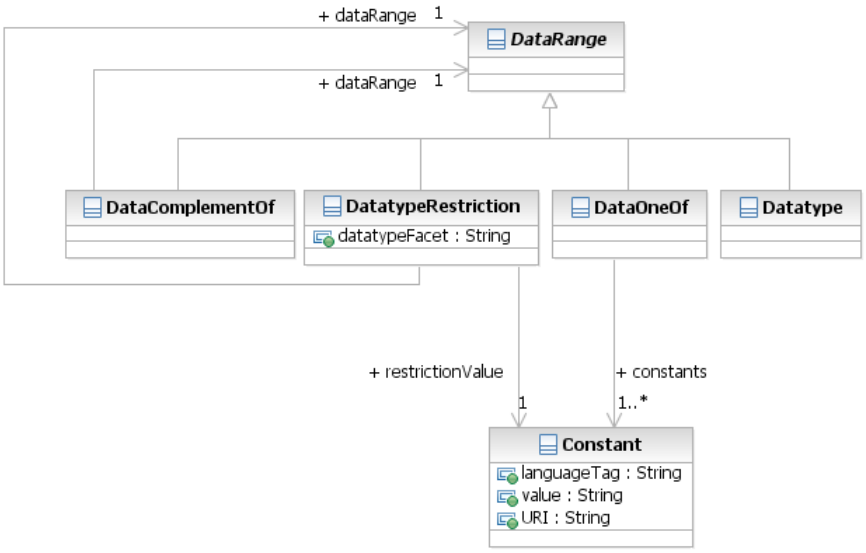


Figure 4.8: OWL metamodel: data ranges

or self.datatypeFacet = 'pattern'
or self.datatypeFacet = 'minInclusive'
or self.datatypeFacet = 'minExclusive'
or self.datatypeFacet = 'maxInclusive'
or self.datatypeFacet = 'maxExclusive'
or self.datatypeFacet = 'totalDigits'
or self.datatypeFacet = 'fractionDigits'

The last data range type, represented by the metaclass *DataOneOf*, defines a data range by enumerating the data values it contains (see Example 6). The enumerated data values are represented as constants.

Example 6 *The following example illustrates the enumeration of data values: DataOneOf("5"^^xsd:integer "15"^^xsd:integer)*

4.3 Class Descriptions

The two remaining groups of constructs are classes and axioms. We first address the classes. Classes group similar resources together and are the basic building blocks of class axioms. The class extension is the set of individuals belonging to the class. To define classes, OWL 1.1 provides next to a simple class definition (metaclass *OWLClass*) several very expressive means for defining classes. The metamodel defines a metaclass *Description* as abstract superclass for all class definition constructs. These constructs can be divided into two main groups: propositional connectives, and restrictions on properties. We present all class descriptions in this section, accompanied by five diagrams.

Firstly, Figure 4.9 presents the class constructs with propositional connectives[6]. Propositional connectives build classes by combining other classes or individuals. The metaclasses *ObjectUnionOf* and *ObjectIntersectionOf* represent constructs defining a class of which each individual belongs to at least one, respectively to all of the specified classes (see Example 7). They both have an association called *classes* specifying the

[6]Note that *OWLClass*, although shown in this figure as a subclass of *Description*, is just a simple class and no propositional connective.

2006–2007 © Copyright lies with the respective authors and their institutions.

involved classes[7]. Moreover, a class can be defined as the complement of another class, represented by the metaclass *ObjectComplementOf* with association *class*, or as an enumeration of a set of (at least one) individuals. The latter one is represented in the metamodel by the metaclass *ObjectOneOf* and its association *individuals*.

Example 7 *The following example illustrates the definition of a class as a subclass of another class, which is defined through a class description as the intersection of three other classes:*
SubClassOf(FluidMedication ObjectIntersectionOf(Fluid Medication BottledProduct))

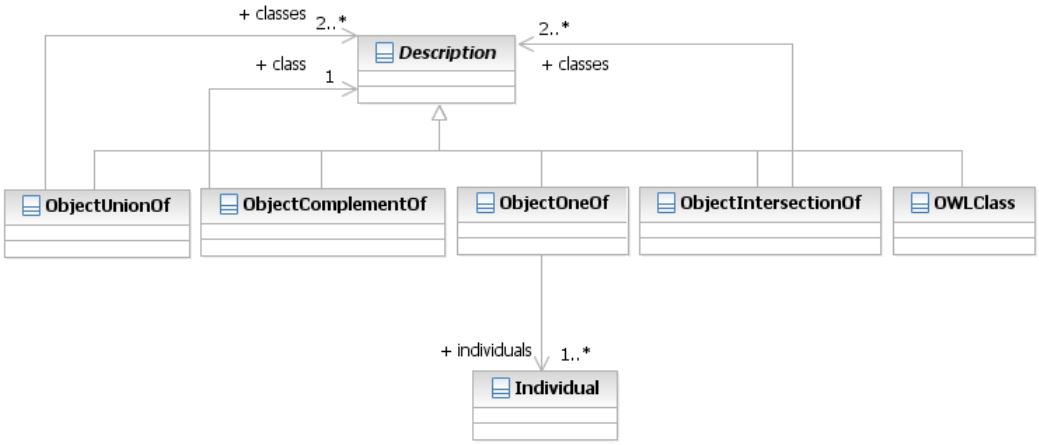


Figure 4.9: OWL metamodel: propositional connectives

All other class definitions in OWL 1.1 describe a class by placing constraints on the class extension. Figure 4.10 gives the next group, which defines restrictions on the property value of object properties for the context of the class. The metaclass *ObjectAllValuesFrom* represents the class construct that defines a class as the

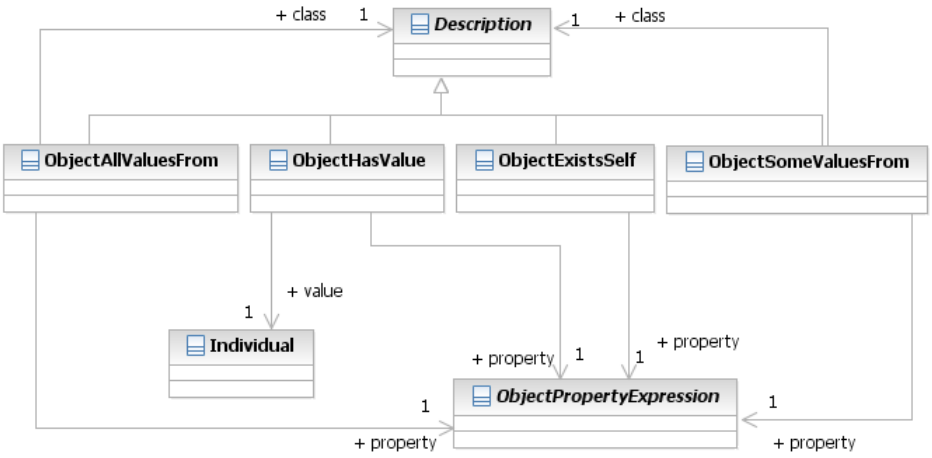


Figure 4.10: OWL metamodel: object property restrictions

[7]Note that, as the involved classes can be defined by any class description construct, they are specified in the metamodel via their abstract superclass.

set of all objects which have only objects from a certain other class description as value for a specific object property (see Example 8).

Example 8 *The following example illustrates the definition of a class as a subclass of another class, which is defined through a class description by restricting an object property:*
SubClassOf(Medication ObjectAllValuesFrom(Treats Disease))

The OWL construct that defines a class as all objects which have at least one object from a certain class description as value for a specific object property, is represented by the metaclass *ObjectSomeValuesFrom*. Both *ObjectAllValuesFrom* and *ObjectSomeValuesFrom* have an association called *class*, specifying the class description of the construct, whereas their association *property* specifies the object property on which the restriction is defined. To define a class as all objects which have a certain individual as value for a specific object property, OWL provides a construct that is represented in the metamodel by the metaclass *ObjectHasValue*, its association *property* specifying the object property, and its association *value* specifying the property value. The metaclass *ObjectExistsSelf* represents the class description of all objects that have themselves as value for a specific object property.

Thirly, Figure 4.11 demonstrates the third group of class definitions, which impose restrictions on the cardinalities of object properties. A cardinality of an object property for the context of a class can be defined

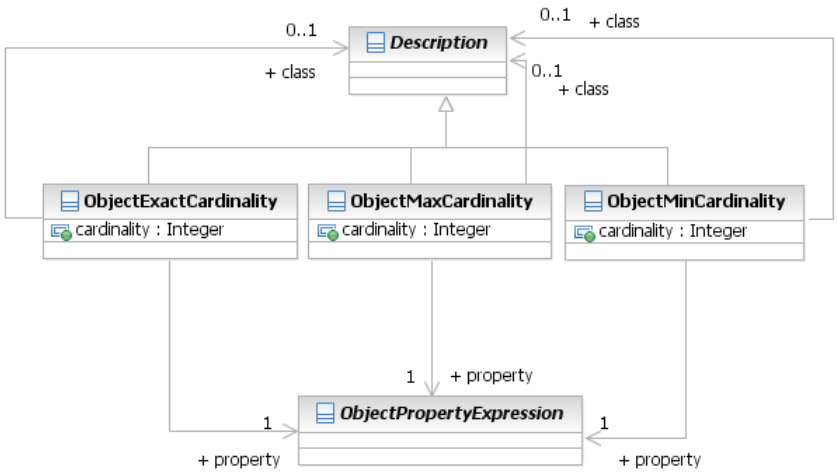


Figure 4.11: OWL metamodel: object property cardinality restrictions

as a minimum, maximum or exact cardinality. The first one of this group is represented by the metaclass *ObjectMinCardinality*, which defines a class of which all individuals have at least N different individuals of a certain class as values for the specified object property (N is the value of the cardinality constraint) (see Example 9).

Example 9 *The following example illustrates the definition of a class as a subclass of another class, which is defined through a class description by restricting the cardinality of an object property:*
SubClassOf(Medication ObjectMinCardinality(1 madeFromIngredient Ingredient))

Secondly, the construct represented by the metaclass *ObjectMaxCardinality* defines a class of which all individuals have at most N different individuals of a certain class as values for the specified object property. Finally, the construct represented by the metaclass *ObjectExactCardinality* defines a class of which all individuals have exactly N different individuals of a certain class as values for the specified object property.

2006–2007 © Copyright lies with the respective authors and their institutions.

To specify the cardinality (N) of these constructs, which is a simple integer, all three metaclasses have an attribute *cardinality*. OCL constraints define that this cardinality must be a nonnegative integer[8]:

1. The cardinality must be nonnegative:
   context ObjectExactCardinality inv:
   self.cardinality >= 0
2. The cardinality must be nonnegative:
   context ObjectMaxCardinality inv:
   self.cardinality >= 0
3. The cardinality must be nonnegative:
   context ObjectMinCardinality inv:
   self.cardinality >= 0

Additionally, they all have an association *class* and an association *property* representing the class respectively the object property involved in the statement. Note that the multiplicity of the associations called *class* have multiplicity 'zero or one' as OWL does not define the explicit specification of the restricting class description as mandatory[9].
Just like with object properties, restrictions can be defined on data properties to define classes. On data properties only three kinds of property value restrictions can be defined (see Example 10).

Example 10 *The following example illustrates the definition of a class as a subclass of another class, which is defined through a class description by specifying a certain value for a data property:*
SubClassOf(MedicationForAdults DataHasValue(hasMinimumAge "12"^^xsd:integer))

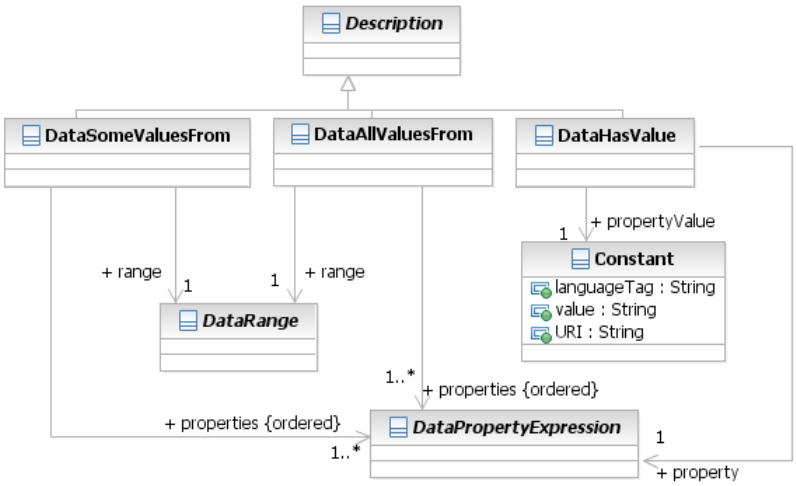


Figure 4.12: OWL metamodel: data property restrictions

Figures 4.12 represents the metaclasses for these constructs, *DataSomeValuesFrom*, *DataAllValuesFrom* and *DataHasValue*. The meaning of the corresponding constructs is the same as with these restrictions on object properties. The value of a data property belongs to a data range, represented by the metaclass

[8]Note that it would make sense to define a constraint which specifies that when a minimum and a maximum cardinality on a property are combined to define a class, the minimum cardinality should be less than or equal to the maximum cardinality. However, as the OWL specifications do not define this restriction and rely on OWL applications to handle this, we also do not define an OCL constraint in the metamodel.
[9]In the case where it is not explicitly defined, called an unqualified cardinality restriction, the description is owl:Thing.

DataRange, and a specific value is represented by the metaclass *Constant*. Moreover, the associations called *properties* that both *DataAllValuesFrom* and *DataSomeValuesFrom* have, are ordered and have multiplicity '1 to many', as OWL allows to specify more than one data property. This supports class definitions like "objects whose width is greater than their height", where the width and height are specified using two data properties.

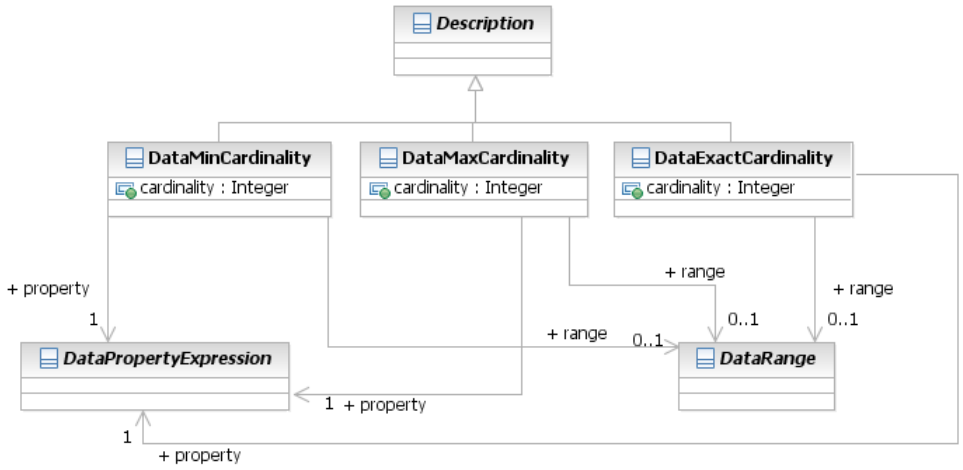


Figure 4.13: OWL metamodel: data property cardinality restrictions

As the last set of class descriptions in OWL, Figure 4.13 gives the cardinality restrictions defined on data properties similar to the cardinality restrictions on object properties. The restrictions specify a minimum, maximum or exact cardinality on a certain data range for the specified data property. Again, OCL constraints restrict the cardinality to nonnegative integers:

1. The cardinality must be nonnegative:
   context DataMinCardinality inv:
   self.cardinality>=0

2. The cardinality must be nonnegative:
   context DataMaxCardinality inv:
   self.cardinality>=0

3. The cardinality must be nonnegative:
   context DataExactCardinality inv:
   self.cardinality>=0

4.4 OWL Axioms

Figure 4.14 demonstrates the six types of OWL axioms. We now introduce the four remaning types of axioms, which are class axioms, object property axioms, data property axioms and facts. They are all defined through an abstract superclass with various subclasses[10].

2006–2007 © Copyright lies with the respective authors and their institutions.

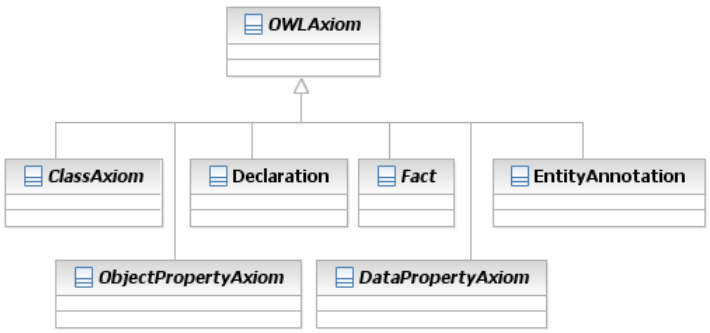


Figure 4.14: OWL metamodel: axioms

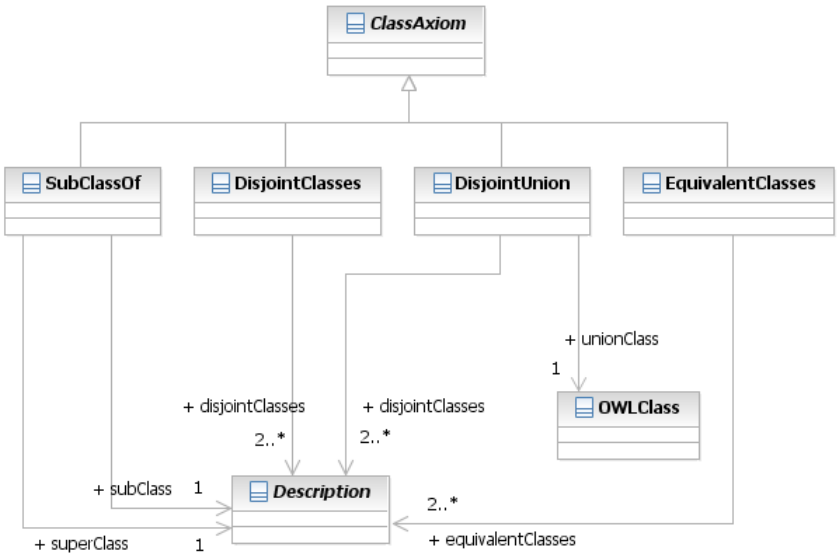


Figure 4.15: OWL metamodel: class axioms

4.5 Class Axioms

Class axioms make statements about the extensions of two or more classes. Figure 4.15 gives the metamodel representations for the four kinds of OWL class axioms. The first class axiom defines that a class is a subclass of another class and is represented in the metamodel as the metaclass *SubClassOf*, connected to the two classes via the associations *subClass* and *superClass*. Two other axioms, represented by the metaclasses *DisjointClasses* and *EquivalentClasses*, define that the extensions of two or more classes are disjoint respectively equivalent (see Example 11). An association *disjointClasses*, respectively *equivalentClasses* connects the metaclasses to the related class descriptions.

Example 11 *The following example illustrates how three classes are defined to be equivalent: EquivalentClasses(Disease Illness Sickness)*

The fourth and last class axiom defines that one class is the union of a set of classes which are all pair-wise disjoint. The metaclass *DisjointUnion* represents this axiom and has an association *unionClass* specifying the class, and an association *disjointClasses* specifying the disjoint classes.

[10]Note that the metamodel models all these statements as metaclasses since OWL specifies them as first-class objects in the form of axioms.

4.6 Object Property Axioms

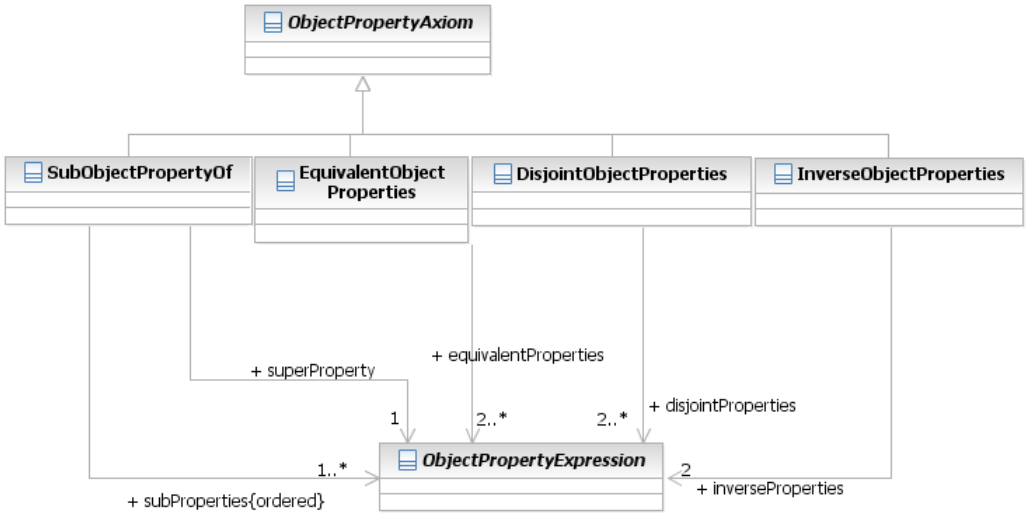


Figure 4.16: OWL metamodel: object property axioms - part 1

The group of object property axioms contains constructs defining relations between different properties, definitions of domain and range and specifications of property characteristics. We first introduce the various relations between object properties in Figure 4.16.

The first one, represented by the metaclass *SubObjectPropertyOf*, defines that the property extension of one object property, specified through the association *subProperties*, is a subset of the extension of another object property, specified through the association *superProperty*. OWL allows to specify a subproperty chain, where the relation defined through the application of the first until the last property of the chain is defined as the subproperty of the specified superproperty. Hence the association *subProperties* is ordered and carries multiplicity 'one to many'. The metaclass *EquivalentObjectProperties* represents a construct that defines that the property extensions of two or more object properties are the same, whereas the class *DisjointObjectProperties* connects two or more object properties that are pair-wise disjoint. Inverse object properties are combined with the construct represented as the metaclass *InverseObjectProperties*, which defines that for every (x,y) in the property extension of one property, there is a pair (y,x) in the other property extension, and vice versa.

To define the class to which the subjects of an object property belong, OWL provides the domain concept (see Example 12), whereas a range specifies the class to which the objects of the property, the property values, belong. Figure 4.17 presents the metamodel elements for the constructs defining an object property domain and range.

Example 12 *The following example illustrates the definition of the domain of an object property: ObjectPropertyDomain(madeFromIngredient Medication)*

The metaclass *ObjectPropertyDomain* specifies an object property and its domain via the associations *property* respectively *domain*. Similarly, the metaclass *ObjectPropertyRange* represents the construct to define the range of an object property.

The remaining OWL object property axioms take an object property and assert a characteristic to it. In doing so, object properties can be defined to be functional, inverse functional, reflexive, irreflexive, symmetric,

2006–2007 © Copyright lies with the respective authors and their institutions.

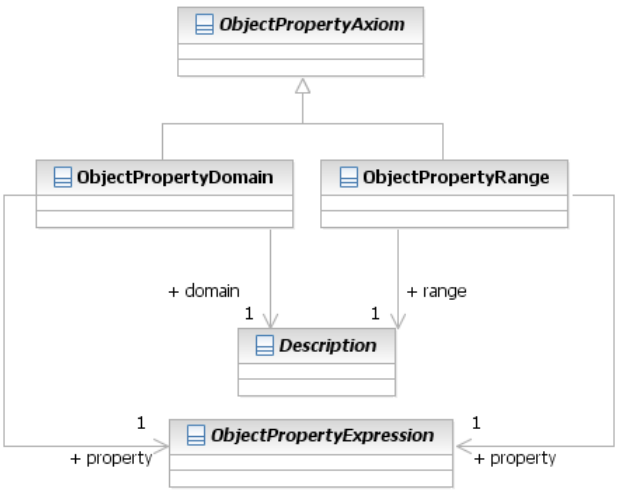


Figure 4.17: OWL metamodel: object property axioms - part 2

antisymmetric, or transitive.

A functional object property is a property for which each subject from the domain, can have only one value in the range, whereas an inverse functional object property can have only one subject in the domain for each value in the range. A transitive property defines that when the subject-object pairs (x,y) and (y,z) belong to the property extension, then the pair (x,z) belongs to the property extension as well.

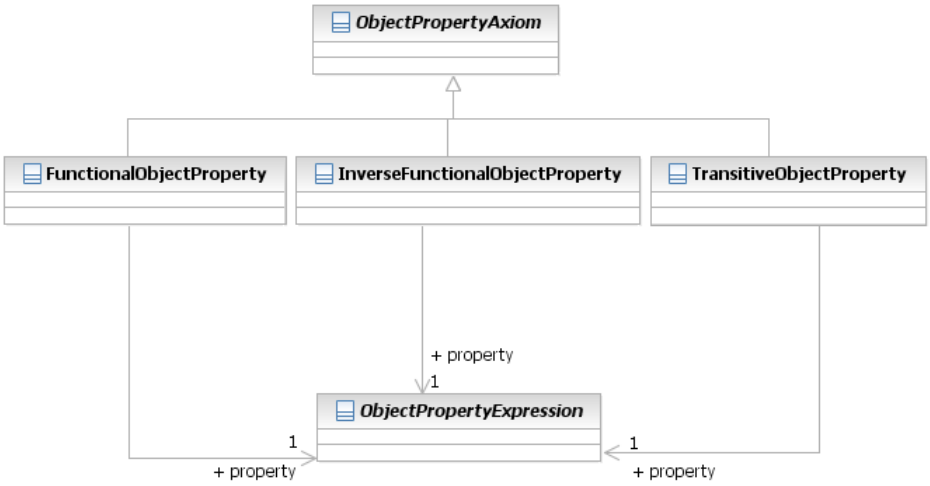


Figure 4.18: OWL metamodel: object property axioms - part 3

When an object property is defined to be reflexive, then for each object x, the subject-object pair (x,x) belongs to the object property extension. The opposite, when for each object x, the subject-object pair (x,x) does not belong to the object property extension, can be specified as an irreflexive object property. When an object property is specified to be symmetric or antisymmetric, then the pair (y,x) does respectively does not belong to the object property extension when the subject-object pair (x,y) belongs to the object property extension[11] (see Example 13).

[11] Note that the correct name for this relation would be 'Asymmetric'. However, the current OWL specifications call it 'antisymmetric'.

Example 13 *The following example illustrates how an object property is defined to be antisymmetric: AntisymmetricObjectProperty(madeFromIngredient)*

Figure 4.19: OWL metamodel: object property axioms - part 4

Figures 4.18 and 4.19 demonstrates that each of these axioms has an own metaclass with an association *property* to the class *ObjectPropertyExpression*, specifying the property on which the characteristic is defined.

4.7 Data Property Axioms

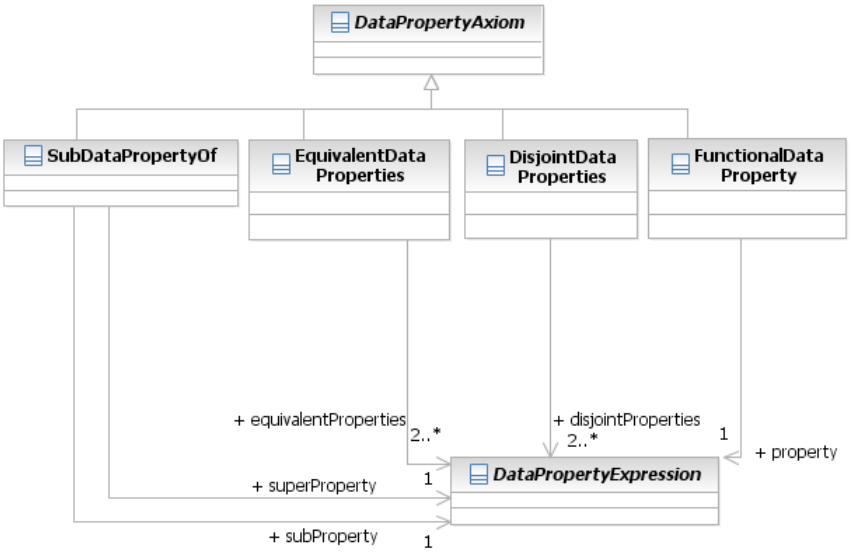


Figure 4.20: OWL metamodel: data property axioms - part 1

Similar to object property axioms, OWL provides six types of axioms on data properties.

2006–2007 © Copyright lies with the respective authors and their institutions.

Data properties can be defined to be a subproperty of another data property, or can be defined as being functional. Additionally, two or more data properties can be defined to be equivalent or disjoint. Figure 4.20 presents the corresponding metamodel elements.
Finally, OWL provides constructs to define the domain and range of a data property (see Example 14).

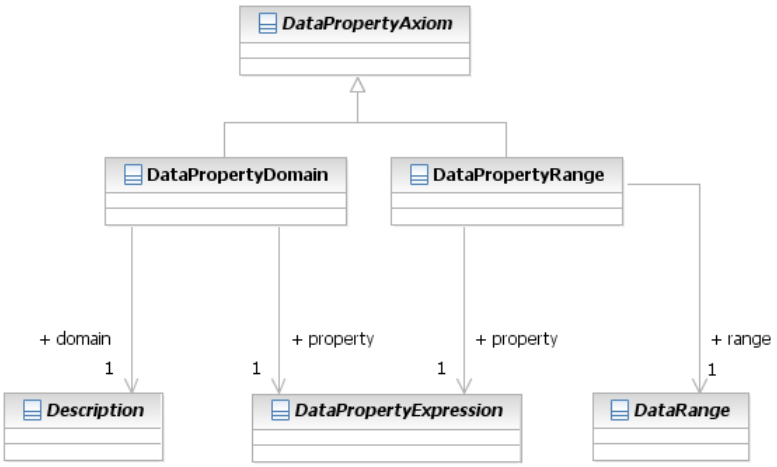


Figure 4.21: OWL metamodel: data property axioms - part 2

Example 14 *The following example illustrates the specification of the range of a data property: DataObjectPropertyRange(hasMinimumAge xsd:nonNegativeInteger)*

Figure 4.21 gives the classes that represent them in the metamodel, *DataPropertyDomain* and *DataPropertyRange*. Both have an association *property* specifying the property, and additionally an association *domain* to *Description* specifying its domain, respectively an association *range* to *DataRange* specifying its range.

4.8 Facts

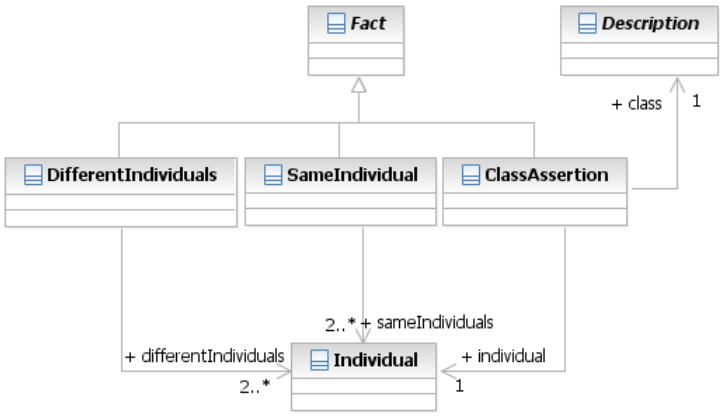


Figure 4.22: OWL metamodel: facts - part 1

Seven different axioms allow to state facts in ontologies. Figure 4.22 gives the metamodel representations for the first three of these axioms, stating facts about classes and individuals. Since OWL is made for use on the web and one can not assume that everyone uses the same name for the same thing on the web, OWL

does not have the so-called unique name assumption. Instead, OWL provides several constructs to define facts about the identity of individuals: individuals can be defined to denote the same object, represented in the metamodel by the metaclass *SameIndividual*, and two or more individuals can be defined to denote different objects, represented by the class *DifferentIndividuals*. Both metaclasses have an association to the metaclass *Individual* connecting the axiom with the individuals it applies. Moreover, a class assertion specifies to which class a certain individual belongs (see Example 15) and is defined in the metamodel as the metaclass *ClassAssertion* with an association *individual* to *Individual* and an association *class* to *Description*.

Example 15 *The following example illustrates how the class to which an individual belongs, is specified:*
ClassAssertion(sinusitis Disease)

The remaining fact axioms state facts about properties. To specify the value of a specified individual under a certain object property, OWL provides the object property assertion construct (see Example 16).

Example 16 *The following example illustrates how the value of an individual under an object property, is defined:*
ObjectPropertyAssertion(madeFromIngredient aspirin acetylsalicylicacid)

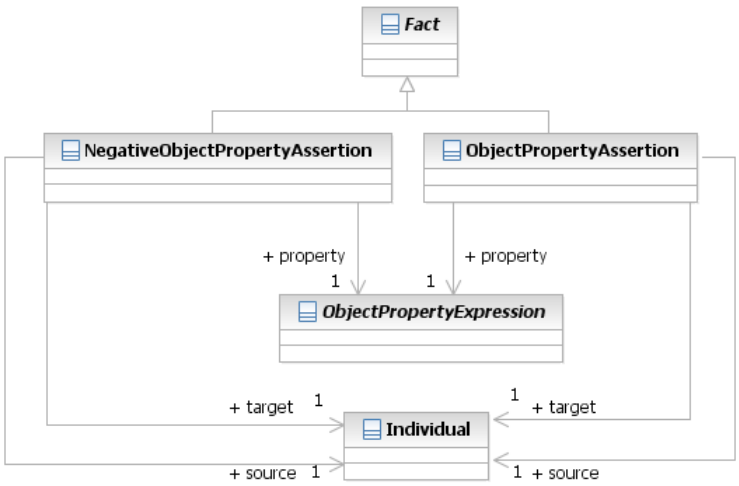


Figure 4.23: OWL metamodel: facts - part 2

The opposite is defined by a negative object property assertion, which defines that an individual is exactly not the value of another individual under the specified object property. Figure 4.23 shows that both constructs are represented in the metamodel as a metaclass, *ObjectPropertyAssertion* respectively *NegativeObjectPropertyAssertion*. Both have an association to *property* representing the involved property, and two associations to *Individual* representing the subject and the object of the assertion.

Finally, as the last constructs for facts in OWL, Figure 4.24 introduces instantiations of data properties. Similar to object property instantiations, OWL allows normal (positive) as well as negative data property assertions. Both metaclasses *DataPropertyAssertion* and *NegativeDataPropertyAssertion* have three associations connecting them to the property, an individual as the subject of the property assertion, and a constant as the property value.

2006–2007 © Copyright lies with the respective authors and their institutions.

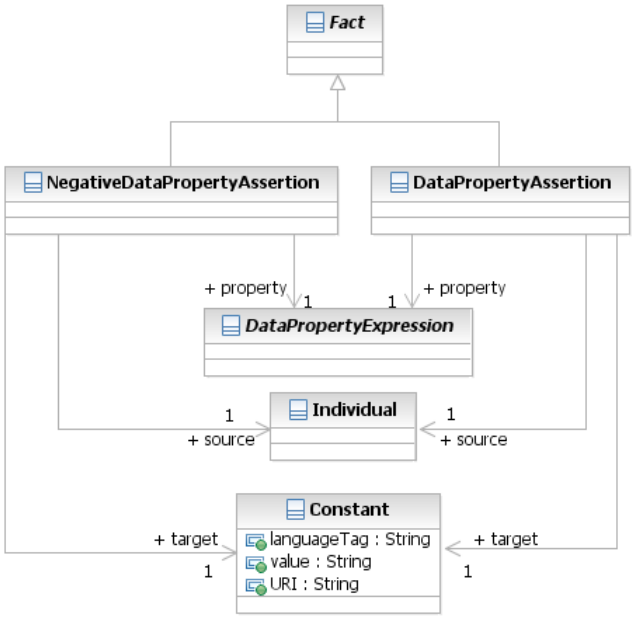


Figure 4.24: OWL metamodel: facts - part 3

Chapter 5

The Rule Metamodel

5.1 A Metamodel for SWRL Rules

We consistently extend our metamodel for OWL 1.1 presented in the previous section with a metamodel for SWRL that directly resembles the extensions of SWRL to OWL. Just like the OWL metamodel, this extension is augmented with OCL constraints to state the metamodel more precisely.

Three subsections introduce the different parts of the metamodel extension along with a discussion of the corresponding SWRL constructs.[1] [2]

The following subsections present the extension of the OWL metamodel for SWRL. Section 5.1.1 starts with rules, after which Section 5.1.2 presents predicate symbols. Finally, Section 5.1.3 presents terms.

5.1.1 Rules

The main component of OWL ontologies is the set of axioms. SWRL specifies a rule also as an axiom and Figure 5.1 demonstrates how our metamodel consequently represents it as a subclass of the abstract superclass *OWLAxiom*, called *Rule*. As all OWL axioms, a SWRL rule can be annotated, which is represented through an association of its superclass (Figure 4.1 on page 33). To identify a rule and assure compatibility with OWL, they can be assigned a URI, represented by the optional attribute *URI*.

A rule in SWRL contains an antecedent, also refered to as 'body', and a consequent, also refered to as 'head'. As we wanted to be able to treat them as first-class objects in our metamodel, we represent them by metaclasses, called *Antecedent* and *Consequent*. Both antecedent and consequent contain a number of atoms, possibly zero, and multiple atoms are treated as a conjunction in SWRL. Consequently, a rule actually says that *if* all atoms in the antecedent hold, *then* the consequent holds[3]. The fact that antecedent and consequent are possibly empty, is reflected in the metamodel by the multiplicity 'zero to many' on the association *bodyAtoms* from *Antecedent* to *Atom* and the association *headAtoms* between *Consequent* and *Atom*.

A SWRL atom is composed of a predicate symbol followed by an ordered set of terms. The connections between the atom and these atom components are represented in the metamodel through the associations from the metaclass *Atom* to the metaclasses *PredicateSymbol* and *Term*. Both *PredicateSymbol* and *Term*

[1] Classes or relationships that already exist in the OWL metamodel, can be recognized in the accompanying UML diagrams by colored elements and an appropriate little icon.

[2] Remember that the language SWRL itself does not belong to our contribution.

[3] An empty antecedent is treated as trivially true, i.e. satisfied by every interpretation, whereas an empty consequent is treated as false, i.e. not satisfied by any interpretation.

2006–2007 © Copyright lies with the respective authors and their institutions.

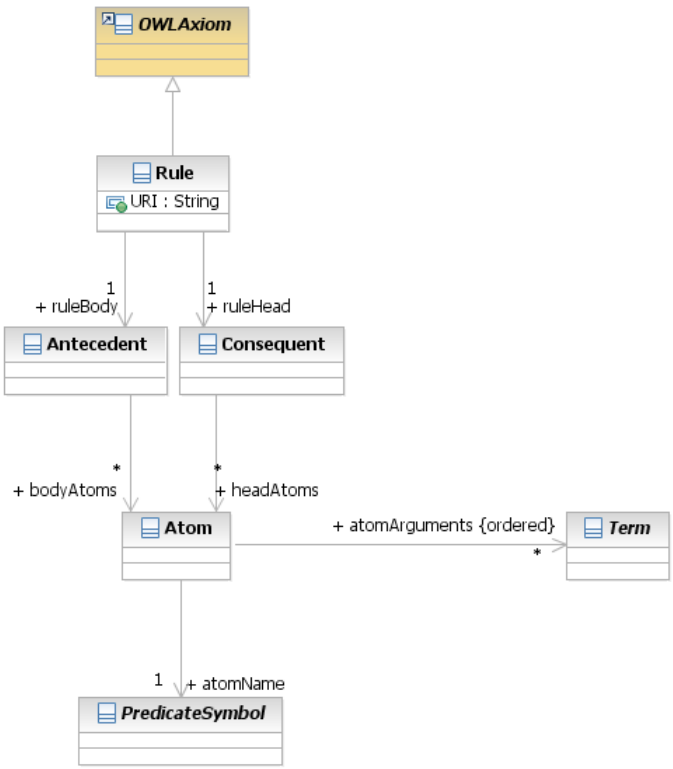


Figure 5.1: SWRL metamodel extension: rules

are abstract classes. Most of their subclasses exist already in the OWL metamodel.

5.1.2 Predicate Symbols

An atom in SWRL rules can have the following forms:

- An OWL class description, defined on a variable or an OWL individual. The atom holds if the value of the variable or individual belongs to the class description.
- An OWL data range specification using a variable or a data value. The atom holds if the variable or individual belongs to the data range.
- An OWL property. In case of an object property, the subject and object of the property are both an individual or a variable. If the specified property is a datatype property, the atom takes an individual or a variable as the subject, and a variable or data value as the object. The atom holds if the object of the property is related to the subject by the specified property.
- A 'sameAs' construct, defined on two objects which are both an individual or a variable. This construct is actually just some syntactic sugar and could be represented using other existing constructs. However, since it is useful in practice, the SWRL specifications define it as one of the basic constructs. The atom holds if the two terms are the same.
- A 'differentFrom' construct, defined on two objects which are both an individual or a variable. Just as well as *sameAs*, also *differentFrom* is syntactic sugar but very practical. The atom holds if the two terms are different.
- A built-in construct with a built-in ID and a set of variables and data values. Built-ins are classified into seven modules, like built-ins for lists or built-ins for comparisons. Implementations could select the

modules to be supported. The atom holds if the relation defined through the built-in ID, holds for the specified terms.

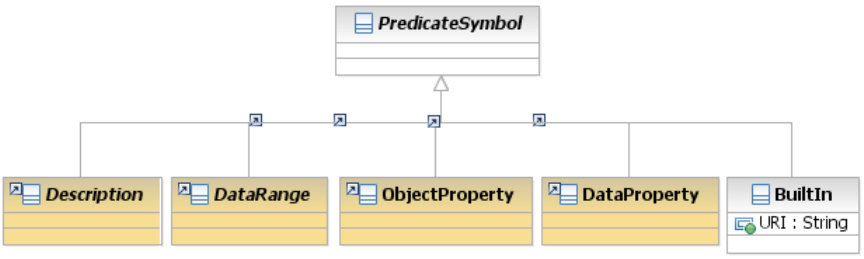


Figure 5.2: SWRL metamodel extension: predicate symbols

Figure 5.2 shows that almost all predicate are available in the OWL metamodel. The set of existing predicate symbols is extended in SWRL with one new predicate type, built-ins[4]. These built-ins are represented by the metaclass *BuiltIn* which has an attribute *URI* for their identification.

5.1.3 Terms

The last component in SWRL rules, is the set of terms. Figure 5.3 presents the metamodel elements for the possible terms in rule atoms: variables, constants and individuals.

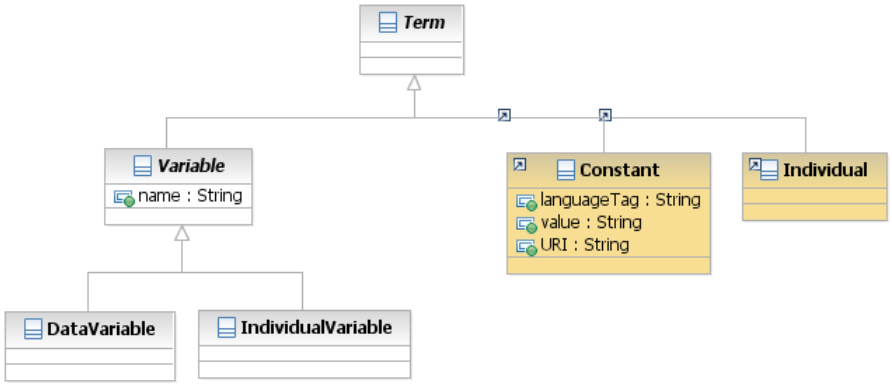


Figure 5.3: SWRL metamodel extension: terms

Variables do not exist in OWL, so they are newly introduced in the SWRL extension of the metamodel. Similar to the representation of data values and individuals, variables are also represented by classes. As two types of variables are allowed, an abstract superclass *Variable* is defined and both types of variables are defined by the subclasses *DataVariable* and *IndividualVariable*. The name of any variable is defined through the attribute *name* of the superclass, and their scope is always limited to the rule itself. Depending on the predicate symbol of the atom, only a certain number of terms as well as certain types of terms are allowed. These restrictions as well as the fact that variables that occur in the consequent must occur in the antecedent, are defined on the metamodel as OCL constraints:

1. When the predicate symbol of the atom is a description, the atom has exactly one argument and this argument must be an individual variable or an individual:
   context Atom inv:
   self.atomName.oclIsTypeOf(Description) implies

[4]Note that the predicates *sameAs* and *differentFrom* are represented through the metaclass *BuiltIn*.

2006–2007 © Copyright lies with the respective authors and their institutions.

self.atomArguments→size()=1 and
(self.atomArguments→at(1).oclIsTypeOf(Individual) or
self.atomArguments→at(1).oclIsTypeOf(IndividualVariable))

2. When the predicate symbol of the atom is a datarange, the atom has exactly one argument and this argument must be a constant or a data variable:
context Atom inv:
self.atomName.oclIsTypeOf(DataRange) implies
self.atomArguments→size()=1 and
(self.atomArguments→at(1).oclIsTypeOf(Constant) or
self.atomArguments→at(1).oclIsTypeOf(DataVariable))

3. When the predicate symbol of the atom is an object property, the atom has exactly two arguments and each of these is either an individual or an individual variable:
context Atom inv:
self.atomName.oclIsTypeOf(ObjectProperty) implies
self.atomArguments→size()=2 and
self.atomArguments→forAll(oclIsTypeOf(Individual) or
oclIsTypeOf(IndividualVariable))

4. When the predicate symbol of the atom is a data property, the atom has exactly two arguments and the first argument must be an individual or an individual variable, and the second argument must be a constant or a data variable:
context Atom inv:
self.atomName.oclIsTypeOf(DataProperty) implies
self.atomArguments→size()=2 and
(self.atomArguments→at(1).oclIsTypeOf(Individual) or
self.atomArguments→at(1).oclIsTypeOf(IndividualVariable)) and
(self.atomArguments→at(2).oclIsTypeOf(Constant) or
self.atomArguments→at(2).oclIsTypeOf(DataVariable))

5. When the predicate symbol of the atom is a built-in, the atom can have zero to many arguments and all these arguments can be either a constant or a data variable:
context Atom inv:
self.atomName.oclIsTypeOf(BuiltIn) implies
self.atomArguments→forAll(oclIsTypeOf(Constant) or oclIsTypeOf(DataVariable))

6. Only variables that occur in the antecedent may occur in the consequent of the rule:
context Rule inv:
self.ruleHead.headAtoms→atomArguments→forAll(t | t.oclIsTypeOf(Variable) implies
self.ruleBody.bodyAtoms→atomArguments→exists(t | true))

5.2 A Metamodel for F-Logic Rules

This chapter presents a metamodel for the language F-Logic. The metamodel for F-Logic is not related to the metamodel for OWL and SWRL in the sense that it is an extension, but it is a stand-alone MOF-based metamodel for a different language. The only relation that exists between the two is the fact that they are both defined in MOF, which allows to use the automatic transformation possibilities of the MDA framework. The F-Logic metamodel also contains OCL constraints to state it more precisely.

The discussion of the F-Logic metamodel starts with introducing the various metaclasses with their properties and constraints, while explaining the F-Logic constructs they represent[5]. UML diagrams accompany this discussion for the sake of clarity.

Eight subsections present the metamodel: Section 5.2.1 starts with F-Logic programs, after which Section 5.2.2 presents terms. Next, Section 5.2.3 presents Formulas and Section 5.2.4 presents rules and queries. Then, Section 5.2.5 gives logical connectives and Section 5.2.6 presents logical quantifiers. Finally, Section 5.2.7 demonstrates F-atoms and F-molecules, whereas Section 5.2.8 P-atoms.

5.2.1 F-Logic Programs

The F-Logic specifications define an F-Logic program as a set of facts and rules. Facts are expressions that represent statements about objects and their relationships. Queries take the form of rules without heads and operate on a given F-Logic program. Typically, F-Logic applications group the facts, rules and queries together and call it an F-Logic ontology.

Figure 5.4 describes the core of our metamodel and defines a metaclass *FLogicOntology* connected to a set of *Formula*s via the association *ontologyFormulas*. The class *Formula* is an abstract superclass of classes representing rules, facts and queries.
Formula also has subclasses that do not represent facts, rules or queries but are partial formulas that are contained in rules and queries. Due to the potential overlap between formulas that are contained directly in an ontology, and the set of partial formulas indirectly contained in an ontology, the metamodel generalizes both groups into one superclass.
OCL constraints specify which of the subclasses can be directly contained in *FLogicOntology*.

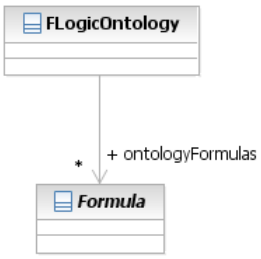


Figure 5.4: F-Logic metamodel: ontologies

5.2.2 Terms

The basic syntactical elements of F-Logic are predicate symbols, function symbols, and variables, where functional terms and variables are called id-terms in F-Logic, which identify objects, methods and classes. Functional terms are id-terms that have other id-terms as arguments. The F-Logic specifications define constants as functional terms with zero arguments. To distinguish between constants and variables, every variable has to be bound to a logical quantifier.

Figure 5.5 depicts how the various types of F-Logic id-terms are derived from an abstract metaclass called *Term*, carrying an attribute *name* that is inherited by all subclasses. Variables and functional terms are represented by the classes *Variable* respectively *FunctionalTerm*. An ordered association *arguments* to the class *Term* specify its arguments in the right order. The association carries a multiplicity of 'zero to many',

[5]Remember, however, that we did not contribute to the language F-Logic itself.

2006–2007 © Copyright lies with the respective authors and their institutions.

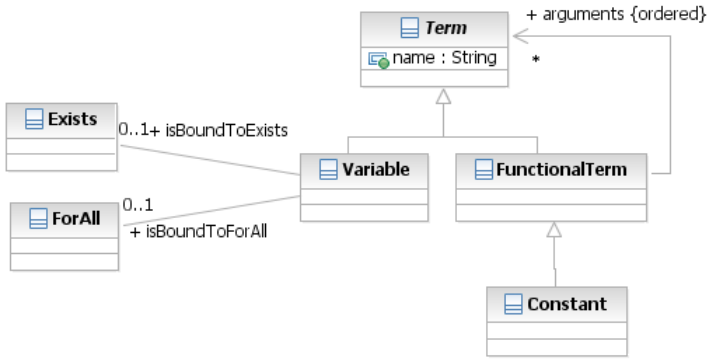


Figure 5.5: F-Logic metamodel: terms

denoted by a star, as the number of arguments in a functional term is unrestricted. For the specific type of functional terms that have zero arguments, constants, the metamodel has a specific class as it is a very common id-term. Two OCL constraints restrict the multiplicity of the association *arguments* for constants as well as for functional terms that are not a constant:

1. A constant is a functional term with zero arguments:
   context Constant inv:
   self.arguments→size()=0

2. A functional term that is not a constant must have at least one argument:
   context FunctionalTerm inv:
   self→forAll(not oclIsTypeOf(Constant) implies self.arguments→size()>0)

Finally, the metaclasses *Exists* and *ForAll* are connected to the class *Variable* via respective associations *isBoundToExists* and *isBoundToForAll* to connect a variable to the logical quantifier it is bound to.
Although every instantiation of *Variable* has to be connected to an instance of one of the two classes, the associations have multiplicity 'one or zero' as two multiplicities 'exactly one' would define that every variable needs to be connected to both classes. An OCL constraint makes sure that an instantiated variable does always have one connection to any of the two:

1. A variable must be bound to exactly one universal quantifier or existential quantifier:
   context Variable inv:
   (self.isBoundToExists=1 or self.isBoundToForAll=1)
   and
   (self.isBoundToForAll=1 implies self.isBoundToForExists=0)

5.2.3 Formulas

Facts, rules and queries are represented in the metamodel by subclasses of the metaclass *Formula*, which is linked to *FLogicOntology* through an association. Figure 5.6 gives an overview of the partial formulas that are derived from the class *Formula*. We distinguish four groups of partial formulas:

- Logical connectives: classes *Conjunction*, *Negation*, *Equivalence*, *Disjunction* and *Implication*
- Logical quantifiers: classes *ForAll* and *Exists*
- Facts: classes *PAtom*, *FMolecule* and *FAtom*

- Rules and queries: classes *Rule* and *Query*

An OCL constraint restricts the metamodel so that the logical connectives and quantifiers can not be contained directly into an ontology:

1. The only subtypes of the class *Formula* that can be directly in an ontology, are F-molecules (and so F-atoms), P-atoms, rules and queries:
   context FLogicOntology inv:
   self.ontologyFormulas→forAll(oclIsTypeOf(FMolecule) or oclIsTypeOf(Rule) or
   oclIsTypeOf(Query) or oclIsTypeOf(PAtom))

The next sections explain the details of the different subclasses of *Formula*, and explains which combinations are possible.

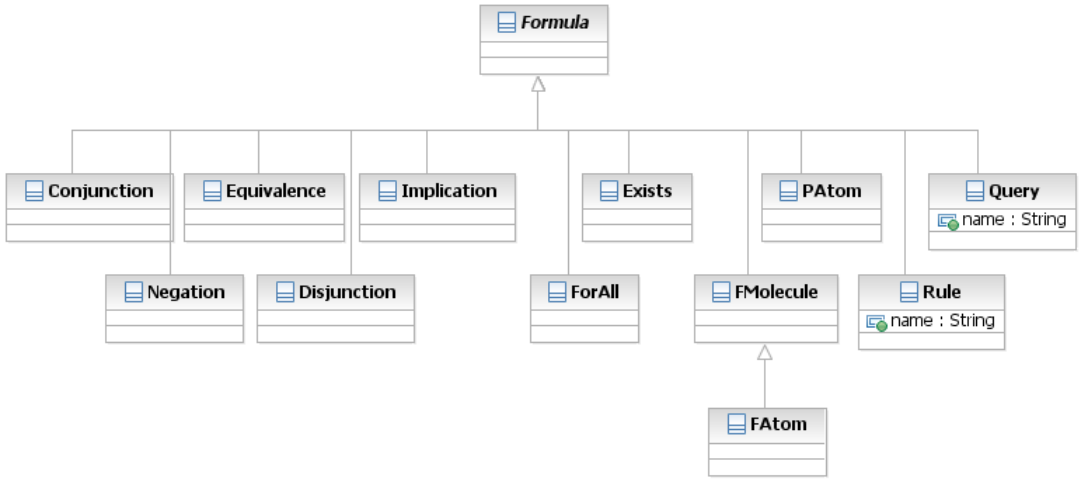


Figure 5.6: F-Logic metamodel: formulas

5.2.4 Rules and Queries

Rules (see Example 17) infer new information from available facts and so extend the object base intensionally. A rule consists of a head (postcondition), an implication sign ('⟵'), and a body (precondition). A rule defines that *when* the formula (or combination of formulas) in the body is satisfied, *then* the formula (or combination of formulas) in the head is satisfied. The rule head is a conjunction of P-atoms, F-Molecules and F-atoms, all connected through 'AND' . In the body, arbitrary formulas are allowed, and P-atoms, F-atoms and F-molecules can be connected by any of the logical connectives: implies ('→'), implied by ('←'), equivalent ('↔'), and ('AND'), or ('OR'), and not ('NOT').

Example 17 *A pharmacy that is a client of a certain pharmaceutical lab, can sell the medications produced by that lab:*
$\forall X, Y, Z\ X$: *Pharmacy*$[$*canSell* $\rightarrow Z]\ \longleftarrow$
X : *Pharmacy*$[$*clientOfÊ* $\rightarrow Y] \wedge Y$: *Laboratory*$[$*produces* $\rightarrow Z]$.

When a rule contains variables, they must be introduced using a quantifier. A universal quantifier ('∀') can appear in front of a rule or in the rule body, and an existential quantifier ('∃') can also appear anywhere in the rule body.

2006–2007 © Copyright lies with the respective authors and their institutions.

From the rules and facts in an F-Logic program, a model is computed on which queries (see Example 18) can be asked. The result of a query is a set of substitutions for the query's variables, that can be derived from the facts and rules in the knowledge base. Note that also schema level queries are allowed, where not only instances and their values but also concept and attribute names can be provided as answers via variable substitutions.

Example 18 *The following example query retrieves sets of variable substitutions 'medication, person, disease' whereas the medication is made from the ingredient 'acetylsalicylicacid', and the person has a disease against which the medication helps. The following values are a possible result of the query: X = Aspirin, Y = APerson, Z = headache:*
$\forall X, Y, Z \longleftarrow X : \mathit{Medication}[\mathit{madeFromIngredient} \rightarrow \mathit{acetylsalicylicacid}, \mathit{helps} \rightarrow Y] \wedge$
$Y[\mathit{hasDisease} \rightarrow Z]$

Figure 5.7 depicts the representation of rules and queries as the metaclasses *Rule* respectively *Query* in the metamodel. As the F-Logic specifications define that rules and queries can be given a name, both classes contains the attribute *name*.
Both *Rule* and *Query* are connected to the formula it contains through an association *containedFormula* carrying multiplicity 'one'[6].

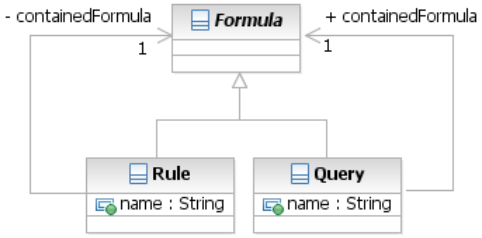


Figure 5.7: F-Logic metamodel: rules and queries

Several OCL constraints define how partial formulas can be combined in a rule:

1. A rule is an implication formula and possibly has a universal quantifier in front:
   context Rule inv:
   (self.containedFormula.oclIsTypeOf(Implication) or
   self.containedFormula.oclIsTypeOf(ForAll))
   and
   (self.containedFormula.oclIsTypeOf(ForAll) implies
   self.containedFormula.oclAsType(ForAll).containedFormula.oclIsTypeOf(Implication))

2. The head of a rule is a conjunction of facts or is just one fact (F-molecule or P-atom)[7]. The following constraint defines this for rules without a universal quantifier:
   context Rule inv:
   self.containedFormula.oclIsTypeOf(Implication) implies
   self.containedFormula.oclAsType(Implication).consequent.oclIsTypeOf(Conjunction) or
   self.containedFormula.oclAsType(Implication).consequent.oclIsTypeOf(FMolecule) or
   self.containedFormula.oclAsType(Implication).consequent.oclIsTypeOf(PAtom)

[6] Remember that when a rule or query consist of a combination of more than one formula, this is also represented as one formula, hence they can always contain maximum one formula.
[7] Note that another fact construct in F-Logic, F-atom, is defined as a subclass of *FMolecule* and thus is also included.

3. A similar constraint is defined for rules with universal quantifier:
   context Rule inv:
   self.containedFormula.oclIsTypeOf(ForAll) implies
   self.containedFormula.oclAsType(ForAll).containedFormula.oclAsType(Implication).
   consequent.oclIsTypeOf(Conjunction) or
   self.containedFormula.oclAsType(ForAll).containedFormula.oclAsType(Implication).
   consequent.oclIsTypeOf(FMolecule) or
   self.containedFormula.oclAsType(ForAll).containedFormula.oclAsType(Implication).
   consequent.oclIsTypeOf(PAtom)

4. When the head of a rule is a conjunction, the combined formulas may only be facts. The following constraint defines this for rules without a universal quantifier:
   context Rule inv:
   self.containedFormula.oclIsTypeOf(Implication) implies
   (self.containedFormula.oclAsType(Implication).consequent.oclIsTypeOf(Conjunction) implies
   self.containedFormula.oclAsType(Implication).consequent.
   oclAsType(Conjunction).connectedFormulas→forAll(oclIsTypeOf(FMolecule) or
   oclIsTypeOf(PAtom)))

5. A similar constraint is defined for rules with universal quantifier:
   context Rule inv:
   self.containedFormula.oclIsTypeOf(ForAll) implies
   (self.containedFormula.oclAsType(ForAll).containedFormula.oclAsType(Implication).
   consequent.oclIsTypeOf(Conjunction) implies
   self.containedFormula.oclAsType(ForAll).containedFormula.oclAsType(Implication).
   consequent.oclAsType(Conjunction).connectedFormulas→forAll(
   oclIsTypeOf(FMolecule) or oclIsTypeOf(PAtom)))

6. The body of a rule can be any formula except a rule or a query. The following constraint defines this for rules without a universal quantifier:
   context Rule inv:
   self.containedFormula.oclIsTypeOf(Implication) implies
   not self.containedFormula.oclAsType(Implication).antecedent.oclIsTypeOf(Rule)
   and
   not self.containedFormula.oclAsType(Implication).antecedent.oclIsTypeOf(Query)

7. A similar constraint is defined for rules with universal quantifier:
   context Rule inv:
   self.containedFormula.oclIsTypeOf(ForAll) implies
   not self.containedFormula.oclAsType(ForAll).containedFormula.oclAsType(Implication).
   antecedent.oclIsTypeOf(Rule) and
   not self.containedFormula.oclAsType(ForAll).containedFormula.oclAsType(Implication).
   antecedent.oclIsTypeOf(Query)

Three OCL constraints speciy how partial formulas are combined in queries:

1. A query is an implication formula with empty head, and can have a universal quantifier in front:
   context Query inv:
   (self.containedFormula.oclIsTypeOf(Implication) or
   self.containedFormula.oclIsTypeOf(ForAll))
   and
   (self.containedFormula.oclIsTypeOf(Implication) implies
   self.containedFormula.oclAsType(Implication).consequent→isEmpty)

2006–2007 © Copyright lies with the respective authors and their institutions.

and
(self.containedFormula.oclIsTypeOf(ForAll) implies
self.containedFormula.oclAsType(ForAll).containedFormula.oclIsTypeOf(Implication)
and self.containedFormula.oclAsType(ForAll).containedFormula.
oclAsType(Implication).consequent→isEmpty)

2. The formula in the body of the query can be any formula except a rule or a query. The following constraint defines this for queries without a universal quantifier:
context Query inv:
self.containedFormula.oclIsTypeOf(Implication) implies
not self.containedFormula.oclAsType(Implication).consequent.oclIsTypeOf(Rule)
and not self.containedFormula.oclAsType(Implication).consequent.oclIsTypeOf(Query)

3. A similar constraint is defined for queries with universal quantifier:
context Query inv:
self.containedFormula.oclIsTypeOf(ForAll) implies
not self.containedFormula.oclAsType(ForAll).containedFormula.
oclAsType(Implication).antecedent.oclIsTypeOf(Rule)
and not self.containedFormula.oclAsType(ForAll).containedFormula.
oclAsType(Implication).antecedent.oclIsTypeOf(Query)

5.2.5 Logical Connectives

Partial formulas can be combined using logical connectives to form more complex formulas. Not only facts (P-atoms, F-atoms and F-molecules) but also logical quantifiers or other formulas which already connected through logical connectives, can be combined. The following table gives the F-Logic symbols and the representation in the metamodel for the logical connectives in F-Logic:

F-Logic Symbol	Metamodel element
AND	class *Conjunction*
↔	class *Equivalence*
→	class *Implication*
NOT	class *Negation*
OR	class *Disjunction*

Figure 5.8 details logical connectives[8]. Each of these subclasses is connected to the class *Formula*, specifying the formulas that are combined.Ê *Conjunction* and *Disjunction* both have an association *connectedFormulas* with multiplicity 'two to many' in the metamodel, as they combine two or more partial formulas. *Equivalence* combines exactly two formulas, denoted by the multiplicity 'exactly two' on the association *connectedFormulas*. As a negation is only containing one formula, the association *connectedFormula* from *Negation* to *Formula* has multiplicity 'exactly one'. Finally, the metamodel provides two separate associations from *Implication* to the two formulas it combines, *antecedent* and *consequent*, as it is necessary to be able to distinguish between the two formulas as being the one before or the one behind the implication-sign. OCL constraints restrict the metamodel in the sense that the five logical connectives cannot be used in combining rules or queries:

1. A conjunction can not combine rules or queries:
context Conjunction inv:
self.connectedFormulas→forAll(not oclIsTypeOf(Rule) and not oclIsTypeOf(Query))

[8]Note that the two classes *Formula* in the figure represent the same metaclass but are only depicted twice for the sake of clarity, as allowed in UML diagrams.

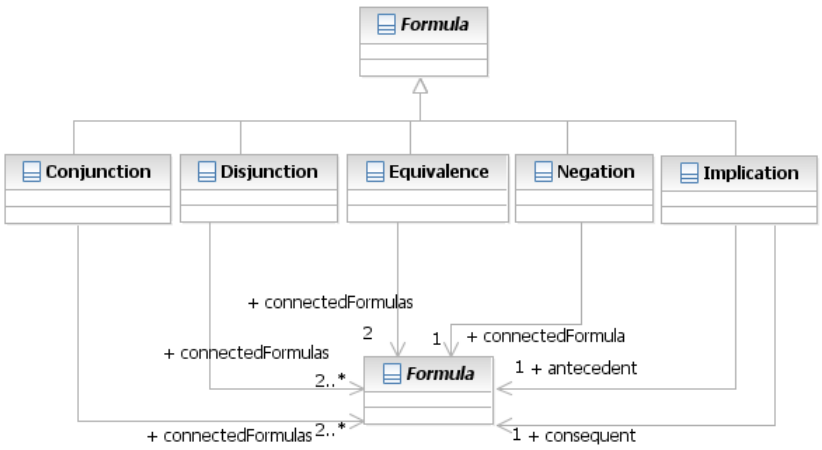


Figure 5.8: F-Logic metamodel: logical connectives

2. A disjunction can not combine rules or queries:
   context Disjunction inv:
   self.connectedFormulas→forAll(not oclIsTypeOf(Rule) and not oclIsTypeOf(Query))

3. An equivalence construct can not combine rules or queries:
   context Equivalence inv:
   self.connectedFormulas→forAll(not oclIsTypeOf(Rule) and not oclIsTypeOf(Query))

4. A negation can not be defined on a rule or a query:
   context Negation inv:
   self.connectedFormula→forAll(not oclIsTypeOf(Rule) and not oclIsTypeOf(Query))

5. An implication can not combine rules or queries:
   context Implication inv:
   self.antecedent→forAll(not oclIsTypeOf(Rule) and not oclIsTypeOf(Query)) and
   self.consequent→forAll(not oclIsTypeOf(Rule) and not oclIsTypeOf(Query))

5.2.6 Logical Quantifiers

F-Logic allows the two logical quantifiers $\forall$ and $\exists$, binding variables in a formula. Figure 5.9 shows how the metamodel represents the two logical quantifiers as the subclasses *Exists* and *ForAll* of the metaclass *Formula*. Both are connected to *Formula* via an association *containedFormula*, specifying the formula that is combined with the quantifier. To specify the variables that are bound to the logical quantifiers, the metamodel provides an association *boundVariables* from both to the metaclass *Variable*. To make sure that a variable that is connected to a quantifier through their association *boundVariables* is also connected to the quantifier in the other direction through the association *isBoundToExists* respectively *isBoundToForAll*, and vice versa, three OCL constraints are defined:

1. When a variable is bound to a quantifier in one direction, than this quantifier must have the variable defined as one of its bound variables in the other direction:
   context Variable inv:
   (self.isBoundToExists=1 implies
   self.isBoundToExists.boundVariables→exists(v: Variable | v=self)) and
   (self.isBoundToForAll=1 implies
   self.isBoundToForAll.boundVariables→exists(v: Variable | v=self))

2006–2007 © Copyright lies with the respective authors and their institutions.

2. When an existential quantifier has defined a variable as one of its bound variables, then this variable must be defined as bound to that quantifier:
   context Exists inv:
   self.boundVariables→forAll(v : Variable | v.isBoundToExists=self)

3. When a universal quantifier has defined a variable as one of its bound variables, then this variable must be defined as bound to that quantifier:
   context ForAll inv:
   self.boundVariables→forAll(v : Variable | v.isBoundToForAll=self)

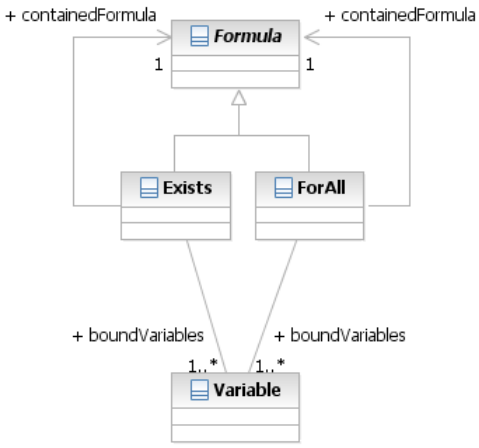


Figure 5.9: F-Logic metamodel: logical quantifiers

5.2.7 F-Atoms and F-Molecules

Let c, c_1, c_2, m, o, r, r_1, r_2, ..., r_n and t be F-Logic id-terms. Then we can define an F-atom as a fact expression with one of the following forms:

- instanceOf assertion: $o : c$ denotes that object o is an instance of class c.
- subclassOf assertion: $c_1 :: c_2$ denotes that class c_1 is a subclass of class c_2.
- single-valued method signature (method definition): $c[m \Rightarrow t]$ is a signature-atom specifying that the application of the single-valued (or functional) method m on an object of class c has as result an obect of type t, where single-valued or functional means that at most one object exists as value for the application of the method on an object.
- multi-valued method signature (method definition): $c[m \Rightarrow\!\!> t]$ is a multi-valued signature-atom denoting that the application of the method m on an instance of class c has a set of possible objects of type t as result.
- single-valued method application (method instantiation): $o[m \rightarrow r]$ expresses that the application of the method m on the object o has the object r as value.
- multi-valued method application (method instantiation): $o[m \twoheadrightarrow \{r_1, r_2, ..., r_n\}]$ expresses that a set of the objects r_1, r_2, ..., r_n is the result of the application of the method m on object o.

In the method signatures and method applications, the parts between the square brackets is referred to as the *method part*, whereas the object in front of the brackets is called the *host* in F-Logic. All method signatures and method applications can additionally have parameters.

To group several F-atoms about an object together conjunctively, F-Logic allows F-molecules. F-molecules can contain several method applications or method signatures in the method part of one construct, and moreover, not only a class but also an *instanceOf* or *subclassOf* assertion can appear as host in front of the method list[9]. For example

X : Researcher[authorOf $\rightarrow Y$; cooperatesWith $\rightarrow Z$]

is equivalent to

X : Researcher $\wedge\, X$[authorOf $\rightarrow Y$] $\wedge\, X$[cooperatesWith $\rightarrow Z$]
Ê

Figure 5.10 describes a metaclass *FMolecule* as a subclass of the metaclass *Formula* to represent an F-molecule. Because F-atoms are actually a special kind of F-molecules which can only have a simple term as host, and can only have one method in the method part, whereas F-molecules are unrestricted as regards the host objects as well as the number of methods, F-atoms are represented as a subclass of the metaclass *FMolecule*, called *FAtom*.
An association called *host* from the metaclass *FMolecule* to the metaclass *HostObject* links an F-molecule (and so an F-atom) to its host object, whereas an association called *methods* links an F-molecule (and so an F-atom) to its methods. Both F-molecules and F-atoms have always exactly one host object, which is denoted in the metamodel by the multiplicity 'exactly one' on the association *host*. The association *methods* has multiplicity 'zero to many' since an F-molecule has one or more methods, but an F-atoms has zero methods in case it has a simple term (a variable, a functional term or a constant) as host object. An OCL constraint restricts the metamodel with respect to the number of methods in F-atoms:

1. An F-atom is an F-molecule with exactly one method in case it has a simple term as host object, and zero methods otherwise:
   context FAtom inv:
   (self.host.oclIsTypeOf(Term) and self.methods→size()=1) or
   (not self.host.oclIsTypeOf(Term) and self.methods→size()=0)

Figure 5.11 shows the representation of the host objects in the metamodel. The three different types of host objects are defined as subclasses of the abstract superclass *HostObject*. The first type of host object is a simple term, represented by the metaclass *Term* which we introduced earlier and is an abstract supertype for variables, constants and functional terms. Secondly, a host object can be a combination of two terms of which one is defined to be the subclass of the other one, represented by the metaclass *SubClassOf*. The metaclass *SubClassOf* is connected to the two terms it combines, via the associations *subclass* and *superclass*. Thirdly, the host object can be a combination of two terms of which the first one is defined to be an instance of a specific class represented by the second term. This last type of host object is represented in the metamodel by the metaclass *InstanceOf*, which is connected to the metaclass *Term* via the associations *instance* and *class* specifying the terms that are combined in the construct.

Figure 5.12 shows how the methods are represented in the metamodel. The four different types of method signatures and method applications are all represented as a subclass of the abstract class *Method*: *Single-ValuedSignature*, *SingleValuedApplication*, *MultiValuedSignature* and *MultiValuedApplication*. Every type of method has a name, which is represented through the association *name* of their superclass. Additionally,

[9]Note that atoms can also be nested. For example, the value of a method application can be defined as an atom itself.

2006–2007 © Copyright lies with the respective authors and their institutions.

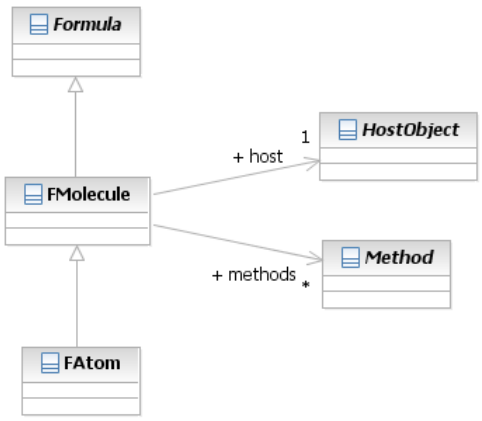


Figure 5.10: F-Logic metamodel: F-molecules

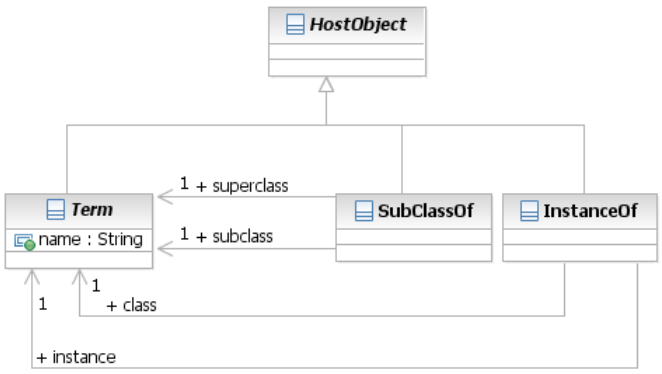


Figure 5.11: F-Logic metamodel: F-molecule host objects

the association *parameters* with multiplicity 'zero to many' represents the optional parameters a method in F-Logic can have.

The fact that all method types except the multi-valued method application have exactly one object as value, is represented by the association *value* with multiplicity 'exactly one' that the three metaclasses *SingleValuedSignature*, *SingleValuedApplication* and *MultiValuedSignature* share. As the multi-valued method application has a set of one or more method values, the association *value* between *MultiValuedApplication* and *MethodValue* has multiplicity 'one to many'[10].

As not only terms but also F-molecules itself are allowed as method values, an abstract metaclass *MethodValue* is defined in the metamodel for the different types of method values. The metaclasses *Term* and *FMolecule* which were introduced already earlier, are defined as subclasses of the metaclass *MethodValue*.

5.2.8 P-Atoms

F-Logic provides P-atoms for compatibility with languages like Datalog. P-atoms consist of a predicate symbol p and a list of id-terms F_1 to F_n as parameters: $p(F_1, ..., F_n)$. Information expressed by F-atoms can usually also be represented by P-atoms.

F-Logic additionally provides some built-in features, including several comparison predicates, the basic arithmetic operators, and so forth. Built-ins are actually specific, predefined P-atoms with at least one

[10]Note that, although a multi-valued method has more method values, the method value of a multi-valued method signature is defined as a class and hence it has not more than one method value.

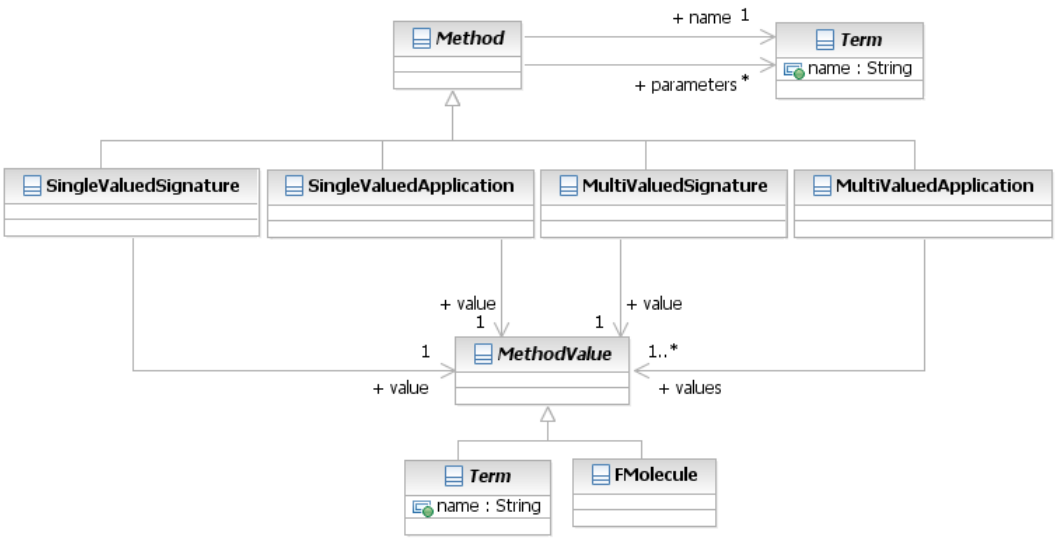


Figure 5.12: F-Logic metamodel: methods

parameter.

Figure 5.13 demonstrates the metamodel representation of P-atoms as the metaclass *PAtom*. The predicate symbol in front of the parameters in a P-atom is represented in the metamodel by the class *PredicateSymbol* which has an attribute *name* specifying the predicate's name. The metaclass *PAtom* is connected to the metaclass *PredicateSymbol* through the association *predicate* with multiplicity 'exactly one', as every P-atom must have a predicate symbol. The P-atoms's parameters are represented by instantiations of any subclass of the metaclass *Term*, to which the metaclass *PAtom* is connected with the ordered association *parameters*. This association carries a multiplicity 'zero to many' as a P-atom does not have any restrictions with respect to the number of parameters: it can contain zero but also many parameters.

As F-Logic built-ins are used in the same way as predicate symbols, the metamodel defines a metaclass *Builtin* as a subclass of the metaclass *PredicateSymbol*. An OCL constraint restricts the metamodel with respect to the number of parameters of a built-in:

1. If the predicate is a built-in, it must have at least one parameter:
   context PAtom inv:
   self.predicate.oclIsTypeOf(BuiltIn) implies self.parameters→size()>0

2006–2007 © Copyright lies with the respective authors and their institutions.

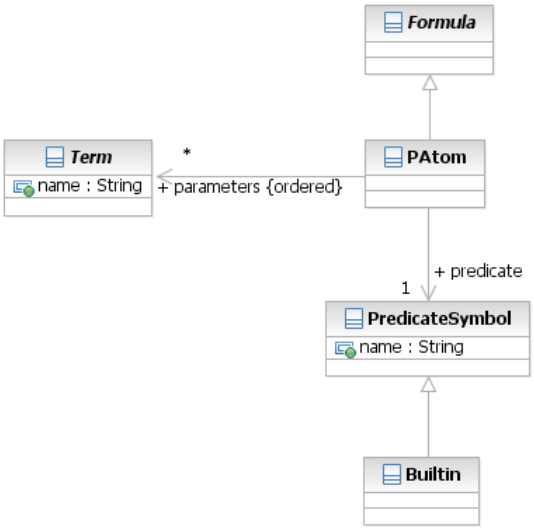


Figure 5.13: F-Logic metamodel: P-atoms

Chapter 6

The Mapping Metamodel

When people are modeling the same domain, they mostly produce different results, even when then use the same language. Mappings have to be defined between these ontologies to achieve an interoperation between applications or data relying on these ontologies. This chapter provides an extension for the metamodel for OWL and SWRL to give additional support for mappings between heterogeneous ontologies.

Section 6.1 introduces the metamodel extension in the same way as we did in the introduction of the metamodel for OWL and SWRL, and for F-Logic. While introducing the various mapping aspects[1], we discuss their representation in the metamodel. Accompanying UML diagrams document the understanding of the metamodel.[2]

The metamodel is a common metamodel for the different OWL mapping languages. On top of this common metamodel, we define two sets of constraints to concretize it to two specific OWL ontology mapping languages, DL-safe mappings and C-OWL. Section 6.2 starting on page 69 presents the extension for C-OWL mappings, consisting of a set of constraints. Similarly, Section 6.3 starting on page 70 presents the extension for DL-Safe Mappings.

6.1 A Common MOF-based Metamodel Extension for OWL Ontology Mappings

This section presents the common metamodel extension for OWL ontology mappings in two subsections: Section 6.1.1 presents mappings, after which Section 6.1.2 presents queries.

6.1.1 Mappings

We use a mapping architecture that has the greatest level of generality in the sense that other architectures can be simulated. In particular, we make the following choices:

- A mapping is a set of mapping assertions that consist of a semantic relation between mappable elements in different ontologies. Figure 6.1 demonstrates how this structure is represented in the metamodel by the five metaclasses *Mapping*, *MappingAssertion*, *Ontology*, *SemanticRelation* and *MappableElement* and their associations.
- Mappings are first-class objects that exist independent of the ontologies. Mappings are directed, and there can be more than one mapping between two ontologies. The direction of a mapping is defined

[1] Remember, however, that the OWL ontology mapping languages and their general aspects, are not part of our contribution.
[2] In doing so, meta-classes that are colored or carry a little icon again denote elements from the metamodel for OWL or SWRL.

2006–2007 © Copyright lies with the respective authors and their institutions.

through the associations *sourceOntology* and *targetOntology* of the metaclass *Mapping*, as the mapping is defined from the source to the target ontology. The cardinalities on both associations denote that to each *Mapping* instantiation, there is exactly one *Ontology* connected as source and one as target.

These choices leave us with a lot of freedom for defining and using mappings. For each pair of ontologies, several mappings can be defined or, in case of approaches that see mappings as parts of an ontology, only one single mapping can be defined. Bi-directional mappings can be described in terms of two directed mappings.

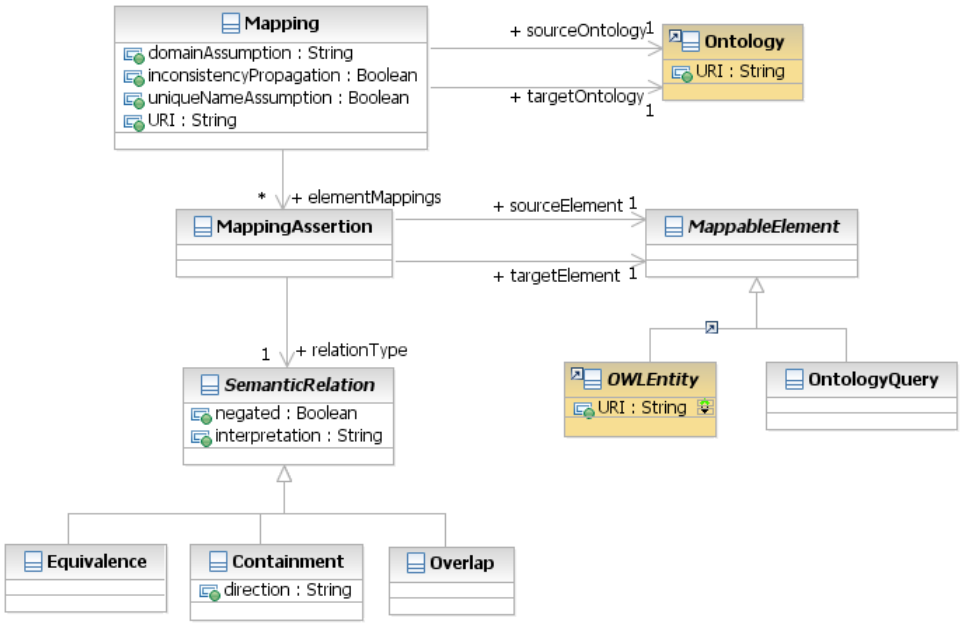


Figure 6.1: OWL mapping metamodel: mappings

The central class in the mapping metamodel, the class *Mapping*, is given four attributes. For the assumptions about the domain, the metamodel defines an attribute *DomainAssumption*. This attribute may take specific values that describe the relationship between the connected domains: *overlap*, *containment* (in either direction) or *equivalence*.

The question of what is preserved by a mapping is tightly connected to the hidden assumptions made by different mapping formalisms. A number of important assumptions that influence this aspect have been identified and formalized in [SSW05a]. A first basic distinction concerns the relationship between the sets of objects (domains) described by the mapped ontologies. Generally, we can distinguish between a global domain and local domain assumption:

Global Domain assumes that both ontologies describe exactly the same set of objects. As a result, semantic relations are interpreted in the same way as axioms in the ontologies. This domain assumption is referred to as *equivalence*, whereas there are special cases of this assumption, where one ontology is regarded as a global schema and describes the set of all objects, other ontologies are assumed to describe subsets of these objects. Such domain assumption is called *containment*.

Local Domains do not assume that ontologies describe the same set of objects. This means that mappings and ontology axioms normally have different semantics. There are variations of this assumption in the sense that sometimes it is assumed that the sets of objects are completely disjoint and sometimes they are assumed to overlap each other, represented by the domain assumption called *Overlap*.

These assumptions about the relationship between the domains are especially important for extensional mapping definitions, because in cases where two ontologies do not talk about the same set of instances, the extensional interpretation of a mapping is problematic as classes that are meant to represent the same aspect of the world can have disjoint extensions.

The second attribute of the metaclass *Mapping* is called *inconsistencyPropagation*, and specifies whether the mapping propagates inconsistencies across mapped ontologies. *uniqueNameAssumption*, the third attribute of the metaclass *Mapping*, specifies whether the mappings are assumed to use unique names for objects, an assumption which is often made in the area of database integration. The fourth attribute, *URI*, is an optional URI which allows to uniquely identify a mapping and refer to it as a first-class object.

The set of mapping assertions of a mapping is denoted by the relationship between the two classes *Mapping* and *MappingAssertion*. The elements that are mapped in a *MappingAssertion* are defined by the class *MappableElement*. A *MappingAssertion* is defined through exactly one *SemanticRelation*, one source *MappableElement* and one target *MappableElement*. This is defined through the three associations starting from *MappingAssertion* and their cardinalities.

A number of different kinds of semantic relations have been proposed for mapping assertions and are represented as subclasses of the abstract superclass *SemanticRelation*:

Equivalence ($\equiv$) Equivalence, represented by the metaclass *Equivalence*, states that the connected elements represent the same aspect of the real world according to some equivalence criteria. A very strong form of equivalence is equality, if the connected elements represent exactly the same real world object. Specific forms of the equivalence relation are to be defined as subclasses of *Equivalence* in the specific metamodels of the concrete mapping formalisms.

Containment ($\sqsubseteq, \sqsupseteq$) Containment, represented by the metaclass *Containment*, states that the element in one ontology represents a more specific aspect of the world than the element in the other ontology. Depending on which of the elements is more specific, the containment relation is defined in the one or in the other direction. This direction is specified in the metamodel by the attribute *direction*, which can be *sound* ($\sqsubseteq$) or *complete* ($\sqsupseteq$). If this attribute value is *sound*, the source element is more specific element than the target element. In case of the attribute value *complete*, it is the other way around, thus the target element is more specific than the source element.

Overlap (o) Overlap, represented by the metaclass *Overlap*, states that the connected elements represent different aspects of the world, but have an overlap in some respect. In particular, it states that some objects described by the element in the one ontology may also be described by the connected element in the other ontology.

In some approaches, these basic relations are supplemented by their negative counterparts, for which the metamodel provides an attribute *negated* for the abstract superclass *SemanticRelation*. For example, a negated *Overlap* relation specifies the disjointness of two elements. The corresponding relations can be used to describe that two elements are *not* equivalent ($\not\equiv$), *not* contained in each other ($\not\sqsubseteq$) or *not* overlapping or disjoint respectively ($\emptyset$). Adding these negative versions of the relations leaves us with eight semantic relations that cover all existing proposals for mapping languages.

In addition to the type of semantic relation, an important distinction is whether the mappings are to be interpreted as extensional or as intensional relationships, specified through the attribute *interpretation* of the metaclass *SemanticRelation*.

Extensional The extension of a concept consists of the things which fall under the concept. In extensional mapping definitions, the semantic relations are interpreted as set-relations between the sets of objects represented by elements in the ontologies. Intuitively, elements that are extensionally the same have to represent the same set of objects.

2006–2007 © Copyright lies with the respective authors and their institutions.

Intensional The intension of a concept consists of the qualities or properties which go to make up the concept. In the case of intensional mappings, the semantic relations relate the concepts directly, i.e. considering the properties of the concept itself. In particular, if two concepts are intensionally the same, they refer to exactly the same real world concept.

As mappable elements, the metamodel contains the class *OWLEntity* that represents an arbitrary part of an ontology specification. While this already covers many of the existing mapping approaches, there are a number of proposals for mapping languages that rely on the idea of view-based mappings and use semantic relations between (conjunctive) queries to connect models, which leads to a considerably increased expressiveness. These queries are represented by the metaclass *OntologyQuery*. Note that the metamodel in principle supports all semantic relations for all mappable elements. OCL constraints for specific mapping formalisms can restrict the combinations of semantic relations and mappable elements.

6.1.2 Queries

A mapping assertion can take a query as mappable element. Figure 6.2 demonstrates the class *Query* that reuses constructs from the SWRL metamodel.

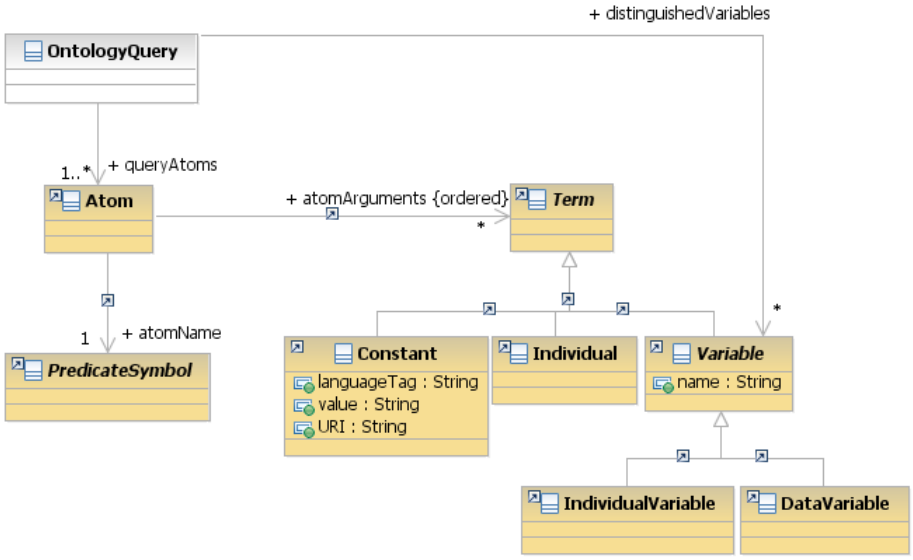


Figure 6.2: OWL mapping metamodel: queries

We reuse large parts of the rule metamodel as conceptual rules and queries are of very similar nature [TF05]: A rule consists of a rule body (antecedent) and rule head (consequent), both of which are conjunctions of logical atoms. A query can be considered as a special kind of rule with an empty head. The distinguished variables specify the variables that are returned by the query. Informally, the answer to a query consists of all variable bindings for which the grounded rule body is logically implied by the ontology. A *Query* atom also contains a *PredicateSymbol* and some, possibly just one, *Term*s. In the SWRL metamodel, we defined the permitted predicate symbols through the subclasses *Description*, *DataRange*, *DataProperty*, *ObjectProperty* and *BuiltIn*. Similarly, the different types of terms, *Individual*, *Constant*, *IndividualVariable* and *DataVariable* are specified as subclasses of *Term*. Distinguished variables of a query are differentiated through an association between *Query* and *Variable*. An OCL constraint a restriction on the use of distinguished variables:

1. A variable can only be a distinguished variable of a query if it is a term of one of the atoms of the query:
   self.distinguishedVariables→forAll(v: Variable |
   self.queryAtoms→exists(a: Atom | a.atomArguments→exists(v | true)))

6.2 OCL Constraints for C-OWL

We define OCL constraints on the common mapping metamodel extension to concretize it according to the specific formalism C-OWL [BGvH+03]. We list the specific characteristics of C-OWL and introduce the necessary constraints for them. For each constraint, firstly the context of the constraint, so the class in the metamodel on which it is to be defined, is defined using the OCL syntax "context *classname* inv:"[3]. Some existing reasoners support only a subset of C-OWL. Additional constraints could be defined to support this.

1. C-OWL does not have unique name assumption. To reflect this in the metamodel, a constraint defines that the value of the attribute *uniqueNameAssumption* of the class *Mapping* is always 'false':
   context Mapping inv:
   self.uniqueNameAssumption = false

2. C-OWL does not have inconsistency propagation. Similary, a constraint is defined to set the value of the attribute *inconsistencyPropagation* of the class *Mapping* to 'false':
   context Mapping inv:
   self.inconsistencyPropagation = false

3. The relationship between the connected domains of a mapping in C-OWL is always assumed to be *overlap*. A constraint sets the value of the attribute *domainAssumption* of the class *Mapping* to 'overlap':
   context Mapping inv:
   self.domainAssumption = 'overlap'

4. C-OWL does not allow to define mappings between queries. Moreover, only object properties, classes and individuals are allowed as mappable elements. A constraint defines that any mappable element in a mapping must be an *ObjectProperty*, an *OWLClass* or an *Individual*:
   context MappableElement inv:
   self.oclIsTypeOf(ObjectProperty) or
   self.oclIsTypeOf(OWLClass) or
   self.oclIsTypeOf(Individual)

5. A mapping assertion can only be defined between elements of the same kind. As the previous constraint defines that mappings can only be defined between three specific types of elements, an additional constraint can easily define that when the source element is one of these specific types, then the target elements must be of that type as well:
   context MappingAssertion inv:
   (self.sourceElement.oclIsTypeOf(OWLClass) implies
   self.targetElement.oclIsTypeOf(OWLClass)) and
   (self.sourceElement.oclIsTypeOf(ObjectProperty) implies
   self.targetElement.oclIsTypeOf(ObjectProperty)) and
   (self.sourceElement.oclIsTypeOf(Individual) implies
   self.targetElement.oclIsTypeOf(Individual))

6. As semantic relations in mappings, C-OWL supports *equivalence*, *containment* (sound as well as complete), *overlap* and *negated overlap* (called *disjoint*). A constraint defines this by specifying which different subclasses of *SemanticRelation* are allowed. When only the non-negated version of the semantic relation is allowed, the constraint defines that the attribute *negated* of the class *SemanticRelation* is set to 'false':
   context SemanticRelation inv:
   (self.oclIsTypeOf (Equivalence) and self.negated = false) or

[3] where 'inv' stands for the constraint type *invariant*.

2006–2007 © Copyright lies with the respective authors and their institutions.

(self.oclIsTypeOf(Containment) and self.negated = false) or
self.oclIsTypeOf(Overlap)

6.3 OCL Constraints for DL-Safe Mappings

This section provides OCL constraints on the common metamodel for OWL ontology mappings to concretize it to the formalism DL-Safe Mappings [HM05]. Again, we highlight the specific characteristics of the language and provide the appropriate constraints.

1. DL-Safe Mappings do not have unique name assumption. To reflect this in the metamodel, a constraint defines that the value of the attribute *uniqueNameAssumption* of the class *Mapping* is always 'false':
   context Mapping inv:
   self.uniqueNameAssumption = false

2. DL-Safe Mappings always have inconsistency propagation. A constraint is defined to set the value of the attribute *inconsistencyPropagation* of the class *Mapping* to 'true':
   context Mapping inv:
   self.inconsistencyPropagation = true

3. The relationship between the connected domains of a DL-Safe Mapping is always assumed to be *equivalence*. A constraint sets the value of the attribute *domainAssumption* of the class *Mapping* to 'equivalence':
   context Mapping inv:
   self.domainAssumption = 'equivalence'

4. DL-Safe Mappings support mappings between queries, properties, classes, individuals and datatypes. A constraint defines that the type of a *MappableElement* must be one of these subclasses:
   context MappableElement inv:
   self.oclIsTypeOf(OntologyQuery) or
   self.oclIsTypeOf(ObjectProperty) or
   self.oclIsTypeOf(DataProperty) or
   self.oclIsTypeOf(OWLClass) or
   self.oclIsTypeOf(Individual) or
   self.oclIsTypeOf(Datatype)

5. In DL-Safe Mappings, elements being mapped to each other must be of the same kind. Thus, when one wants to map for instance a concept to a query, the concept should be modelled as a query. A constraint defines that when the source element is of a specific type, the target element must be of the same type:
   context MappingAssertion inv:
   (self.sourceElement.oclIsTypeOf(OntologyQuery) implies
   self.targetElement.oclIsTypeOf(OntologyQuery)) and
   (self.sourceElement.oclIsTypeOf(ObjectProperty) implies
   self.targetElement.oclIsTypeOf(ObjectProperty)) and
   (self.sourceElement.oclIsTypeOf(DataProperty) implies
   self.targetElement.oclIsTypeOf(DataProperty)) and
   (self.sourceElement.oclIsTypeOf(OWLClass) implies
   self.targetElement.oclIsTypeOf(OWLClass)) and
   (self.sourceElement.oclIsTypeOf(Individual) implies
   self.targetElement.oclIsTypeOf(Individual)) and
   (self.sourceElement.oclIsTypeOf(Datatype) implies
   self.targetElement.oclIsTypeOf(Datatype))

6. DL-Safe Mappings specify that queries that are mapped to each other, must contain the same distinguished variables. A constraint defines that when both elements are a query, each variable that exists as distinguished variable in the source element, must exist as distinguished variable in the target element:
   context MappingAssertion inv:
   self.sourceElement.oclIsTypeOf(Query) and
   self.targetElement.oclIsTypeOf(Query) implies
   self.sourceElement.oclAsType(Query).distinguishedVariables→
   forAll(v: Variable | self.targetElement.oclAsType(Query).distinguishedVariables→
   exists(v | true))

7. The interpretation of semantic relations in DL-Safe Mappings is always assumed to be extensional. A constraint defines that the value of the attribute *interpretation* of the class *SemanticRelation* must be set to 'extensional':
   context SemanticRelation inv:
   self.interpretation = 'extensional'

8. DL-Safe Mappings support the semantic relations *equivalence* and *containment* (sound as well as complete). A constraint specifies this by defining which types *SemanticRelation* can have and what its value for the attribute *negated* should be:
   context SemanticRelation inv:
   (self.oclIsTypeOf(Equivalence) and self.negated = false) or
   (self.oclIsTypeOf(Containment) and self.negated = false)

2006–2007 © Copyright lies with the respective authors and their institutions.

Chapter 7

The Metamodel for Modular Ontologies

In software engineering, a modular software is a program in which significant parts, modules, are identified. A module generally plays a particular role in the whole program that, when combined with the other modules, contributes to the objective of the overall software. Well known advantages of modular design include clarity, reusability, and extensibility: a module is designed to be an intrinsically self-contained and reusable component, defined and managed independently from the programs it is intended to be included in.

It is a common opinion that ontology engineering shares a lot with software engineering, and the advantages of having modular ontologies instead of big and monolithic ones are easy to understand. Ontology modules are made to be reusable knowledge components, facilitating collaborative design and maintenance within a network of ontologies. Therefore, the goal of this chapter is to come up with a language for defining and managing ontology modules, as a part of the NeOn metamodel. We here directly build on the OWL metamodel for the specification of the content of a module and the mapping metamodel to relate the content of heterogeneous modules.

7.1 Design Considerations

Modularization is essential to support collaborative ontology editing and browsing, reuse and selection. Within the networked scenario, ontologies are mainly built in distributed locations, which means naturally, they are modules in a network of ontologies. At the same time, ontologies are difficult to manage especially when they are interconnected in a network. Interdependencies between modules of interconnected ontologies have to be made clear so that these modules can be easily managed (e.g., reused, shared). Ontology selection and reuse will play a major part in the future of the Semantic Web because with the increasing number of ontologies available over the internet, people will more likely use ontologies just like program libraries by selecting and reusing ontology modules [dSM06]. Modularization will offer the possibility for a more fine-grained sharing mechanism where ontology modules are well encapsulated and ready for future reuse and development. Many formalisms for handling ontology modules have been proposed [BS03, SK03, SP04], however, they have their own limitations in some scenarios [SR06, BCH06b].

7.1.1 Existing Formalisms for Ontology Modularization

In the following we provide an overview of several modular ontology formalisms.

Modularization with OWL Import The OWL ontology language provides limited support to modular ontologies: an ontology document – identified via its ontology URI – can be imported by another document using the `owl:imports` statement. The semantics of this import statement is that all definitions contained in the imported ontology document are included in the importing ontology, as if they were defined in the importing ontology document. It is worth to mention that `owl:imports` is directed –only the importing ontology is

affected– and transitive –if A imports B and B imports C, then A also imports the definitions contained in C. Moreover, cyclic imports are allowed (e.g. A imports B and B imports A).

One of the most commonly mentioned weaknesses of the importing mechanism in OWL is that it does not provide any support for partial import [VOM02, PSZ06]. Even if only a part of the imported ontology is relevant or agreed in the importing ontology, every definition is included. Moreover, there is no logical difference between imported definitions and proper definitions in the importing ontology: they share the same interpretations.

Distributed Description Logics Unlike import mechanisms that include elements from some modules into the considered one, Distributed Description Logics (DDLs) [BS02] adopt a *linking mechanism*, relating the elements of "local ontologies" (called *context*) with elements of external ontologies (contexts). Each context M_i is associated to its own local interpretation. Semantic relations are used to draw correspondences between elements of local interpretation domains. These relations are expressed using *bridge rules* of the form:

- $i : \phi \xrightarrow{\sqsubseteq} j : \psi$ (*into* rule), with semantics: $r_{ij}(\phi^{I_i}) \subseteq \psi^{I_j}$
- $i : \phi \xrightarrow{\sqsupseteq} j : \psi$ (*onto* rule), with semantics: $r_{ij}(\phi^{I_i}) \supseteq \psi^{I_j}$

where $I_i = (\Delta_i, .^{\mathcal{I}_i})$ (respectively $I_j = (\Delta_j, .^{\mathcal{I}_j})$) is the local interpretation of M_i (respectively M_j), ϕ and ψ are formulae, and r_{ij} is a *domain relation* mapping elements of the interpretation domain of M_i to elements of the interpretation domain of M_j ($r_{ij} \subseteq \Delta_i \times \Delta_j$). We only discuss bridge rules between concepts (meaning that ϕ and ψ are concept names or expressions) since it is the only case that has reported reasoning support [ST05].

Bridge rules between concepts cover one of the most important scenarios in modular ontologies: they are intended to assert relationships, like concept inclusions, between elements of two different local ontologies. However, mainly because of the local interpretation, they are not supposed to be read as classical DL axioms. In particular, a bridge rule only affects the interpretation of the target element, meaning for example that $i : \phi \xrightarrow{\sqsubseteq} j : \psi$ is not equivalent to $j : \psi \xrightarrow{\sqsupseteq} i : \phi$.

Arbitrary domain relations may not preserve concept unsatisfiability among different contexts which may result in some reasoning difficulties [BCH06c]. Furthermore, while subset relations (between concept interpretations) is transitive, DDL domain relations are not transitive, therefore bridge rules cannot be *transitively reused* by multiple contexts. Those problems are recently recognized in several papers [BCH06b, BCH06c, SSW05b, SSW05a] and it is proposed that arbitrary domain relations should be avoided. For example, domain relations should be one-to-one [SSW05a, BCH06c] and non-empty [SSW05b].

Integrity and Change of Modular Terminologies in DDL Influenced by DDL semantics, [SK03] adopts a view-based information integration approach to express relationships between ontology modules. In particular, in this approach ontology modules are connected by conjunctive queries. This way of connecting modules is more expressive than simple one-to-one mappings between concept names but less expressive than the logical language used to describe concepts.

[SK03] defines an ontology module – abstracted from a particular ontology language – as a triple $M = (C, \mathcal{R}, \mathcal{O})$, where C is a set of concept definitions, $\mathcal{R}$ is a set of relation definitions and $\mathcal{O}$ is a set of object definitions. A conjunctive query Q over an ontology module $M = (C, \mathcal{R}, \mathcal{O})$ is defined as an expression of the form $q_1 \wedge ... \wedge q_m$, where q_i is a query term of the form $C(x)$, $R(x, y)$ or $x = y$, C and R concept and role names, respectively, and x and y are either variable or object names.

In a modular terminology it is possible to use conjunctive queries to define concepts in one module in terms of a query over another module. For this purpose, the set of concept definitions C is divided into two disjoint sets of internally and externally defined concepts C_I and C_E, respectively, with $C = C_I \cup C_E, C_I \cap C_E = \emptyset$.

2006–2007 © Copyright lies with the respective authors and their institutions.

An *internal concept* definition is specified using regular description logics based concept expressions with the form of $C \sqsubseteq D$ or $C \equiv D$, where C and D are atomic and complex concepts, respectively.

An *external concept* definition is an axiom of the form $C \equiv M : Q$, where M is a module and Q is a conjunctive query over M. It is assumed that such queries can be later reduced to complex concept descriptions using the query-rollup techniques from [HT00] in order to be able to rely on standard reasoning techniques. A modular ontology is then simply defined as a set of modules that are connected by external concept definitions. The semantics of these modules is defined using the notion of a distributed interpretation as introduced in the previous paragraph.

Although the definition of a module, in its abstract form shown above, may allow arbitrary concept, relation and object definitions, only concept definitions is studied in [SK03]. This is due to the focus of the approach to improve terminological reasoning with modular ontologies by precompiling implied subsumption relations. In that sense it can be seen as a restricted form of DDLs that enables improved efficiency for special T-Box reasoning tasks.

$\mathcal{E}$-Connection While DDLs are focused on one type of relation between module elements, concept inclusion, the $\mathcal{E}$-connection approach [KLWZ03, GPS04] allows to define *link properties* from one module to another. For example, if a module M_1 contains a concept named $1 : Fish$ and a module M_2 contains a concept named $2 : Region$, one can connect these two modules by defining a link property named $livesIn$ between $1 : Fish$ and $2 : Region$.

Formally, given ontology modules $\{L_i\}$, a (one-way binary) link $E \in \mathcal{E}_{ij}$, where $\mathcal{E}_{ij}, i \neq j$ is the set of all links from the module i to the module j, can be used to construct a concept in module i, with the syntax and semantics specified as follows:

- $\langle E \rangle (j : C)$ or $\exists E.(j : C) : \{x \in \Delta_i | \exists y \in \Delta_j, (x, y) \in E^M, y \in C^M\}$
- $\forall E.(j : C) : \{x \in \Delta_i | \forall y \in \Delta_j, (x, y) \in E^M \rightarrow y \in C^M\}\}$
- $\leq nE.(j : C) : \{x \in \Delta_i | \#(\{y \in \Delta_j | (x, y) \in E^M, y \in C^M\}) \leq n\}$
- $\geq nE.(j : C) : \{x \in \Delta_i | \#(\{y \in \Delta_j | (x, y) \in E^M, y \in C^M\}) \geq n\}$

where $M = \langle \{m_i\}, \{E^M\}_{E \in \mathcal{E}_{ij}} \rangle$ is a model of the $\mathcal{E}$-connected ontology, m_i is the local model of L_i; C is a concept in L_j, with interpretation $C^M = C^{m_j}$; $E^M \subseteq \Delta_i \times \Delta_j$ is the interpretation of a $\mathcal{E}$-connection E.

$\mathcal{E}$-connection restricts the terms of modules, as well as their local domains, to be disjoint. This can be a serious limitation in some scenarios, particularly because, to enforce domain disjointness, subclass relations cannot be declared between concept of two different modules.

P-DL **P-DL**, Package-based Description Logics [BCH06d], uses importing relations to connect local models. In contrast to OWL, which forces the model of an imported ontology be completely embedded in a global model, the P-DL importing relation is *partial* in that only commonly shared terms are interpreted in the overlapping part of local models. The semantics of P-DL is given as the follows: the *image domain relation* between local interpretations $\mathcal{I}_i$ and $\mathcal{I}_j$ (of package P_i and P_j) is $r_{ij} \subseteq \Delta_i \times \Delta_j$. P-DL importing relation is:

- one-to-one: for any $x \in \Delta_i$, there is at most one $y \in \Delta_j$, such that $(x, y) \in r_{ij}$, and vice versa.
- compositionally consistent: $r_{ij} = r_{ik} \circ r_{jk}$, where $\circ$ denotes function composition. Therefore, semantic relations between terms in i and terms in k can be inferred even if k doesn't directly import terms from i.

Thus, a P-DL model is a virtual model constructed from partially overlapping local models by merging "shared" individuals.

P-DL also supports selective knowledge sharing by associating ontology terms and axioms with “scope limitation modifiers (SLM)”. A SLM controls the visibility of the corresponding term or axiom to entities on the web,in particular, to other packages. The scope limitation modifier of a term or an axiom t_K in package K is a boolean function $f(p, t_K)$, where p is a URI of an entity, the entity identified by p can access t_K iff $f(p, t) = true$. For example, some representative SLMs can be defined as follows:

- $\forall p, public(p, t) := true$, means t is accessible everywhere.
- $\forall p, private(p, t) := (t \in p)$, means t is visible only to its home package.

P-DL semantics ensures that distributed reasoning with a modular ontology will yield the same conclusion as that obtained by a classical reasoning process applied to an integration of the respective ontology modules [BCH06c]. However, reported result [BCH06a] only supports reasoning in P-DL as extensions of the $\mathcal{ALC}$ DL. Reasoning algorithms for more expressive P-DL TBox, as well as for ABox reasoning, are still to be investigated.

7.1.2 Requirements for a Module Definition Languages

A Module is an Ontology. As shown in the above overview, there is generally no clear distinction between the notion of ontology and the one of ontology module. A modular ontology is made of smaller local ontologies that can be seen as self-contained and inter-related modules, combined together for covering a broader domain. Indeed, an ontology is not inherently a module, but rather plays the role of a module for other ontologies because of the way it is related to them in an ontology network. In other terms, an ontology module is a self-contained ontology, seen according to a particular perspective, namely *reusability*. The content of an ontology module does not differ from the one of an ontology, but a module should come with additional information about how to reuse it, and how it reuse other modules.

Encapsulation / Information Hiding. The idea of encapsulation is crucial in modular software development, but has not really been studied and implemented in the domain of ontologies yet. In software engineering, it rely on the distinction between the implementation, i.e. internal elements manipulated by the developer of the module, and the interface, i.e. the elements that are exposed to be reused. This distinction between interface and implementation cannot be clearly stated when using ontology technologies like OWL. However, the essential role of a module interface is to guide the reusability of the module, by exposing reusable elements and hiding intermediary internal ones (the "implementation details"). By defining the set of reusable entities of an ontology module (the *export interface*), the developer of this module provide entry points to it, and clearly states which are the elements that can be "safely reused". Elements that are hidden behind the interface can then evolve, be re-designed or changed, without affecting the importing modules relying on this export interface.

Partial Import. As already mentioned, the `owl:imports` mechanism has been criticized in several papers for being "global" (see e.g. [VOM02, PSZ06]): it is not possible when using this mechanism to import only the relevant and useful elements in the importing ontology. Allowing partial import has many advantages, among which *scalability* is probably the most obvious. For this reason, some intermediary solutions have been recently proposed, using, prior to import, ontology partitioning [SK04, GPSK05] techniques or some forms of reduction to a sub-vocabulary [SR06, Stu06, dSM06]. We believe that the set of elements that are used in an importing module should be explicitly stated in the module definition, so that the influence of the imported module is clarified. The semantics of the module definition language should reflect the idea of partial import by "ignoring" the definitions that are not related to the imported elements (the *import interface*), preventing the importing module to deal with irrelevant knowledge, and giving the developer of such a module the possibility to disagree with some part of the imported modules.

2006–2007 © Copyright lies with the respective authors and their institutions.

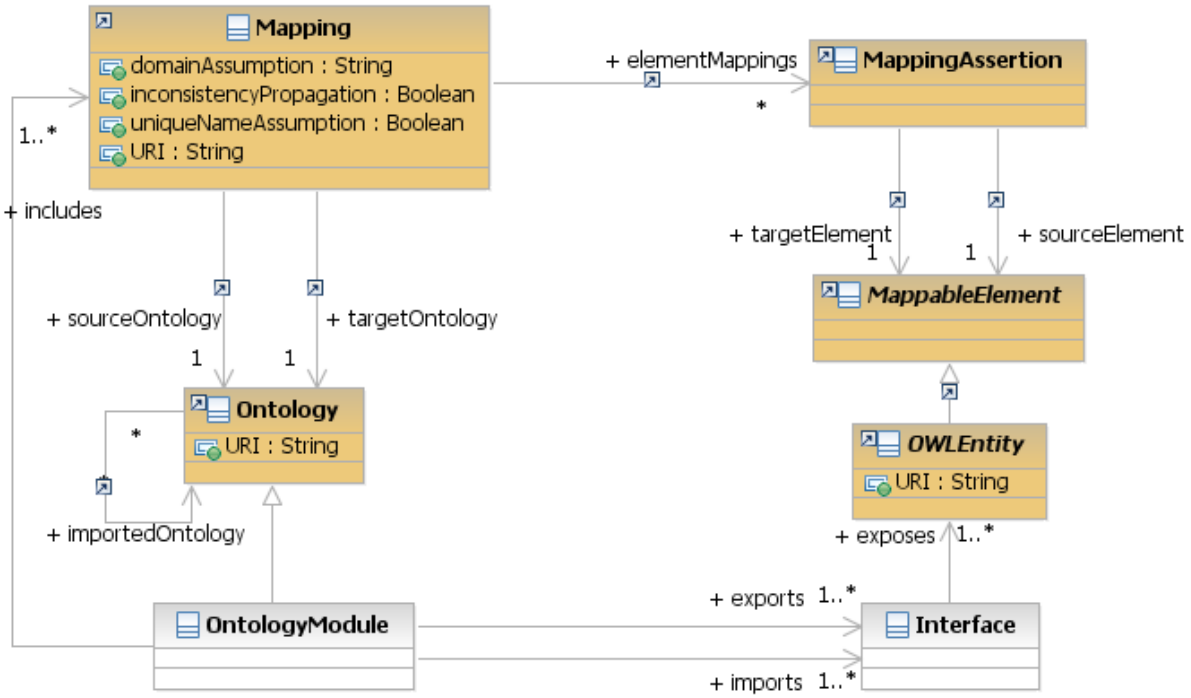


Figure 7.1: Metamodel extensions for ontology modules

Links Between Modules. The formalisms for modular ontologies presented in the previous section can be divided in two main approaches: importing and linking. The previous requirements are focused on the importing approach, whereas languages like C-OWL and $\mathcal{E}$-connection exclusively deal with the linking approach. In the NeOn framework, these two aspects are relevant, and should be considered together: even when they are not imported, elements from different modules can be related through mappings. The NeOn metamodel already provide the required elements for expressing mappings, and these can easily be considered as a part of the content of an ontology module. However, it is important to take this aspect into account when designing the semantics of the module description language, as well as the operations for manipulating modules, so that the two approaches, importing and linking, are well integrated. Indeed, scenarios where, for example, there exist mappings between imported modules are not hard to imagine.

7.2 A Metamodel for Modular Ontologies

We propose a generic metamodel for modular ontologies according to the design considerations discussed above. The metamodel is a consistent extension of the metamodels for OWL DL ontologies and mappings.

Figure 7.1 shows elements of the metamodel for modular ontologies. The central class in the metamodel is the class `OntologyModule`. A module is modeled as a specialization of the class `Ontology`. The intuition behind this modeling decision is that every module is also considered an ontology, enriched with additional features. In other words, a module can also be seen as a role that a particular ontology plays. In addition, an ontology provides an export interface and a set of import interfaces. The export interface, modeled via the `exports` association, `exposes` the set of `OntologyElement`s that are intended to be reused by other modules.

The reuse of a module by another module is modeled via the regular `imports` relationship already provided by the class `Ontology`. To each imported ontology module is associated an import interface, modeled through the `imports` association, that `exposes` the `OntologyElement`s the importing module

reuse from the imported module. Additionally, a `Module` also provides an `imports` relationship with the `Mapping` class, which is used to relate different ontology modules via ontology mappings.

7.2.1 Abstract Syntax for Ontology Modules

The goal of this section is to come up with an abstract syntax for the ontology modularization. The purpose of this is to identify the necessary information to be accommodated in an ontology module as well as structural properties of a modularized networked ontology. This will be done on a solid formal basis which will enable us to define a corresponding semantics at a later stage.
We start by defining sets of identifiers being used for unambiguously referring to ontology modules and mappings that might be distributed over the Web. Obviously, in practice, URIs will be used for this purpose. So we let

- Id_{Modules} be a set of MODULE IDENTIFIERS and
- Id_{Mappings} be a set of MAPPING IDENTIFIERS.

Next we introduce generic sets describing the used ontology language. They will be instantiated depending on the concrete ontology language formalism used (e.g., OWL). Hence, let:

- Nam be a set of language NAMED ELEMENTS.
  In the case of OWL, Nam will be thought to contain all class names, role names and individual names.
- $Elem$ be a the set of ONTOLOGY ELEMENTS.
  In the OWL case $Elem$ would contain e.g. all complex class descriptions. Clearly, $Elem$ will depend on Nam (or roughly speaking: Nam delivers the "building blocks" for $Elem$).
- We use $L : 2^{Nam} \to 2^{Elem}$ to denote the function assigning to each set P of named elements the set of ontology elements which can be generated out of P by the language constructs[1],
- For a given set O of ontology axioms, let $\mathrm{Sig}(O)$ denote the set of named elements occurring in O, so it represents those elements the axioms from O deal with.

Having stipulated those basic sets in order to describe the general setting, we are now ready to state the notion of an ontology module on this abstract level.

Definition 2 *An* ONTOLOGY MODULE $\mathcal{OM}$ *is a tuple* $\langle Imp, \mathcal{I}, M, O, E\rangle$ *where*

- Id_{Modules} *is the identifier of* $\mathcal{OM}$
- $Imp \subseteq Id_{\mathrm{Modules}}$ *is a set of identifiers of imported ontology modules (referencing those other modules whose content has to be (partially) incorporated into the module),*
- $\mathcal{I}$ *is the set* $\}I_{id}\}_{id\in Imp}$ *of* IMPORT INTERFACES*, with* $I_{id} \subseteq Nam$ *(characterizing which named elements from the imported ontology modules will be "visible" inside* $\mathcal{OM}$*),*
- $M \subseteq Id_{\mathrm{Mappings}}$ *is a set of identifiers of imported mappings (referencing – via mapping identifiers – those mappings between ontology modules, which are to be taken into account in* $\mathcal{OM}$*),*
- O *is a set of* ONTOLOGY AXIOMS *(hereby constituting the actual content of the ontology),*
- $E \subseteq \mathrm{Sig}(O) \cup \bigcup_{id\in Imp} \mathcal{I}_{id}$ *is called* EXPORT INTERFACE *(telling which named entities from the ontology module are "published", i.e., can be imported by other ontology modules).*

[1] In most cases – and in particular for OWL – $L(P)$ will be infinite, even if P is finite.

2006–2007 © Copyright lies with the respective authors and their institutions.

As a simple example let us consider an ontology module $\mathcal{OM}_i$ (with module identifier i) completely importing another ontology module $\mathcal{OM}_k$ (with module identifier k) and exporting everything:
$\mathcal{OM}_i = \langle i, \{k\}, \{\mathrm{Sig}(O_k)\}, \emptyset, O_i, \mathrm{Sig}(O_i) \cup \mathrm{Sig}(O_k)\rangle$
In a further step we formally define the term mapping (which is supposed to be a directed link between two ontology modules establishing semantic correspondences between them).

Definition 3 *A* MAPPING M *is a tuple* $\langle s, t, C\rangle$ *with*

- $s, t \in Id_{\mathrm{Modules}}$*, with* s *being the identifier of the source ontology module and* t *being the identifier of the target ontology module,*
- C *is a set of* CORRESPONDENCES *of the form* $e_1 \rightsquigarrow e_2$ *with* $e_1, e_2 \in Elem$ *and* $\rightsquigarrow \in R$ *for a fixed set* R *of* CORRESPONDENCE TYPES[2]

To finalize the definitional part, it remains to formally establish the connection between the ontology modules and mappings and their identifiers. So we define the *module space* which abstractly reflects the URI reference structure of the Web.

Definition 4 *A* MODULE SPACE *is defined as pair* $\langle \mathfrak{Mod}, \mathfrak{Map}\rangle$ *where*

- $\mathfrak{Mod} = \{\mathcal{OM}_{id}\}_{id \in Id_{\mathrm{Modules}}}$ *is a set of ontology modules (each endowed with a unique module identifier) and*
- $\mathfrak{Map} = \{M_{\mathrm{mid}}\}_{\mathrm{mid} \in Id_{\mathrm{Mappings}}}$ *is a set of mappings (with associated unique mapping identifiers).*

We conclude this section by defining additional requirements to modules and module spaces which will be assumed in the sequel:
We call a module space IMPORT-EXPORT-COMPATIBLE, if for every $\mathcal{OM}_{id_1} = \langle id_1, Imp, \mathcal{I}, M, O, E\rangle \in \mathfrak{Mod}$ and every $id_2 \in Imp$ with $\mathcal{OM}_{id_2} = \langle id_2, Imp', \mathcal{I}', M', O', E'\rangle$ we have that $I_{id_2} \subseteq E'$. Import-export-compatibility just states that every primitive being imported from a module must be exposed in that module's export.
For a given $id \in Id_{\mathrm{Modules}}$, we call the module $\mathcal{OM}_{id} = \langle Imp, \mathcal{I}, M, O, E\rangle$ of a module space $\langle \mathfrak{Mod}, \mathfrak{Map}\rangle$ MAPPING-COMPATIBLE, if for every $\mathrm{mid} \in M$ with $M_{\mathrm{mid}} = \langle s, t, C\rangle$ we have that $s, t \in Imp \cup \{id\}$. Mapping compatibility demands that if a mapping is imported into a module, both source module and target module of that mapping must be imported as well (or be the importing module itself).

[2] In accordance with the NeOn metamodel, this set will be fixed to $R = \{\sqsubseteq, \sqsupseteq, \equiv, \perp, \not\sqsubseteq, \not\sqsupseteq, \not\equiv, \not\perp\}$

Part III

The NeOn UML Profile for Networked Ontologies

2006–2007 © Copyright lies with the respective authors and their institutions.

Chapter 8

A UML Profile for Modeling Ontologies and Ontology Mappings

This chapter introduces the visual syntax for rule-extended ontologies and ontology mappings to support users unfamiliar with specific logical ontology modeling languages. We rely on the UML profile mechanism to adapt the language to the specifics needs in modeling ontologies.

Even when different ontologies are modeled using the same language, different people often model the same domain differently. Mappings have to be defined between these heterogeneous ontologies to achieve an interoperation between applications or data relying on these ontologies. To support the user in defining such mappings, we define an extension of the UML profile for ontologies and rules, to allow visual modeling of ontology mappings as well.
Our goal is to allow the user to specify mappings without having decided yet on a specific mapping language or even on a specific semantic relation. This is reflected in the proposed visual syntax which is, like the metamodel, independent from a concrete mapping formalism. The UML profile is consistent with the design considerations taken for the profiles for OWL ontologies and rule extensions.

Design Considerations Our goal is to provide a UML profile that it is cognitively more adequate for people familiar with UML concepts. Our profile utilizes the maximal intersection of features of both. Hence, classes are depicted as classes, properties as associations and individuals as UML objects. Furthermore, we heavily rely on the custom stereotypes and dependencies, and few stereotype tags. We provide an argumentation for the specific notations throughout the next sections.

We present the ontology profile in a series of examples while covering all OWL 1.1 constructs. Hereby, every example is presented both in the visual syntax and in the functional-style syntax. Section 8.1.6 starting on page 89 gives an overview of the profile. The profile extensions for rules and ontology mappings are represented similarly in Section 8.2 starting on page 92 and Section 8.3 starting on page 95.

8.1 A UML Profile for Ontologies

For presenting the UML constructs we propose for modelling ontologies, we start with the constructs for ontologies, their import statements and annotations. Further, entities (except properties) and data ranges are addressed, followed by class descriptions and class axioms. Next, we discuss the different properties and property axioms, after which we end with facts in OWL ontologies.

Figure 8.1: Ontology(http://www.uni-karlsruhe.de/PharmaceuticalDomain)

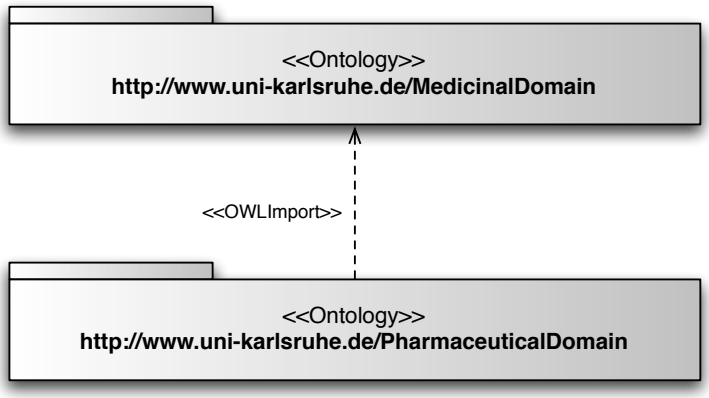


Figure 8.2: Ontology(http://www.uni-karlsruhe.de/PharmaceuticalDomain Import(http://www.uni-karlsruhe.de/Medication))

8.1.1 UML Syntax for Ontologies

Ontologies are represented by packages, which are *the* UML construct for packaging elements together. In this way, the ontology's axioms are contained in the package representing the ontology. The ontology's URI is represented by the package name. Fig 8.1 shows an example of an ontology called *PharmaceuticalDomain*. Note that we left out any axioms in the example.

An appropriately stereotyped dependency between two ontology packages indicates the import of one ontology into another. That way, Figure 8.2 demonstrates an ontology called *PharmaceuticalDomain* which imports the ontology *Medication*. Although UML provides a predefined import dependency between packages, it can not be used for our import statements since it means that the elements of the imported packages are integrated into the importing package. Also the specific access dependency in UML works in this way and so is not suited either.

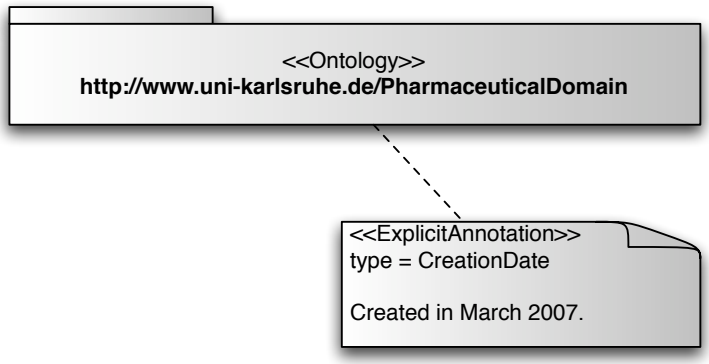


Figure 8.3: Ontology(PharmaceuticalDomain Annotation(CreationDate "Createdin March 2007."))

2006–2007 © Copyright lies with the respective authors and their institutions.

Figure 8.3 shows how ontologies can be annotated using the profile. We reuse the UML comment construct and specialize it with stereotypes, as it is originally provided by UML to annotate any UML construct. The example shows an explicit annotation which does not only carry a stereotype denoting the type of OWL annotation, but in case of an explicit annotation, the stereotype additionally has a tag to define the type of the annotation more specifically. When an annotation is defined on a combination of several UML constructs, the UML comment is attached to the main element connecting the others.

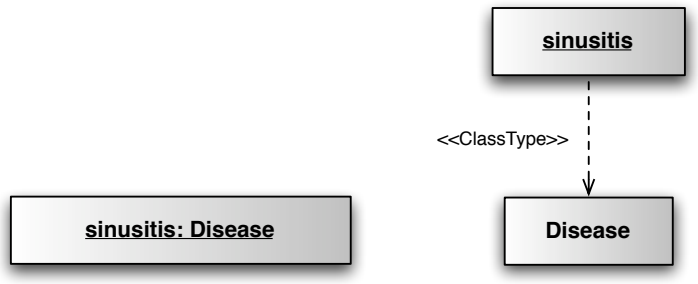
8.1.2 UML Syntax for Entities and Data Ranges

Figure 8.4: OWLClass(Medication)

We use the UML class notation for atomic classes, as well as for class descriptions where we rely on stereotypes. The first compartment of the class notation is mandatory and contains the class name as well as stereotypes. The second compartment can specify data properties. The third compartment is not being used, since no operations are specified in ontologies. Figure 8.4 shows an example of a class definition.

Figure 8.5: Individual(Aspirin)

Figure 8.5 shows how individuals are modeled using the UML object notation. Although, strictly seen, class assertions do not belong in this section on entities and data ranges, we address them here already since the discussion on class assertions is very close to the discussion of individuals as entities. Figure 8.6(a) demonstrates how a class specification names the class to which the individual belongs.



(a) Object (b) Association

Figure 8.6: ClassAssertion(sinusitis Disease)

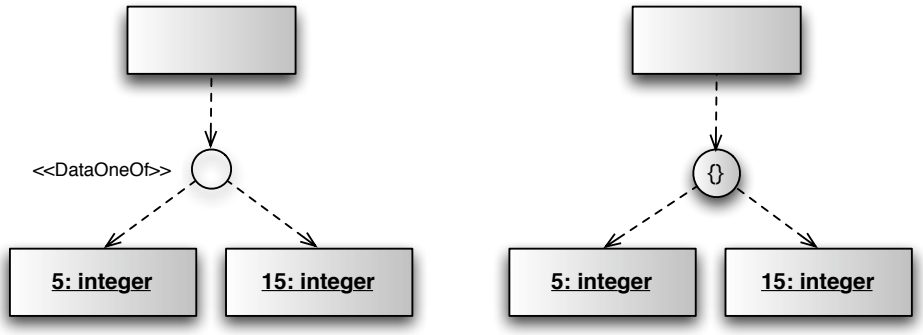
Figure 8.6(b) demonstrates an alternative notation using a stereotype «*ClassType*», which is necessary in the case of an instance of an anonymous class description.

The last group of constructs in this section are the data ranges. Figure 8.7 presents an example of a datatype. By default, a datatype is modeled in the form of a class. In case of a primitive datatype a specific stereotype

Figure 8.7: Datatype(xsd:nonNegativeInteger)

is provided. We will see later that the datatype can be depicted differently when it defines the data range of a data property.

For the enumeration of data values, an anonymous class is connected to the enumerated data values (in object notation) using dependencies connected with a small stereotyped circle (which is an available UML notation). Figure 8.8(a) shows an example of a datavalue enumeration in this notation. The example in Figure 8.8(b) applies a suitable icon as an alternative to the textual declaration with the stereotype. The specifications of the enumeration construct that is available in the UML metamodel are in several aspects not suitable for the OWL enumeration. One main reason for this is the restriction that in the UML enumeration, all values have to be from the same type. Since these specifications do not correspond to the enumeration characteristics in OWL, this can not be used as a possible notation.



(a) White dot (b) Icon

Figure 8.8: DataOneOf("5"^^xsd:integer "15"^^xsd:integer)

The complement of a data range is denoted via a stereotype $<<DataComplementOf>>$ in the data range. For the data range restriction, the restriction value and the facet type are added to the data range as attributes. Figure 8.9 shows an example of such a datatype restriction.

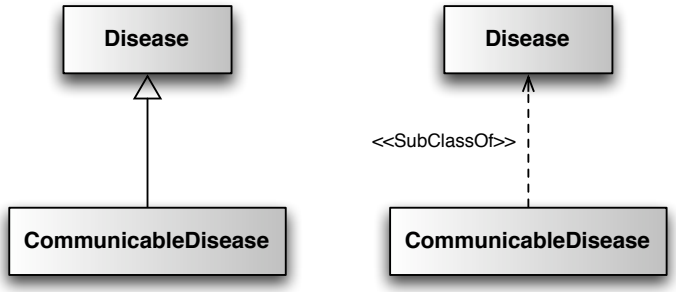
Figure 8.9: DatatypeRestriction(xsd:string minLength 4)

8.1.3 UML Syntax for Class Axioms and Class Descriptions

Figure 8.10(a) depicts how subclass-statements are depicted as generalizations. Class equality between two classes can be presented by a double-sided generalization arrow, as a simplified notation for two generalization arrows in opposite directions. Strictly speaking, this construction is not UML-conform, since the

2006–2007 © Copyright lies with the respective authors and their institutions.

UML-metamodel does not allow cycles in a generalization hierarchy. A UML-conform notation could depict subclasses and equivalent classes via appropriately stereotyped dependencies. Figure 8.10(b) shows this alternative notation. For equivalent classes, the dependency would be double-sided and carrying the stereotype $<<EquivalentClasses>>$. Note that by representing these constructs with generalization arrows, stereotypes are not needed.



(a) Generalization (b) Dependency

Figure 8.10: SubClassOf(CommunicableDisease Disease)

OWL 1.1 allows more than two classes in the construct for equivalent classes. Although the specific notation for two classes is most useful and intuitive for this particular case, an elaborated notation for the construct with more than two classes is necessary. For this we need a certain construct connected to the different classes of the definition. Clearly, a class box is inappropriate for this since we do not want to specify another class but only specify some characteristic of existing classes. For such constructs in OWL 1.1 where a characteristic is defined between several objects, we draw a stereotyped small circle with an appropriate stereotype and connect it to the classes of the definiton. Figure 8.11(a) gives an example of three classes defined to be equivalent. Figure 8.11(b) demonstrates an icon that provides an alternative for the stereotyped notation.

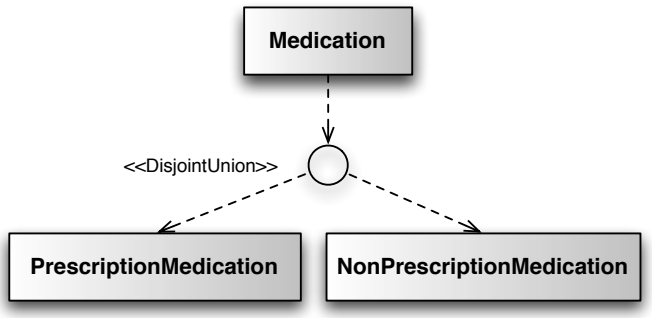
(a) White dot (b) Icon

Figure 8.11: EquivalentClasses(Disease Illness Sickness)

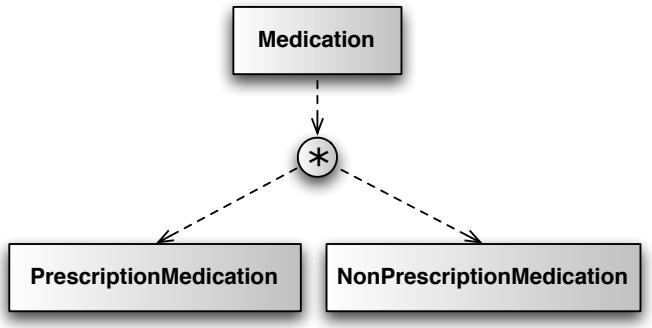
The disjointness of two classes is depicted similarly to the equivalence between two classes. A double-sided dependency with the stereotype $<<DisjointClasses>>$ connects the two classes. Also when more classes are involved in the statement, the notation is similar to the one for equivalent classes. The profile provides the '$\perp$' sign as an icon for the alternative notation.

As the last OWL 1.1 class axiom, Figure 8.12 presents the UML notation for defining a class as a union of other classes, all of which are pair-wise disjoint. Although the notation is very similar to the notation we just introduced for equivalent and disjoint classes, it is different in the sense that it has one class with a particular position. For this specific class, the dependency is directed towards instead of away from the icon or dot, as we saw it before for the enumeration of data values. Note that this notation for a disjoint union can be used with any number of classes in the definition.

After discussing the UML constructs for class axioms, we now examine the constructs for class descriptions. Except for *OWLClass*, which we addressed in the very beginning of this chapter, class descriptions are not



(a) White dot



(b) Icon

Figure 8.12: DisjointUnion(Medication PrescriptionMedication NonPrescriptionMedication)

used on their own, but are used in axioms. To give the reader a clear idea of how the constructs would be used, we will represent the class descriptions in an axiom defining a class as a subclass of the description. We provide a compact notation where the new class to be defined is affiliated with the anonymous class defined by the description itself. Additionally, for the descriptions we always take simple classes in our examples. The first group of descriptions, unions and intersections, can be depicted by applying the stereotypes $<<ObjectUnionOf>>$ and $<<ObjectIntersectionOf>>$ on the visual notation that we introduced earlier. Figure 8.13 demonstrates the union class description.

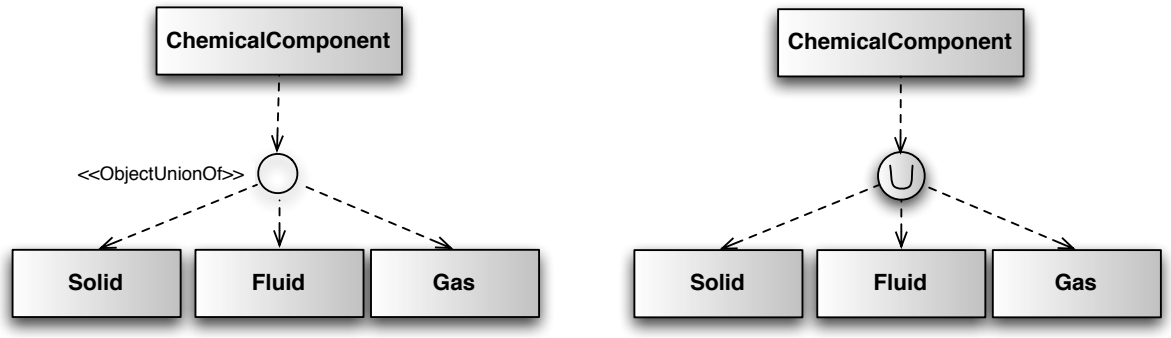
The OWL 1.1 construct defining the complement of a description has a connection from the anonymous class to exactly one description. As only one element is involved on each side, the dot notation is not necessary and a normal $<<ObjectComplementOf>>$ stereotyped dependency can be used.

The enumeration of individuals follows the same presentation pattern as the constructs before and the enumeration of data values. The class is connected with the enumerated individuals using a $<<ObjectOneOf>>$ stereotyped group of dependencies, or an icon.

Another means of defining classes in OWL 1.1 is by defining restrictions on properties. We represent the property by an association to the qualifying property range, which is a class in case of an object property and a data range in case of a data property. The intention of a UML association is exactly what we need to represent a property. We only add a restriction-specific stereotype to the association, and depict the property's name at the end of the arrow. For restrictions on data properties, an alternative notation would be the UML class attribute notation. This is only possible if the datarange is a datatype and otherwise, only the notation that we just explained can be used. The special notation for restrictions on data properties with data types as value, we introduce later on in other constructs using this notation. Also for an argumentation and a deeper discussion on the property notation, we refer the reader to the next section.

For cardinality restrictions, we rely on UML association multiplicities to define the cardinality. Figure 8.14 shows how UML multiplicities are applied for an OWL minimum cardinality on an object property. The notation

2006–2007 © Copyright lies with the respective authors and their institutions.



(a) White dot (b) Icon

Figure 8.13: SubClassOf(ChemicalComponent ObjectUnionOf(Solid Fluid Gas))

for maximum and exact cardinalities, as well as for these cardinalities on data properties, are similar.

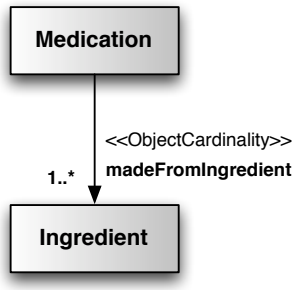


Figure 8.14: SubClassOf(Medication ObjectMinCardinality(1 madeFromIngredientIngredient))

When only the cardinality is explicitly specified but no range, the association of the unqualified cardinality restriction leads to the universal class owl:Thing (in case of an object property) or the unary datatype rdfs:Literal (in case of a data property).

It could be useful to allow merging several cardinality restrictions on the same property and so merge it into one visual construct. Note here that UML does not permit that the highest cardinality is lower than the minimum cardinality.

For value restrictions and existential restrictions, the indication of a multiplicity is not applicable. Otherwise, the notation stays the same as for cardinality restrictions. The stereotypes $<<$*ObjectSomeValuesFrom*$>>$, $<<$*ObjectAllValuesFrom*$>>$ and $<<$*ObjectExistsSelf*$>>$ are used. The notation for cardinality restrictions on data properties is exactly the same as for object properties, and utilizes the stereotypes $<<$*DataSomeValuesFrom*$>>$ and $<<$*DataAllValuesFrom*$>>$.

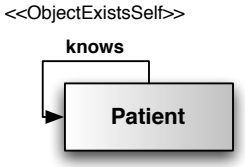


Figure 8.15: SubClassOf(Patient ObjectExistsSelf(knows))

The example of Figure 8.15 looks different but uses exactly the same pattern of a stereotyped association

with property name from a class to the range. In this case, the range is defined to be the class itself.

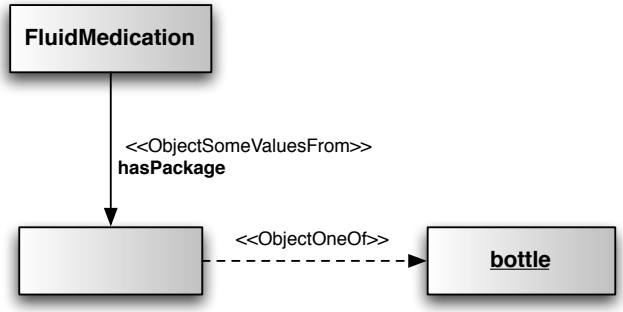


Figure 8.16: SubClassOf(FluidMedication ObjectHasValue(hasPackage bottle))

Figure 8.17: SubClassOf(MedicationForAdults DataHasValue(hasMinimumAge"12"^^xsd:integer))

Finally, the last description construct provided by OWL 1.1 is the property filler, for which we can not use the same pattern as we used for the descriptions before because UML does not allow to connect a class with an object using an association. Taking this into account, no more compact notation conform to the UML metamodel exists than a combination of an existential restriction and an enumeration [1]. Figure 8.16 demonstrates an example. In this manner, we actually build an anonymous class which consists of the individual defined in the property filler construct. Consequently, we define a class which has an instance of the anonymous class as value for the given property. Since the anonymous class contains only one individual, in this way we indirectly defined the property filler. When the property is a data property, the value for the property filler is defined to be a constant and we can utilize the compact class notation with its attributes. Figure 8.17 demonstrates this notation. A similar notation can be used for the restrictions we addressed before, in case of a datatype.

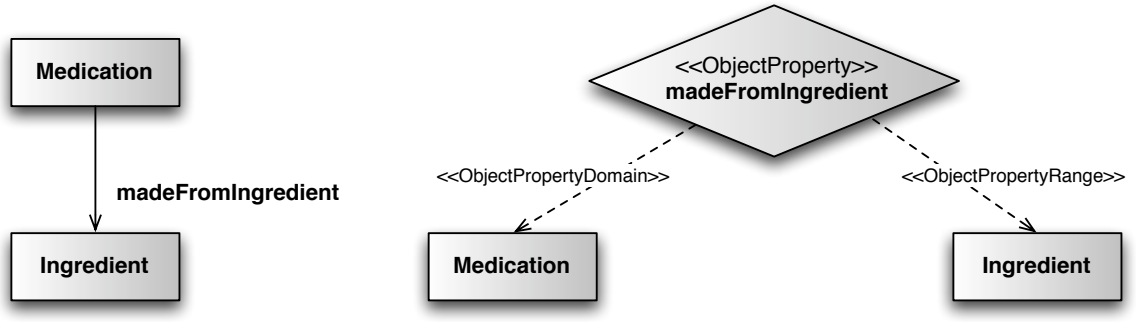
8.1.4 UML Syntax for Properties and Property Axioms

In UML, attributes and associations are *the* means to model characteristics of classes or relations between classes. To accomodate the UML-user, we represent properties by attributes and associations not only in the context of restrictions which we addressed in the former section. We discuss the associated problems and possible ways out.

Figure 8.18(a) shows that for object properties, the associated classes serve as initial and end point of a directed association. Figure 8.19(a) demonstrates how a data property is depicted as an attribute of the domain, whose type determines the range. This representation in which the properties are interpreted as attributes of its domain, can not be harmonized with the global character of an OWL property without violating the UML-metamodel. A property belongs to the namespace of its domain. For the illustration of properties using attributes, additional conventions are necessary. Moreover, this notation is perceived to be less intuitive when a property contains multiple domains and ranges. Then, for domains D_i and ranges R_i, it insinuates $D_1 \times R_1 \cup ... \cup D_n \times R_n$. Correct would be $(D_1 \cap ... \cap D_n) \times (R_1 \cap ... \cap R_n)$. Figure 8.19(b) demonstrates an alternative consisting in utilizing associations as with object properties. Figures 8.18(b) and 8.19(c) present one more alternative notation for properties. To be able to distinguish them from normal

[1] Note that a dependency is not suitable either, as the property name can not be defined on a dependency as on an association.

2006–2007 © Copyright lies with the respective authors and their institutions.



(a) Association (b) Diamond class

Figure 8.18: ObjectPropertyDomain(madeFromIngredient Medication) ObjectPropertyRange(madeFromIngredient Ingredient)

classes, we depict them in the form of a diamond. The classes of domains and ranges are thereby connected explicitly through stereotyped dependencies.

An object property can be connected to its inverse with a two-sided dependency arrow with stereotype *<<InverseObjectProperties>>*. Functionality, inverse functionality, reflexivity, irreflexivity, symmetry, antisymmetry and transitivity of object properties, as well as functionality of data properties is indicated by a stereotype. Figure 8.20 gives an example of an antisymmetric object property. Note that a model element in UML 2 can have several stereotypes. Although some of these characteristics could be represented utilizing cardinalities, we chose not to do so because of uniformity.

Similar to classes, property inclusion is depicted with a generalization arrow or the alternative dependency notation with the stereotypes *<<SubObjectPropertyOf>>* or *<<SubDataPropertyOf>>*. When more subproperties are contained in the definition, we apply a usual UML combination of generalization arrows into one arrow.

For properties defined to be equivalent, we provide the same notation as for classes. When only two properties are defined to be equivalent, a two-sided generalization arrow as well as an alternative notation with a *<<EquivalentObjectProperties>>* or *<<EquivalentDataProperties>>* dependency can be used. When more than two properties are specified, we reuse the notation with the dot or icon.

Disjoint properties are depicted in exactly the same way but with the stereotype *<<DisjointObjectProperties>>* or *<<DisjointDataProperties>>*. The generalization notation is not applicable in this case.

8.1.5 UML Syntax for Facts

We augment the UML notation for class assertions that was introduced earlier on with the remaining six types of facts in OWL 1.1. Figure 8.21 presents the notation for equality and inequality between two individuals using UML object relations. An appropriate stereotype, *<<SameIndividual>>* or *<<DifferentIndividual>>* defines the type of relation.

Although this notation is very useful when only two individuals are contained in the definition, we need an additional construct since OWL 1.1 allows more than two individuals in the construct. The notation we used before, for instance to define several classes to be equivalent (see Figure 8.11), seems adequate for this as well, and allows us to provide a UML profile with a maximum degree of uniformity.

Figure 8.22 demonstrates that the value of an object property is depicted as an object relation. When the construct defines a negative object property assertion, the association carries an appropriate stereotype *<<Not>>*.

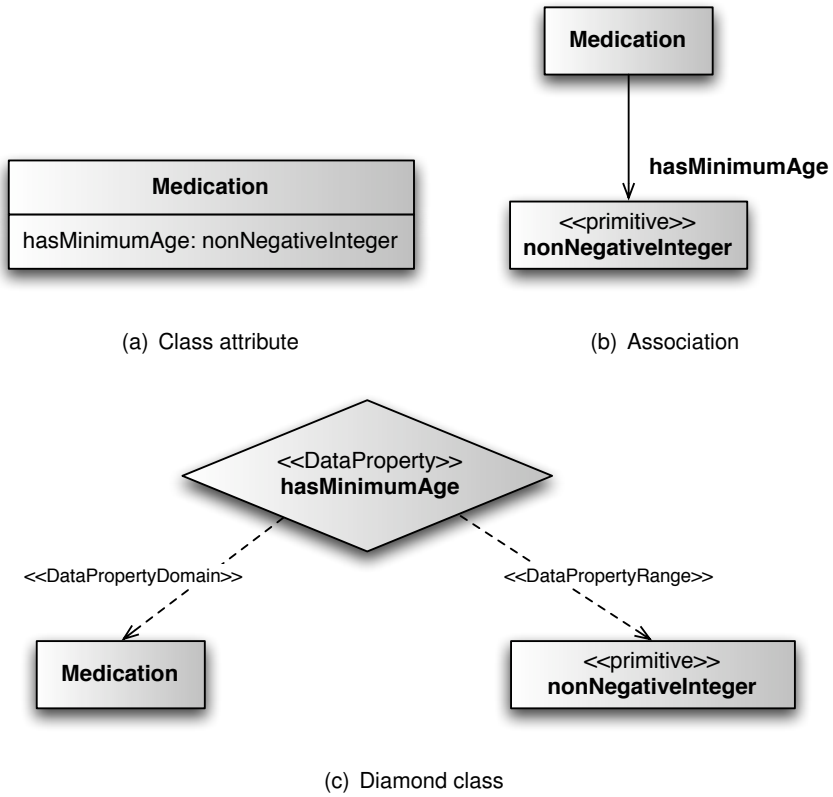


Figure 8.19: DataPropertyDomain(hasMinimumAge Medication)
DataObjectPropertyRange(hasMinimumAge xsd:nonNegativeInteger)

In case of a data property, the property value can alternatively be depicted as an attribute of the respective individual. Similary as with object properties, a stereotype is added when it is a negative property assertion. Figure 8.23 shows an example of such a negative data property assertion.

8.1.6 The Ontology UML Profile

The following table gives a complete overview of the profile for OWL ontologies. For each stereotype, we specify possible tags, as well as the element of the UML metamodel it is defined on.

2006–2007 © Copyright lies with the respective authors and their institutions.

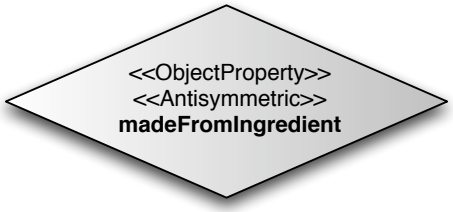


Figure 8.20: AntisymmetricObjectProperty(madeFromIngredient)

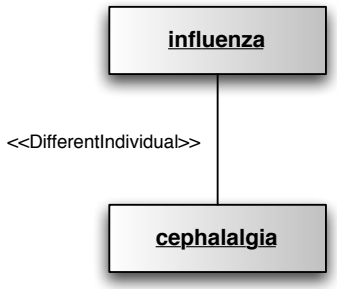


Figure 8.21: DifferentIndividual(influenza cephalalgia)

Stereotype	Tags	UML metamodel element
Ontologies		
Ontology		Package
OWLImport		Dependency
Annotations		
Comment		Comment
Label		Comment
ExplicitAnnotation	type	Comment
Entities		
OWLClass		Class
ClassType		Dependency
Datatype		Class, Property
Primitive		Class, Property
Data ranges		
DataOneOf		Connector
DataComplementOf		Class
Class axioms		
SubClassOf		Dependency
EquivalentClasses		Dependency, Connector
DisjointClasses		Dependency, Connector
DisjointUnion		Connector

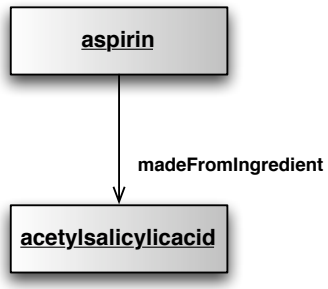


Figure 8.22: ObjectPropertyAssertion(madeFromIngredient aspirin acetylsalicylicacid)

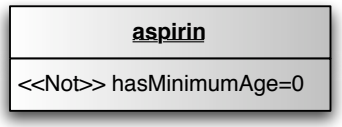


Figure 8.23: NegativeDataPropertyAssertion(hasMinimumAge aspirin 0)

Stereotype	Tags	UML metamodel elements
Class descriptions		
ObjectUnionOf		Connector
ObjectIntersectionOf		Connector
ObjectComplementOf		Dependency
ObjectOneOf		Connector
ObjectCardinality		Association
DataCardinality		Association, Property
ObjectSomeValuesFrom		Association
ObjectAllValuesFrom		Association
ObjectExistsSelf		Association
DataSomeValuesFrom		Association, Property
DataAllValuesFrom		Association, Property
DataHasValue		Property
Properties		
ObjectProperty		Class
DataProperty		Class
ObjectPropertyDomain		Dependency
ObjectPropertyRange		Dependency
DataPropertyDomain		Dependency
DataPropertyRange		Dependency
InverseObjectProperties		Dependency
Functional		«ObjectProperty» Class, «DataProperty» Class
InverseFunctional		«ObjectProperty» Class
Symmetric		«ObjectProperty» Class
Antisymmetric		«ObjectProperty» Class
Transitive		«ObjectProperty» Class
Reflexive		«ObjectProperty» Class

2006–2007 © Copyright lies with the respective authors and their institutions.

Stereotype	Tags	UML metamodel elements
Irreflexive		«ObjectProperty» Class
SubObjectProperty		Dependency
SubDataProperty		Dependency
EquivalentObjectProperties		Dependency, Connector
EquivalentDataProperties		Dependency, Connector
DisjointObjectProperties		Dependency, Connector
DisjointDataProperties		Dependency, Connector
Facts		
SameIndividual		Connector, Connector
DifferentIndividual		Connector, Connector
Not		Association, Property

8.2 A UML Profile Extension for Rules

Most of the notations used in the profile for rules exist already in the profile for OWL. We introduce the profile in the order we used when discussing the SWRL metamodel in Section 5.1 starting on page 49.

8.2.1 UML Syntax for Rules

Figure 8.24 shows an example of a rule defining that a pharmacy that is a client of a certain pharmaceutical lab can sell the medications produced by that lab. The figure demonstrates that for collecting the atoms of a rule, we reuse the package notation which is used in UML to represent collections of elements[2]. An appropriate stereotype «*Rule*» denotes the rule construct. An optional stereotype tag *URI* can specify the name of the rule. The complete rule is packed in one package, and stereotypes on the different atoms denote whether they belong to the antecedent or the consequent. Another possible notation for the rule construct would be to take two separate packages for antecedent and consequent, and connect them using a dependency. Because this would make even a very simple rule look very complex, we decided not to use this notation.

The figure shows that three variable definitions as well as three property assertions between these variables are defined in the example. Two of the property assertions build the antecedent of the rule, whereas the third one defines the consequent. We explain the specific design considerations of these concepts in the following subsections.

8.2.2 UML Syntax for Terms

Although the OWL profile already comprises a visual syntax for individuals and data values, namely by applying the UML object notation, it does not include a notation for variables as OWL ontologies do not contain variables. We depict variables in the UML object notation as well, since a variable can be seen as a partially unknown class instance. A stereotype «*Variable*» distinguishes a variable from an individual. Figure 8.25 shows an example for each type of term: a variable *x*, an individual *aspirin* belonging to the class *Medication*, and a data value *6* which is a *nonNegativeInteger*.

8.2.3 UML Syntax for Predicate Symbols in Atoms

Class description and data range

A visual notation for individuals as instances of certain classes is already provided in the profile for OWL. Exactly the same notation is used for rule atoms defining that a variable belongs to a certain class. In this

[2]Note that UML packages can be nested. Hence rules can be contained in the ontology package.

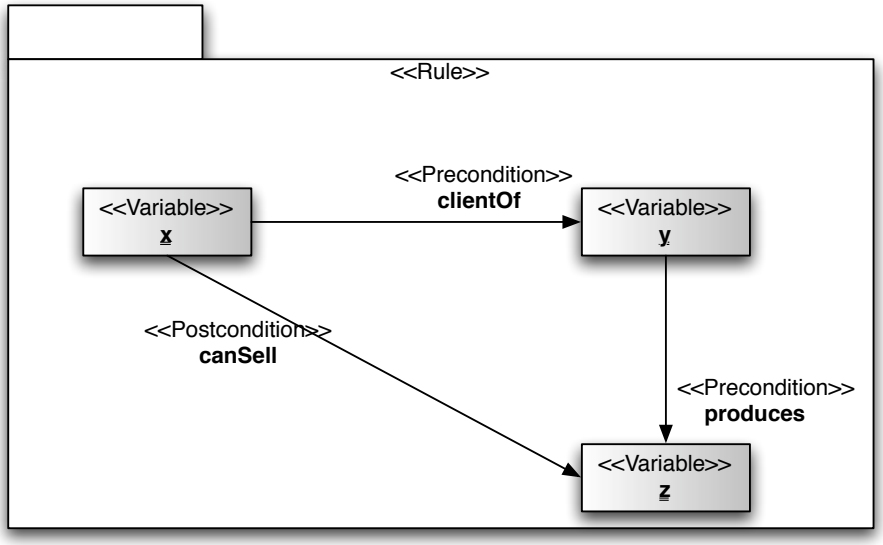


Figure 8.24: canSell$(x, z) \leftarrow$ clientOf$(x, y) \wedge$ produces(y, z)

(a) Variable (b) Individual (c) Data value

Figure 8.25: Rule terms

way, when the class is a simple class, the class name is added to the notation from Figure 8.25(a) like in the notation for individuals (Figure 8.25(b)). The example in Figure 8.26 contains several such constructs, for instance the definition of the variable x belonging to the class *Person*. The rule defines that when a person has at least the minimum age for the medication that helps against the disease he/she suffers, then the medication helps the person. Among others, the figure also contains a variable *y* defined as a *nonNegativeInteger*. Similar to the adaptation of the class assertion notation to support variables, the available notation for data ranges with individuals is adapted.

Properties

We depicted object property assertions as directed associations between the two involved elements. A datatype property can be pictured as an attribute or as an association. These notations were provided for individuals by the OWL profile, and we follow them to depict properties of variables. Figure 8.24 contains three such object properties between variables, *clientOf*, *produces* and *canSell*. The example rule depicted in Figure 8.26 contains amongst other things two datavalued properties *hasAge* and *hasMinimumAge*.

sameAs and differentFrom

According to the ontology profile, equality and inequality between objects are depicted using object relations. Again, because of the similarity between individuals and variables, we use the same visual notation for *sameIndividual* and *differentIndividuals* relations between variables or between a variable and an object.

2006–2007 © Copyright lies with the respective authors and their institutions.

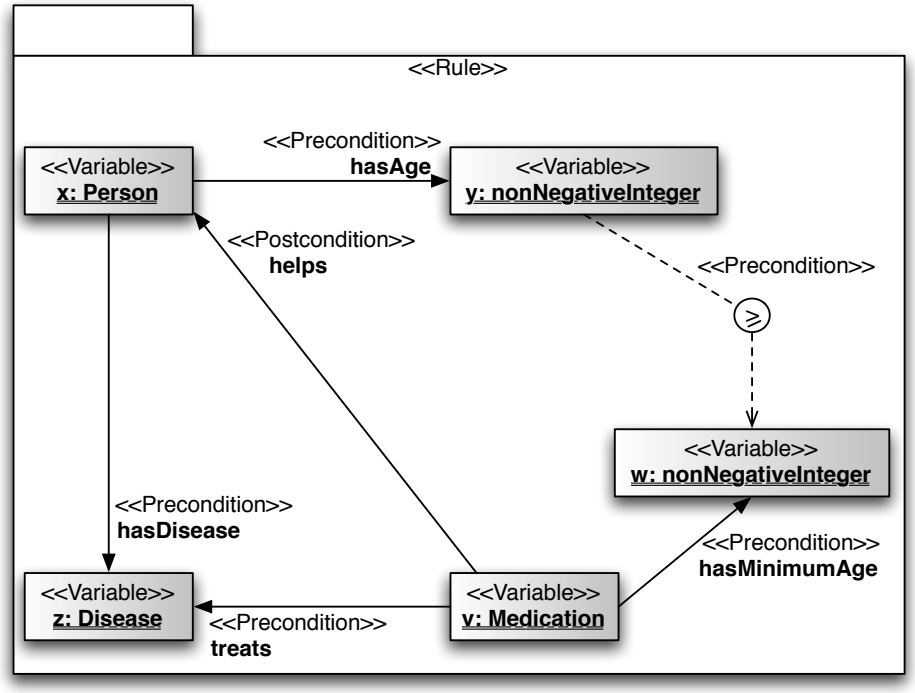


Figure 8.26: helps$(v, x) \leftarrow$ Person(x), nonNegativeInteger(y), hasAge(x, y), Disease(z), hasDisease(x, z), Medication(v), treats(v, z), nonNegativeInteger(w), hasMinimumage(v, w), swrlb:greaterThanOrEqual(y, w)

Built-in predicates

For the visual representation of built-in relations, one would want to use the notation with the dot or the icon with the dependencies as we introduced for quite some constructs in OWL for the sake of uniformity. However, because the order of the different elements in the built-in relation should be specified, dependencies are not suitable. In consequence of that, built-ins are represented in an object notation with the specific built-in ID and an appropriate stereotype. All participating variables and data values are connected through associations that have a number as name, to denote their order. Figure 8.27 shows an example of a built-in relation *swrlb:greaterThan*, which defines whether the first involved argument is greater than the second one. For some binary built-in predicates, a dependency with an appropriate icon can be used. The example rule of Figure 8.26 uses this alternative notation for the built-in predicate *greaterThanOrEqual*.

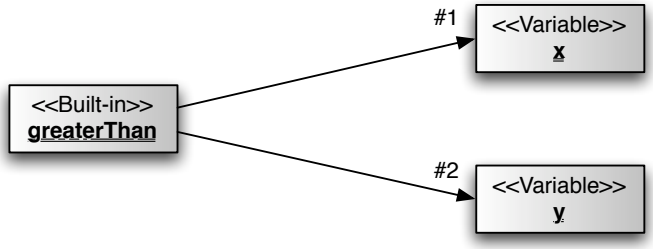


Figure 8.27: Built-in predicates

8.2.4 The Rule UML Profile

After introducing the different visual constructs using examples, the following table provides an overview of the SWRL extension of the OWL profile. For each stereotype, we specify possible tags and the UML metamodel element it is defined on. Note that the profile would not be used on its own as listed here, but always together with the profile for ontologies, which we listed in Section 8.1.6 starting on page 89.

Stereotype	Tags	UML metamodel element
Rule	name	Package
Precondition		Dependency, Class, InstanceSpecification, Association, Generalization, OWL profile-stereotyped Connector, Actor
Postcondition		Dependency, Class, InstanceSpecification, Association, Generalization, OWL profile-stereotyped Connector, Actor
Variable		InstanceSpecification
Built-in		InstanceSpecification

8.3 A UML Profile Extension for Ontology Mappings

As the last part of the profile, this section introduces the extension to support the visual definition of OWL ontology mappings. The metamodel for ontology mappings reuses elements of the metamodel for OWL and SWRL for defining the mappable elements. The metaclass *OWLEntity* is directly available in the OWL metamodel, whereas *OntologyQuery* is not but it is defined as consisting of elements which are all available in the SWRL metamodel. Similary, also the profile extension for ontology mappings reuses much from the profile for ontologies and rules. In fact, all mappable elements except queries are already available.

Figures 8.28 and 8.29 present examples of ontologies depicted using the UML profile for ontologies that we introduced in the previous sections. Based on these example ontologies, we now demonstrate and explain the UML notation for ontology mappings. Additionally, Section 8.3.3 on page 97 shows an overview of the profile extension.

8.3.1 UML Syntax for Mappings between Entities

When users want to define mapping assertions, they first specify the mapping between the two ontologies. Figure 8.30 presents the visual notation for a mapping between two ontologies. As ontologies are represented as packages, and dependencies are the only allowed means to connect UML packages, our construct builds on dependencies. As defined in the metamodel, mapping definitions between two ontologies do not only have a name but also several attributes. As a dependency could carry predefined attributes but no user-defined name, we define a mapping definition using a class with an appropriate stereotype and the specified attributes. Stereotyped dependencies connect the mapping definition to the two ontologies. Dependencies with the stereotype $<<$Mapping$>>$ link the mapping definition to the defined mappings. We explain the notation for mapping assertions into more detail in the following examples. Note that the packages around the elements in the examples give a rather complicated impression, but the reader should remember that these packages always represent a full ontology which we do not show in every mapping example.

Figure 8.31 shows the first example mapping assertion, that defines that the class *MedicalProduct* in the source ontology represents a more specific aspect of the world than the class *Product* in the target ontology. The semantic relation used in this example is the so-called *sound containment* relation. Note also that a

2006–2007 © Copyright lies with the respective authors and their institutions.

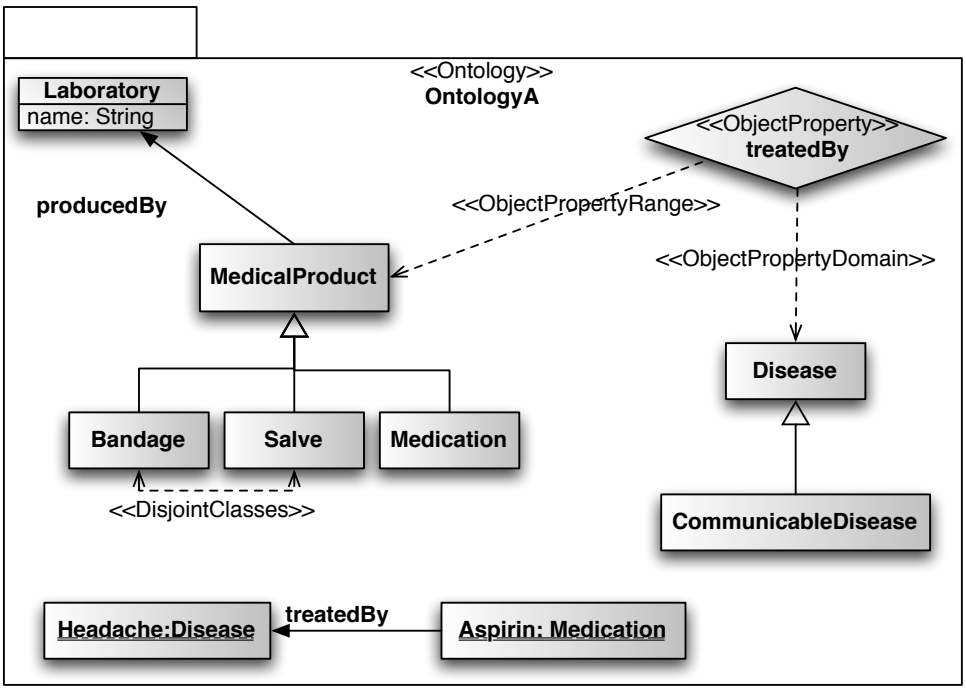


Figure 8.28: A first sample ontology depicted using the UML profile

notation with stereotypes can be used instead of using icons on the dependency. However, we present the icon-versions here.

The second example, depicted in Figure 8.32 relates two properties *treatedBy* and *isTreatableWith* using an *extensional equivalence* relationship. The attribute *extensional* denotes that the semantic relation is to be interpreted extensionally. For this specific mappings, it means that the set of objects, so the property assertions, of the source property *treatedBy* is exactly the same set as the set of property instantiations of the target property *isTreatableWith*. Note that the metamodel defines the default value of the attribute *interpretation* as 'intensional'. Hence, the previous examples of 8.31 is interpreted intensionally. When using the stereotype notation, the stereotypes $<<Intensional>>$ and $<<Extensional>>$ explicitly define the interpretation.

Figure 8.33 pictures a more complex example mapping assertion. The example defines that the union of the classes *PrescriptionMedication* and *NonPrescriptionMedication* in the target ontology, is equivalent to the class *Medication* in the source ontology. Although more classes are involved in the union of the target element, the mapping itself is defined on the anonymous class.

8.3.2 UML Syntax for Mappings between Queries

A mapping can be defined not only between usual ontology entities as in the previous examples, but also between queries. Figure 8.34 shows an example of an equivalence relation between two queries. A query is packed into a stereotyped package. Note that the package we show in the example, is the query package and not the ontology package as in the previous examples. The first query in the example contains a *MedicalProduct* X produced by a *Laboratory* Y named Z. The distinguished variables, which are the elements that are effectively being mapped, are denoted with an appropriate stereotype. The target query is about a *Product* X with *producer* Z. The mapping assertion defines an equivalence relation between the distinguished variables in the queries.

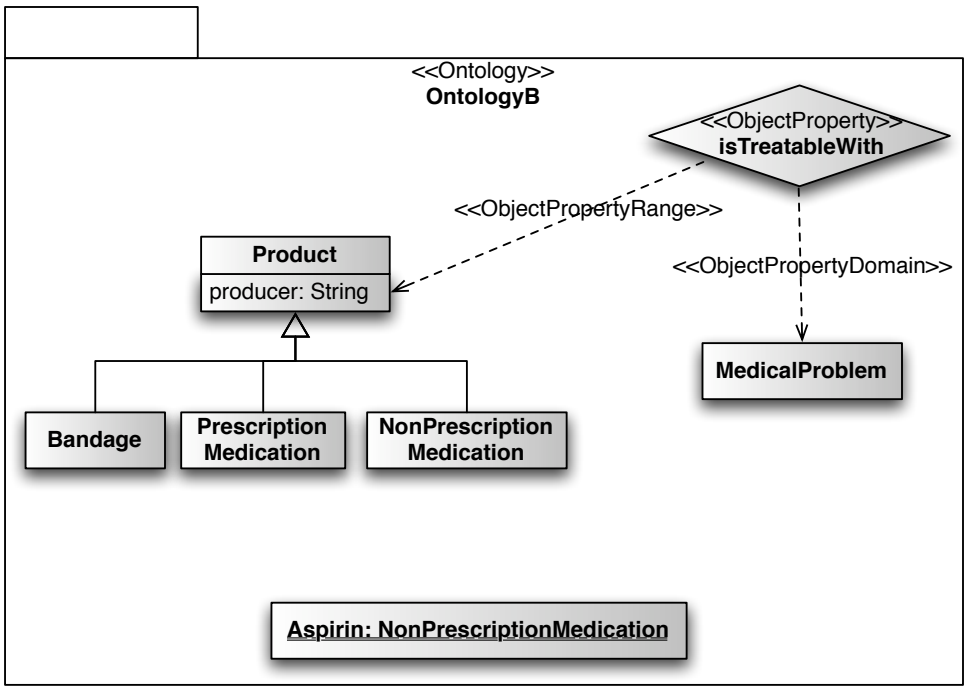


Figure 8.29: A second sample ontology depicted using the UML profile

8.3.3 The Ontology Mapping UML Profile

An overview of the profile for ontology mappings is provided in the following table. For each stereotype, we specify the UML metamodel element it is defined on[3]. Note that the profile would not be used on its own as listed here, but always together with the profile for ontologies and rules, as listed in Sections 8.1.6 starting on page 89 and 8.2.4 on page 95.

Stereotype	**UML metamodel element**
MappingDefinition	Class
SourceOntology	Dependency
TargetOntology	Dependency
Mapping	Dependency
SourceQuery	Package
TargetQuery	Package
Distinguished	«Variable» InstanceSpecification
SoundContainment	Dependency
CompleteContainment	Dependency
Equivalence	Dependency
Overlap	Dependency
NotSoundContainment	Dependency
NotCompleteContainment	Dependency
NotEquivalence	Dependency
NotOverlap	Dependency
Intensional	Stereotyped Dependency
Extensional	Stereotyped Dependency

[3]Note that this profile extension does not have any tags.

2006–2007 © Copyright lies with the respective authors and their institutions.

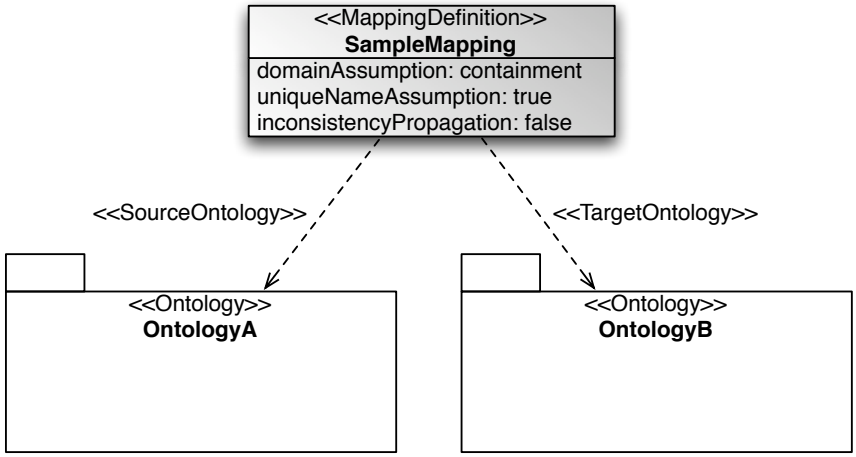


Figure 8.30: Sample mapping definition between two ontologies

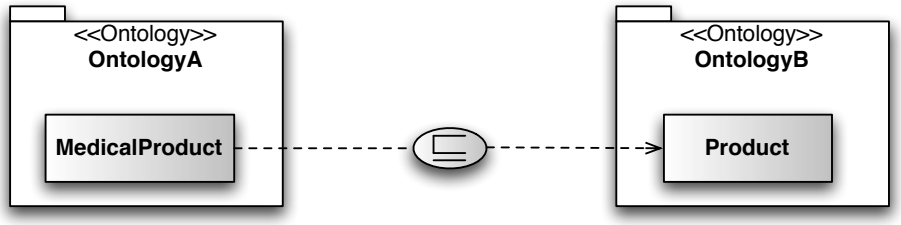


Figure 8.31: Sample sound containment relation between two concepts

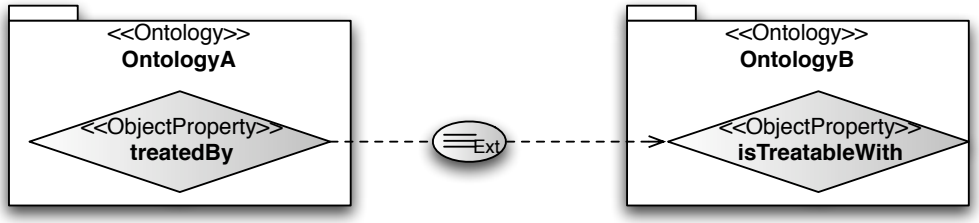


Figure 8.32: Sample extensional equivalence relation between two properties

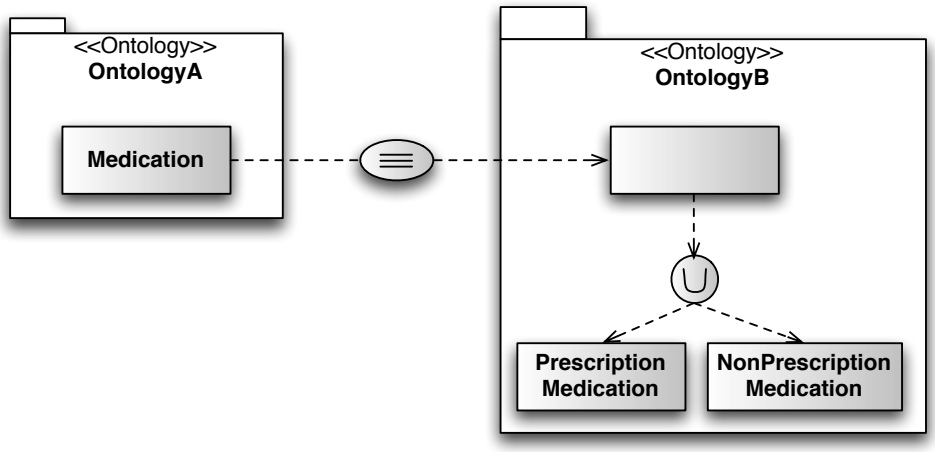


Figure 8.33: Sample equivalence relation between complex class descriptions

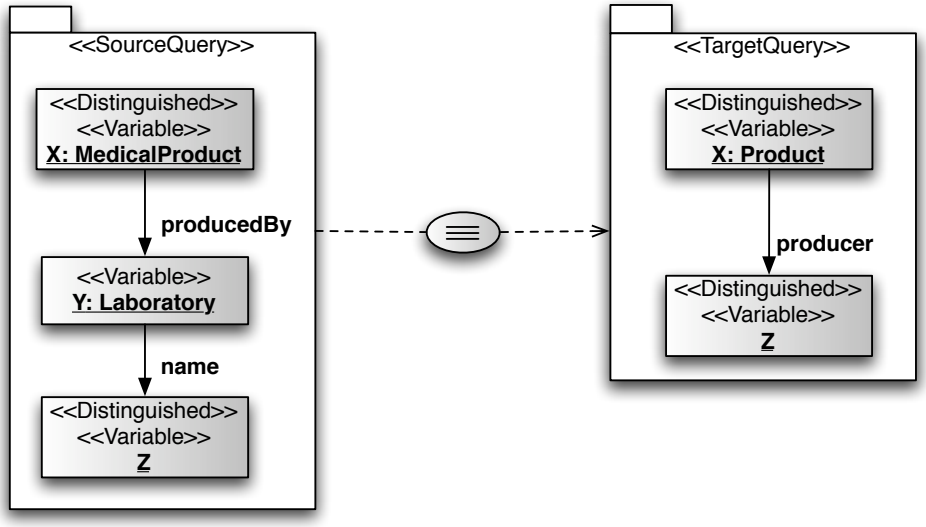


Figure 8.34: Sample equivalence relation between two queries

2006–2007 © Copyright lies with the respective authors and their institutions.

Part IV

Metadata for Networked Ontologies

Chapter 9

The NeOn Ontology Metadata Vocabulary

Ontologies have undergone an enormous development and have been applied in many domains within the last years, especially in the context of the Semantic Web. Currently however, efficient knowledge sharing and reuse, a pre-requisite for the realization of the Semantic Web vision, is a difficult task. It is hard to find and share existing ontologies because of the lack of standards for documenting and annotating ontologies with metadata information. Without ontology-specific metadata, developers are not able to reuse existing ontologies, which leads to problems of interoperability as well as duplicate efforts. Then, in order to provide a basis for an effective access and exchange of ontologies across the web, it is necessary to agree on a standard for ontology metadata. This standard then provides a common set of terms and definitions describing ontologies and is called metadata vocabulary.

In the remainder of this chapter we present a compact overview of our contribution to the alleviation of this situation: the ontology metadata standard OMV (**O**ntology **M**etadata **V**ocabulary), which specifies reusability-enhancing ontology features for human and machine processing purposes.

9.1 Preliminary considerations

9.1.1 Metadata Categories

OMV differentiates among the following three occurrence constraints for metadata elements—according to their impact on the prospected reusability of the described ontological content:

- **Required** – mandatory metadata elements. Any missing entry in this category leads to an incomplete description of the ontology.
- **Optional** – important metadata facts, but not strongly required.
- **Extensional** – specialized metadata entities, which are not considered to be part of the core metadata scheme.

Complementary to this classification we organize the metadata elements, according to the type and purpose of the contained information, as follows:

- **General** – elements providing general information about the ontology.
- **Availability** – information about the location of the ontology (e.g. its URI or URL where the ontology is published on the Web)
- **Applicability** – information about the intended usage or scope of the ontology.
- **Format** – information about the physical representation of the resource. In terms of ontologies, these elements include information about the representation language(s) in which the ontology is formalized.

2006–2007 © Copyright lies with the respective authors and their institutions.

- **Provenance** – information about the organizations contributing to the creation of the ontology.
- **Relationship** – information about relationships to other resources. This category includes versioning, as well as conceptual relationships such as extensions, generalization/specialization and imports.
- **Statistics** - various metrics on the underlying graph topology of an ontology (e.g. number of classes)
- **Other** - information not covered in the categories listed above.

Note that the introduced classification dimensions are not intended to be part of the metadata scheme itself, but will be taken into consideration by the implementation of several metadata support facilities. The first dimension is relevant for a metadata creation service in order to ensure a minimal set of useful metadata entries for each of the described resources. The second can be used in various settings mainly to reduce the user-perceived complexity of the metadata scheme whose elements can be structured according to the corresponding classes.

9.2 Ontology Metadata Requirements

We elaborated an inventory of requirements for the metadata model as a result of a systematic survey of the state of the art in the area of ontology reuse. Besides a scientific analysis, we conducted extensive literature research, which focused on theoretical methods [PM01, GPS99, LTGP04], but also on case studies on reusing existing ontologies [UHW+98, RVMS99, PBMT05], in order to identify real-world needs of the community w.r.t. a descriptive metadata format for ontologies. Further on, the requirements analysis phase was complemented by a comparative study of existing (ontology-independent) metadata models and of tools such as ontology repositories and libraries (implicitly) making use of metadata-like information. Several aspects are similar to other metadata standards such as Dublin Core. Differences arise however if we consider the semantic nature of ontologies, which are much more than plain Web information sources. In accordance to one of the major principles in Ontological Engineering, an ontology comprises a conceptual model of a particular domain of interest, represented at the knowledge level, and multiple implementations using knowledge representation languages. These two components are characterized by different properties, can be developed and maintained separately. The main requirements identified in this process step are the following:

Accessibility: Metadata should be accessible and processable for both humans and machines. While the human-driven aspects are ensured by the usage of natural language concept names, the machine-readability requirement can be implemented by the usage of Web-compatible representation languages (such as XML or Semantic Web languages, see below).

Usability: It is important to build a metadata model which 1) reflects the needs of the majority of ontology users—as reported by current case studies in ontology reuse—but at the same time 2) allows proprietary extensions and refinements in particular application scenarios. From a content perspective, usability can be maximized by taking into account multiple metadata types, which correspond to specific viewpoints on the ontological resources and are applied in various application tasks. Despite the broad understanding of the metadata concept and the use cases associated to each definition, several key aspects of metadata information have already been established across computer science fields [Org04]:

- **Structural metadata** relates to statistical measures on the graph structure which underlies an ontology. In particular we mention the number of specific ontological primitives (e.g. number of classes or instances). The availability of structural metadata influences the usability of an ontology in a concrete application scenario, as size and structure parameters constraint the type of tools and methods which can be applied to aid the reuse process.

- **Descriptive metadata** relates to the domain modeled in the ontology in form of keywords, topic classifications, textual descriptions of the ontology contents etc. This type of metadata plays a crucial role in the selection of appropriate reuse candidates, a process which includes requirements w.r.t. the domain of the ontologies to be re-used.
- **Administrative metadata** provides information to help manage ontologies, such as when and how it was created, rights management, file format and other technical information.

Interoperability: Similarly to the ontology it describes, metadata information should be available in a form which facilitates metadata exchange among applications. While the syntactical aspects of interoperability are covered by the usage of standard representation languages (see "Accessibility"), the semantical interoperability among machines handling ontology metadata information can be ensured by means of an formal and explicit representation of the meaning of the metadata entities, i.e. by conceptualizing the metadata vocabulary itself as an ontology.

9.3 Ontology Metadata Vocabulary

This section gives an overview of the core design principles applied for the realization of the OMV metadata scheme, which is described in detail later in this section.

9.3.1 Core and Extensions

Following the usability constraints identifies during the requirements analysis, we decided to design the OMV scheme modularly; OMV distinguishes between the OMV Core and various OMV Extensions. The former captures information which is expected to be relevant to the majority of ontology reuse settings. However, in order to allow ontology developers and users to specify task- or application-specific ontology-related information we foresee the development of OMV extension modules, which are physically separated from the core scheme, while remaining compatible to its elements.

9.3.2 Ontological representation

Due to the high accessibility and interoperability requirements, as well as the nature of the metadata, which is intended to describe Semantic Web ontologies, the conceptual model designed in the previous step was implemented in the OWL language. An implementation as XML-Schema or DTD was estimated to restrict the functionality of the ontology management tools using the metadata information (mainly in terms of retrieval capabilities) and to impede metadata exchange at semantical level. Further on, a language such as RDFS does not provide a means to distinguish between required and optional metadata properties. The implementation was performed manually by means of a common ontology editor.

9.3.3 Identification, Versioning and Location

An important issue that has to be addressed when describing ontologies is the ability to identify and manage multiple versions and physical representations of one ontology. The OWL ontology language itself does not provide the means to address this issue: In OWL, an ontology is identified by a URI. Here it is important to note, that this URI is merely a logical identifier, it does not (necessarily) relate to the physical location of the ontology (e.g. in a file), nor does it prescribe a versioning scheme.

Versioning OWL does not distinguish between the notion of an ontology and a version of an ontology at all. It may thus be that different versions of ontology carry the same logical URI.

2006–2007 © Copyright lies with the respective authors and their institutions.

In general, a version is a variant of an ontology that is usually created after applying changes to an existing variant. Therefore we need a way to unambiguously identify the different versions as well as to keep track of the relationships between them. Based on [KF01], we consider that changes in ontologies are caused by: (i) changes in the domain; (ii) changes in the shared conceptualization; (iii) changes in the specification. Taking the definition of an ontology as a specification of a conceptualization, (i) and (ii) are semantic changes that lead to the creation of a new conceptualization, while (iii) is just a change in the representation of the same conceptualization (also known as a new revision) (e.g. updates of natural language descriptions of ontology elements). In any case, the change(s) result in a different physical representation of the ontology (i.e. different version). Consequently, it should be possible to identify each of those versions. Normally an ontology is identified by an URI, which according to [BLFM98] is a compact string of characters for identifying an abstract or physical resource. In [KF01] the authors propose that any version that constitutes a new conceptualization (i.e. changes of type (i) and (ii)) should have a unique URI, however in practice different versions of same ontology might share the same URI. Furthermore, even if a revision constitutes the same conceptualization of an ontology it is physically represented in a different file which might have additional metadata (e.g. updated ontology description, descriptions in different natural languages, different file location, etc.).

In OMV we describes a particular representation of an ontology, i.e. an ontology in a particular version at a particular physical location. That means that every different version of an ontology has a different OMV related metadata and consequently, a particular OMV instance is identified by the URI plus the version of the ontology it is describing.

Resource Location An addition to the issue of versioning, an ontology (or a version of an ontology) can be located at different locations. Thus, ontologies with the same logical URI may exist at different physical locations, possibly even with different content.

Similar to approach for versioning, we rely on a composite identifier consisting of the logical identifier (URI plus optional version identifier) and a resource locator that specifies the actual physical location.

Of course, the optional version identifier and the optional resource locator can be combined, such that we end up with a tripartite identifier (URI, version, resource locator).

Illustrative Example In order to clarify the previous discussion consider the following scenario: Initially, we have the first implementation of ontology OWLODM (i.e. `http://owlodm.ontoware.org/OWL1.0`) which provides a metamodel for the ontology language OWL 1.0. A fragment of the OMV description for OWLODM version 1.0 is the following:

```
<omv:Ontology rdf:about="&j;OWL1.0v1.0">
    <omv:URI rdf:datatype="&xsd;string">http://owlodm.ontoware.org/OWL1.0</omv:URI>
    <omv:version rdf:datatype="&xsd;string">1.0</omv:version>
    <omv:resourceLocator rdf:datatype="&xsd;string">
    http://ontoware.org/frs/download.php/307/owl10.owl</omv:resourceLocator>
    <omv:acronym rdf:datatype="&xsd;string">OWLODM</omv:acronym>
    <omv:description rdf:datatype="&xsd;string">OWL Object Definition Metamodel
    (ODM) allows interoperability of OWL ontologies with MOF-compatible
    software environments</omv:description>
    <omv:name rdf:datatype="&xsd;string">OWL Ontology Definition Metamodel</omv:name>
    <omv:numberOfClasses rdf:datatype="&xsd;unsignedInt">35</omv:numberOfClasses>
    <omv:numberOfProperties rdf:datatype="&xsd;unsignedInt">22</omv:numberOfProperties>
    <omv:hasCreator rdf:resource="#PeterHaase"/>
    <omv:hasDomain rdf:resource="&c;Knowledge Representation"/>
    <omv:creationDate rdf:datatype="&xsd;string">2007-02-12</omv:creationDate>
    ...
</omv:Ontology>
```

A change in the domain modelled by OWLODM (i.e. the definition of OWL 1.1) was reflected in a new *version* of the OWLODM ontology, namely version 1.1. This change leaded to a semantic change of the ontology (i.e. change of type (i)), and therefore a new URI was defined for OWLODM (i.e. `http://owlodm.ontoware.org/OWL1.1`). A fragment of the OMV description for OWLODM version 1.1 is the following:

```
<omv:Ontology rdf:about="&j;OWL1.1v1.1">
    <omv:URI rdf:datatype="&xsd;string">http://owlodm.ontoware.org/OWL1.1</omv:URI>
    <omv:version rdf:datatype="&xsd;string">1.1</omv:version>
    <omv:resourceLocator rdf:datatype="&xsd;string">
    http://ontoware.org/frs/download.php/365/owl11.owl</omv:resourceLocator>
    <omv:acronym rdf:datatype="&xsd;string">OWLODM</omv:acronym>
```

```
    <omv:description rdf:datatype="&xsd;string">OWL Object Definition Metamodel
    (ODM) allows interoperability of OWL ontologies with MOF-compatible
    software environments</omv:description>
    <omv:name rdf:datatype="&xsd;string">OWL Ontology Definition Metamodel</omv:name>
    <omv:numberOfClasses rdf:datatype="&xsd;unsignedInt">76</omv:numberOfClasses>
    <omv:numberOfProperties rdf:datatype="&xsd;unsignedInt">35</omv:numberOfProperties>
    <omv:hasCreator rdf:resource="#PeterHaase"/>
    <omv:hasDomain rdf:resource="&c;Knowledge Representation"/>
    <omv:creationDate rdf:datatype="&xsd;string">2007-08-09</omv:creationDate>
    ...
</omv:Ontology>
```

Finally, a new version of OWLODM (i.e. version 1.2) was released as a result of a refinement. In this case, the change was at the level of the specification of the ontology (i.e. change type (iii)), in particular the renaming of a property and hence the URI was not updated. A fragment of the OMV description for the OWLODM version 1.2 is the following:

```
<omv:Ontology rdf:about="&j;OWL1.1v1.2">
    <omv:URI rdf:datatype="&xsd;string">http://owlodm.ontoware.org/OWL1.1</omv:URI>
    <omv:version rdf:datatype="&xsd;string">1.2</omv:version>
    <omv:resourceLocator rdf:datatype="&xsd;string">
    http://ontoware.org/frs/download.php/366/owl11.owl</omv:resourceLocator>
    <omv:acronym rdf:datatype="&xsd;string">OWLODM</omv:acronym>
    <omv:description rdf:datatype="&xsd;string">OWL Object Definition Metamodel
    (ODM) allows interoperability of OWL ontologies with MOF-compatible
    software environments</omv:description>
    <omv:name rdf:datatype="&xsd;string">OWL Ontology Definition Metamodel</omv:name>
    <omv:numberOfClasses rdf:datatype="&xsd;unsignedInt">76</omv:numberOfClasses>
    <omv:numberOfProperties rdf:datatype="&xsd;unsignedInt">35</omv:numberOfProperties>
    <omv:hasCreator rdf:resource="#PeterHaase"/>
    <omv:hasDomain rdf:resource="&c;Knowledge Representation"/>
    <omv:creationDate rdf:datatype="&xsd;string">2007-08-10</omv:creationDate>
    ...
</omv:Ontology>
```

As we can see from the previous simple example, the URI is not enough to identify individually each version of the ontology. Besides, in practice not every semantic change leads to the definition of a new URI (as in this example). Even more, in this example it was enough to use the URI plus the version to identify each physical implementation, however it could also be possible that the same ontology version is located at two (or more) different physical location, where each of them could have even different content as we anticipated in the previous section. In that case we will also need to use the location of the ontology to identify a particular implementation (i.e. URI plus version plus location).

9.3.4 OMV core metadata entities

The main classes and properties of the OMV ontology are illustrated in Figure 9.1. [1]

Additionally to the main class `Ontology` the metadata scheme contains further elements describing various aspects related to the creation, management and usage of an ontology. We will briefly discuss these in the following. In a typical ontology engineering process `Person`s or `Organisation`(s) are developing ontologies. We group these two classes under the generic class `Party` by a `subclass-of` relation. A `Party` can have several locations by referring to a `Location` individual and can *create*, *contribute* to ontological resources i.e. `Ontology Implementation`s. Review details and further information can be captured in an extensional OMV module (see Chapter [HP05]). Further on we provide information about the engineering process the ontology originally resulted from in terms of the classes `OntologyEngineeringMethodology`, `OntologyEngineeringTool` and the attributes `version`, `status`, `creationDate` and `modificationDate`. Again these can be elaborated as an extension of the core metadata scheme. The usage history of the ontology is modelled by classes such as the `OntologyTask` and `LicenceModel`. The scheme also contains a representation of the most significant intrinsic features of an ontology. Details on ontology languages are representable with the help of the classes `OntologySyntax`, `OntologyLanguage` and `KnowledgeRepresentationParadigm`. Ontologies might be categorized along a multitude of dimensions. One of the most popular classification differentiates among `application`, `domain`, `core`, `task` and `upper-level` ontologies. A further classi-

[1] Please notice, that not all classes and properties are included. The ontology is available for download in several ontology formats at `http://omv.ontoware.org/`

2006–2007 © Copyright lies with the respective authors and their institutions.

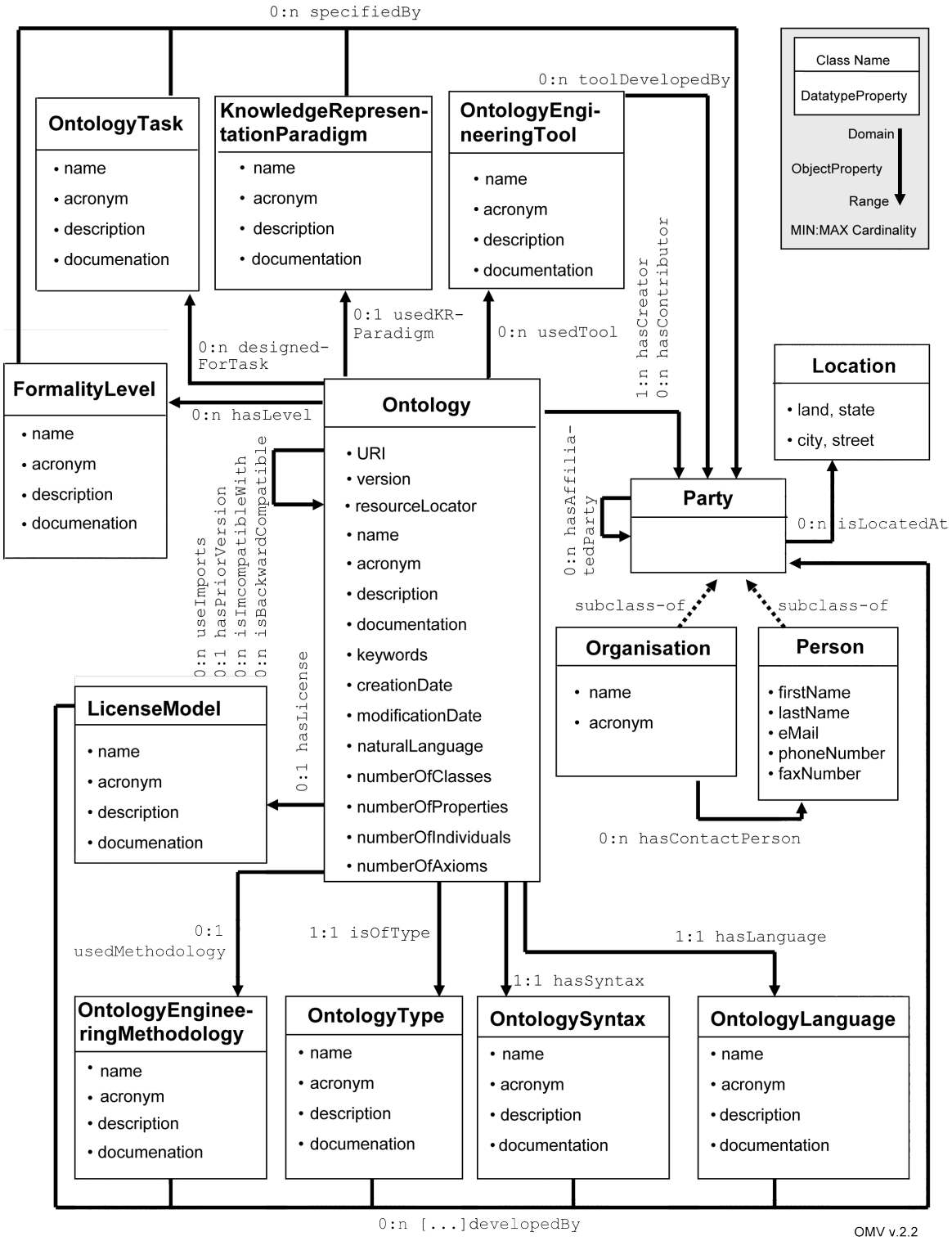


Figure 9.1: OMV overview

fication relies on their level of formality and types of Knowledge Representation (KR) primitives supported, introducing catalogues, glossaries, thesauri, taxonomies, frames etc. as types of ontologies. These can be modeled as instances of the class `OntologyType`, while generic formality levels are introduced with the help of the class `FormalityLevel`. The domain the ontology describes is represented by the class `OntologyDomain` referencing a pre-defined topic hierarchy such as the DMOZ hierarchy. Further content information can be provided as values of the DatatypeProperties `description`, `keywords`, and `documentation`. Finally the metadata scheme gives an overview of the graph topology of an `Ontology` with the help of several graph-related metrics represented as integer values of the DatatypeProperties `numberOfClasses`, `numberOfProperties`, `numberOfAxioms`, `numberOfIndividuals`.

In the appendix of this deliverable we provide a detailed description of the OMV core model. For additional information please refer to [HP05].

2006–2007 © Copyright lies with the respective authors and their institutions.

Chapter 10

Conclusion

10.1 Summary

Next generation semantic applications will be characterized by a large number of networked ontologies; as a consequence the complexity of semantic applications increases. In the NeOn project we address this challenge by creating an open infrastructure, and associated methodology, to support the development life-cycle of such a new generation of semantic applications. In this deliverable we have presented an updated version of our metamodel for the specification of networked ontologies that will serve as a foundation for this infrastructure. We have followed the metamodeling approach of Model Driven Architectures for the specification of the metamodel. Specifically, we have presented four modules of the metamodel:

1. The OWL 1.1 metamodel serves as the core of our networked ontology model,
2. the rule metamodel covering both SWRL and F-Logic,
3. the mapping metamodel allows for the specification of ontology mappings that describe correspondences between ontology elements in a network of ontologies, and
4. the metamodel for modular ontologies builds on the OWL and mapping metamodel to model ontologies as reusable components in a network of ontologies.

Further, in this deliverable we have presented the current version of OMV as an Ontology Metadata Vocabulary, which aims at facilitating the task of sharing, searching, reusing and managing the relationships between ontologies.

We expect that our models presented in this deliverable will not immediately deprecate our initial models presented in D1.1.1. Instead, to a large extend our new models complement our initial models. In particular, we expect that the OWL 1.0 will co-exist with our OWL 1.1 metamodel until OWL 1.1 is accepted as a standard and fully supported in the tool infrastructure.

Appendix A

Naming Conventions

Choosing a naming convention for ontology modeling and adhere to these conventions makes the ontology easier to understand and helps to avoid some common modeling mistakes. Therefore, for the modeling of ontologies in the context of the NeOn project, we adopted the following set of conventions for ontology elements (i.e.classes, properties and instances)

A.1 Delimiters and capitalization

- **Class Names** - Class names are capitalized. If the class name contains more than one word, we use concatenated words and capitalize each new word. I.e. "Ontology" "OntologySyntax"
- **Property Names** - Property names use lower case. If the property name contains more than one word, we use concatenated words where the first word is all in lower case and capitalize each subsequent new word. I.e. "name" "naturalLanguage" "hasLicense"
- **Instance Names** - Instance names use lower case. If the instance name contains more than one word, we use concatenated words where the first word is all in lower case and capitalize each subsequent new word. I.e. "peter".

A.2 Property naming

The use of property naming conventions helps to identify in an easy way ontology elements. NeOn uses specific prefixes to identify **DatatypeProperty** and **ObjectProperty**. Thus, the ObjectProperties start with a verb specifying how the two classes are related to each other. I.e. "specifiedBy" "usedOntologyEngineeringTool" "hasOntologySyntax".

A.3 Singular form

The convention adopted by NeOn was to use names for classes, properties and instances in singular form. The decision was based on the fact that singular form is used more often in practice in many domains. Besides, when working with XML, for example, importing legacy XML or generating XML feeds from the ontology, it is necessary to make sure to use a singular form since this is expected convention for XML tags.

A.4 Additional considerations

- When a word within a name is all capitals, the next word should start in lower case. An hypothetical example: "URLoriginal"

2006–2007 © Copyright lies with the respective authors and their institutions.

- Do not add strings such as "class" or "attribute", and so on to the names.
- Do not concatenate the name of the class to the properties or instances, i.e. there is no "ontologyName" "ontologySyantxName"
- Try to avoid abbreviations on the names of the ontology elements. The use of abbreviations in the names can lead to names difficult to understand, therefore unless the names are self explanatory, we will avoid them.

Appendix B

OMV Core Ontology

In the following we introduce the metadata elements of OMV, the first metadata standard for ontologies. As aforementioned, OMV is formalized as an OWL ontology. A metadata element is modelled either by means of classes and individuals or by means of valued properties. The decision for one of these two alternatives was justified by the complexity of the corresponding metadata element. If the value/content of a metadata element can be easily mapped to conventional data types (numerical, literal, list values) the metadata element is usually represented as a DatatypeProperty. Complex metadata elements which do not fall into the previous category are modelled by means of additional classes linked by ObjectProperties.

The description of the model is grouped along the core classes of the ontology. For each class we describe the meaning of its properties and additional usage and occurrence constraints.

2006–2007 © Copyright lies with the respective authors and their institutions.

B.1 Ontology

Aspects of specific realizations are covered modular (and extendable) by the class `Ontology`.

Ontology	
Name Type Identifier	Ontology `class`
Definition	An implementation of a conceptual model
OMV version	0.1
Comments	None

Table B.1: Class: Ontology

URI	
Name Type Identifier	URI `DatatypeProperty`
Occurrence Constraint Category	`required` General information
Definition	The URI of the ontology which is described by this metadata. It serves as a logical identifer and is not necessarily the physical location

Domain
Range
Cardinality
	`omv:Ontology`
	`xsd:string`
	1:1

OMV version	0.1
Comments	None

Table B.2: Property: URI

name	
Name Type Identifier	name `DatatypeProperty`
Occurrence Constraint Category	`required` General information
Definition	The name by which an ontology is formally known

Domain
Range
Cardinality
	`omv:Ontology`
	`xsd:string`
	1:n

OMV version	0.1
Comments	The ontology can have many names (e.g. names in different languages)

Table B.3: Property: name

acronym	
Name Type Identifier	acronym `DatatypeProperty`
Occurrence Constraint Category	`optional` General information
Definition	A short name by which an ontology is formally known

Domain
Range
Cardinality
	`omv:Ontology`
	`xsd:string`
	1:1

OMV version	0.1
Comments	None

Table B.4: Property: acronym

description	
Name Type Identifier	description DatatypeProperty
Occurrence Constraint Category	required General information
Definition	Free text description of an ontology
Domain Range Cardinality	omv:Ontology xsd:string 1:1
OMV version	0.1
Comments	None

Table B.5: Property: description

documentation	
Name Type Identifier	documentation DatatypeProperty
Occurrence Constraint Category	optional General information
Definition	URL for further documentation
Domain Range Cardinality	omv:Ontology xsd:string 0:1
OMV version	0.2
Comments	None

Table B.6: documentation

notes	
Name Type Identifier	notes DatatypeProperty
Occurrence Constraint Category	optional General information
Definition	Describes additional information about the ontology that is not included somewhere else (e.g. information that you do not want to include in the documentation)
Domain Range Cardinality	omv:Ontology xsd:string 0:1
OMV version	2.2
Comments	None

Table B.7: notes

keywords	
Name Type Identifier	keywords DatatypeProperty
Occurrence Constraint Category	optional General information
Definition	List of keywords related to an ontology
Domain Range Cardinality	omv:Ontology xsd:string 0:n
OMV version	0.1
Comments	Typically this set includes words that describe the content of the ontology

Table B.8: Property: keywords

2006–2007 © Copyright lies with the respective authors and their institutions.

keyClasses	
Name Type Identifier	keyClasses DatatypeProperty
Occurrence Constraint Category	optional General information
Definition	Indicates what the central/best represented classes of the ontology are
Domain Range Cardinality	omv:Ontology xsd:string 0:n
OMV version	2.2
Comments	none

Table B.9: Property: keyClasses

status	
Name Type Identifier	status DatatypeProperty
Occurrence Constraint Category	optional General information
Definition	It specifies the tracking information for the contents of the ontology
Domain Range Cardinality	omv:Ontology xsd:string 0:1
OMV version	0.1
Comments	Pre-defined values

Table B.10: Property: status

creationDate	
Name Type Identifier	creationDate DatatypeProperty
Occurrence Constraint Category	required General information
Definition	Date when the ontology was initially created
Domain Range Cardinality	omv:Ontology xsd:date 1:1
OMV version	0.1
Comments	None

Table B.11: Property: creationDate

modificationDate	
Name Type Identifier	modifiedDate DatatypeProperty
Occurrence Constraint Category	optional General information
Definition	Date of the last modification made to the ontology
Domain Range Cardinality	omv:Ontology xsd:date 0:1
OMV version	0.1
Comments	None

Table B.12: Property: modificationDate

hasContributor	
Name	hasContributor
Type	ObjectProperty
Identifier	
Occurrence Constraint	optional
Category	Provenance information
Definition	Contributor to the ontology
Domain	omv:Ontology
Range	omv:Party
Cardinality	0:n
OMV version	0.1
Comments	None

Table B.13: Property: hasContributor

hasCreator	
Name	hasCreator
Type	ObjectProperty
Identifier	
Occurrence Constraint	required
Category	Provenance information
Definition	Main responsible for the ontology
Domain	omv:Ontology
Range	omv:Party
Cardinality	1:n
OMV version	0.1
Comments	None

Table B.14: Property: hasCreator

usedOntologyEngineeringTool	
Name	usedOntologyEngineeringTool
Type	ObjectProperty
Identifier	
Occurrence Constraint	optional
Category	Provenance information
Definition	Information about tool used to create the ontology
Domain	omv:Ontology
Range	omv:OntologyEngineeringTool
Cardinality	0:n
OMV version	0.1
Comments	None

Table B.15: Property: usedOntologyEngineeringTool

usedOntologyEngineeringMethodology	
Name	usedOntologyEngineeringMethodology
Type	ObjectProperty
Identifier	
Occurrence Constraint	optional
Category	Provenance information
Definition	Information about the method model used to create the ontology
Domain	omv:Ontology
Range	omv:OntologyEngineeringMethodology
Cardinality	0:n
OMV version	0.1
Comments	None

Table B.16: Property: usedOntologyEngineeringMethodology

2006–2007 © Copyright lies with the respective authors and their institutions.

usedKnowledgeRepresentationParadigm	
Name Type Identifier	usedKnowledgeRepresentationParadigm ObjectProperty
Occurrence Constraint Category	optional Provenance information
Definition	Information about the paradigm model used to create the ontology
Domain Range Cardinality	omv:Ontology omv:KnowledgeRepresentationParadigm 0:n
OMV version	0.1
Comments	None

Table B.17: Property: usedKnowledgeRepresentationParadigm

hasDomain	
Name Type Identifier	hasDomain ObjectProperty
Occurrence Constraint Category	optional Applicability information
Definition	Specifies the domain topic of an ontology
Domain Range Cardinality	omv:Ontology omv:OntologyDomain 0:n
OMV version	0.8
Comments	Typically, the domain can refer to established topic hierarchies such as the general purpose topic hierarchy DMOZ or the domain specific topic hierarchy ACM for the computer science domain

Table B.18: Property: hasDomain

isOfType	
Name Type Identifier	isOfType ObjectProperty
Occurrence Constraint Category	required Applicability information
Definition	The nature of the content of the ontology
Domain Range Cardinality	omv:Ontology omv:OntologyType 1:1
OMV version	0.1
Comments	Pre-defined values. See section B.2 for details

Table B.19: Property: isOfType

naturalLanguage	
Name Type Identifier	naturalLanguage DatatypeProperty
Occurrence Constraint Category	optional Applicability information
Definition	The language of the content of the ontology, i.e. English, German, etc.
Domain Range Cardinality	omv:Ontology xsd:string 0:n
OMV version	0.1
Comments	Pre-defined values

Table B.20: Property: naturalLanguage

designedForOntologyTask	
Name Type Identifier	designedForOntologyTask ObjectProperty
Occurrence Constraint Category	optional Applicability information
Definition	Declares for which purpose the ontology was originally designed
Domain Range Cardinality	omv:Ontology omv:OntologyTask 0:n
OMV version	0.9
Comments	See Section B.10

Table B.21: Property: designedForOntologyTask

hasFormalityLevel	
Name Type Identifier	hasFormalityLevel ObjectProperty
Occurrence Constraint Category	optional Applicability information
Definition	Level of formality of the ontology
Domain Range Cardinality	omv:Ontology omv:FormalityLevel 0:1
OMV version	0.9.1
Comments	Pre-defined values

Table B.22: Property: hasFormalityLevel

knownUsage	
Name Type Identifier	knownUsage DatatypeProperty
Occurrence Constraint Category	optional Applicability information
Definition	The applications where the ontology is being used
Domain Range Cardinality	omv:Ontology xsd:string 0:n
OMV version	2.2
Comments	None

Table B.23: Property: knownUsage

hasOntologyLanguage	
Name Type Identifier	hasOntologyLanguage ObjectProperty
Occurrence Constraint Category	required Format information
Definition	Specifies the ontology language
Domain Range Cardinality	omv:Ontology omv:OntologyLanguage 1:1
OMV version	0.1
Comments	Pre-defined values

Table B.24: Property: hasOntologyLanguage

2006–2007 © Copyright lies with the respective authors and their institutions.

hasOntologySyntax	
Name Type Identifier	hasOntologySyntax ObjectProperty
Occurrence Constraint Category	required Format information
Definition	It specifies the presentation syntax for the ontology language
Domain Range Cardinality	omv:Ontology omv:OntologySyntax 1:1
OMV version	0.1
Comments	Pre-defined values

Table B.25: Property: hasOntologySyntax

consistencyAccordingToReasoner	
Name Type Identifier	consistencyAccordingToReasoner DatatypeProperty
Occurrence Constraint Category	optional Format information
Definition	Indicates whether a reasoner has classified the ontology correctly
Domain Range Cardinality	omv:Ontology xsd:boolean 0:1
OMV version	2.2
Comments	The definition of consistency is independent of a reasoner. The assumption is that the reasoner used for consistency checking operates correctly, i.e. according to the well-defined semantics of the ontology language

Table B.26: Property: consistencyAccordingToReasoner

resourceLocator	
Name Type Identifier	resourceLocator DatatypeProperty
Occurrence Constraint Category	required Availability information
Definition	The location where the ontology can be found. It should be accessible via a URL
Domain Range Cardinality	omv:Ontology xsd:string 1:n
OMV version	0.1
Comments	None

Table B.27: Property: resourceLocator

version	
Name Type Identifier	version DatatypeProperty
Occurrence Constraint Category	required Availability information
Definition	Specifies the version information of the ontology
Domain Range Cardinality	omv:Ontology xsd:string 1:1
OMV version	0.1
Comments	Version information could be useful for tracking, comparing and merging ontologies. It is highly recommended the use of a well defined numbering schema for the version information (e.g. X.Y.Z where X is a major release, Y is minor release and Z is a revision number)

Table B.28: Property: version

hasLicense	
Name Type Identifier	hasLicense ObjectProperty
Occurrence Constraint Category	optional Availability information
Definition	Underlying license model
Domain Range Cardinality	omv:Ontology omv:LicenseModel 0:1
OMV version	0.1
Comments	Reference to a concrete LicenseModel Pre-defined values

Table B.29: Property: hasLicense

useImports	
Name Type Identifier	useImports ObjectProperty
Occurrence Constraint Category	optional Relationship information
Definition	References another ontology metadata instance that describes an ontology containing definitions, whose meaning is considered to be part of the meaning of the ontology described by this ontology metadata instance
Domain Range Cardinality	omv:Ontology omv:Ontology 0:n
OMV version	0.1
Comments	Each reference consists of a URI

Table B.30: Property: useImports

hasPriorVersion	
Name Type Identifier	hasPriorVersion ObjectProperty
Occurrence Constraint Category	optional Relationship information
Definition	Contains a reference to another ontology metadata instance
Domain Range Cardinality	omv:Ontology omv:Ontology 0:1
OMV version	0.1
Comments	Identifies the ontology metadata instance which describes an ontology that is a prior version of the ontology described by this ontology metadata instance. It may be used to organize ontologies by versions and is NULL for initial ontology

Table B.31: Property: hasPriorVersion

2006–2007 © Copyright lies with the respective authors and their institutions.

isBackwardCompatibleWith	
Name Type Identifier	isBackwardCompatibleWith ObjectProperty
Occurrence Constraint Category	optional Relationship information
Definition	This property identifies the ontology metadata instance which describes an ontology that is a compatible prior version of the ontology described by this ontology metadata instance
Domain Range Cardinality	omv:Ontology omv:Ontology 0:n
OMV version	0.1
Comments	This also indicates that all identifiers from the previous version have the same intended interpretations in the new version

Table B.32: Property: isBackwardCompatibleWith

isIncompatibleWith	
Name Type Identifier	isIncompatibleWith ObjectProperty
Occurrence Constraint Category	optional Relationship information
Definition	This property indicates that the described ontology is a later version of the ontology described by the metadata specified, but is not backward compatible with it. It can be used to explicitly state that ontology cannot upgrade to use the new version without checking whether changes are required.
Domain Range Cardinality	omv:Ontology omv:Ontology 0:n
OMV version	0.1
Comments	None

Table B.33: Property: isIncompatibleWith

numberOfClasses	
Name Type Identifier	numberOfClasses DatatypeProperty
Occurrence Constraint Category	required Statistic information
Definition	Number of classes in the ontology
Domain Range Cardinality	omv:Ontology xsd:unsignedLong 1:1
OMV version	0.1
Comments	Language specific value

Table B.34: Property: numberOfClasses

numberOfProperties	
Name Type Identifier	numberOfProperties DatatypeProperty
Occurrence Constraint Category	required Statistic information
Definition	Number of properties in the ontology
Domain Range Cardinality	omv:Ontology xsd:unsignedLong 1:1
OMV version	0.1
Comments	Language specific value

Table B.35: Property: numberOfProperties

numberOfIndividuals	
Name Type Identifier	numberOfIndividuals DatatypeProperty
Occurrence Constraint Category	required Statistic information
Definition	Number of individuals in the ontology
Domain Range Cardinality	omv:Ontology xsd:unsignedLong 1:1
OMV version	0.1
Comments	Language specific value

Table B.36: Property: numberOfIndividuals

numberOfAxioms	
Name Type Identifier	numberOfAxioms DatatypeProperty
Occurrence Constraint Category	required Statistic information
Definition	Number of axioms in the ontology
Domain Range Cardinality	omv:Ontology xsd:unsignedLong 1:1
OMV version	0.1
Comments	Language specific value

Table B.37: Property: numberOfAxioms

2006–2007 © Copyright lies with the respective authors and their institutions.

B.2 OntologyType

This class subsumes types of ontologies according to well-known classifications in the Ontology Engineering literature [GPFLC03].

OntologyType	
Name Type Identifier	OntologyType class
Definition	Categorizes ontologies
OMV version	0.3
Comments	None

Table B.38: Class: OntologyType

name	
Name Type Identifier	name DatatypeProperty
Occurrence Constraint Category	required General information
Definition	The name by which an ontology type is formally known
Domain Range Cardinality	omv:OntologyType xsd:string 1:1
OMV version	0.1
Comments	None

Table B.39: Property: name

acronym	
Name Type Identifier	acronym DatatypeProperty
Occurrence Constraint Category	optional General information
Definition	A short name by which an ontology type is formally known
Domain Range Cardinality	omv:OntologyType xsd:string 0:1
OMV version	0.1
Comments	None

Table B.40: Property: acronym

description	
Name Type Identifier	description DatatypeProperty
Occurrence Constraint Category	optional General information
Definition	Free text description of an ontology type
Domain Range Cardinality	omv:OntologyType xsd:string 0:1
OMV version	0.1
Comments	None

Table B.41: Property: description

documentation	
Name Type Identifier	documentation DatatypeProperty
Occurrence Constraint Category	optional General information
Definition	URL for further documentation
Domain Range Cardinality	omv:OntologyType xsd:string 0:1
OMV version	0.2
Comments	None

Table B.42: documentation

definedBy	
Name Type Identifier	definedBy ObjectProperty, inverseOf(definesOntologyType)
Occurrence Constraint Category	optional General information
Definition	References a party that defined the ontology type
Domain Range Cardinality	omv:OntologyType omv:Party 0:n
OMV version	0.6
Comments	None

Table B.43: typeDefinedBy

2006–2007 © Copyright lies with the respective authors and their institutions.

B.2.1 Pre-defined ontology types

Individuals of the class `OntologyType` refer to well-known classifications for ontologies in the literature. Currently the OMV model resorts to a classification on the generality levels of the conceptualisation [Gua98, WW02]:

- upper level ontologies describing general, domain-independent concepts e.g. space, time.
- core ontologies describing the most important concepts in a specific domain
- domain ontology describing some domain of the world
- task ontology describing generic types of tasks or activities e.g. selling, selecting.
- application ontology describing some domain in an application-dependent manner

The class can be extended to support additional classifications (e.g. the one in [McG02]).

B.3 LicenseModel

LicenseModel	
Name Type Identifier	LicenseModel class LM
Definition	A license model describing the usage conditions for an ontology
OMV version	0.3
Comments	None

Table B.44: Class: LicenseModel

name	
Name Type Identifier	name DatatypeProperty Used Identifier for this entity.
Occurrence Constraint Category	required Availability information
Definition	The name by which a license model is formally known
Domain Range Cardinality	omv:LicenseModel xsd:string 1:1
OMV version	0.1
Comments	None

Table B.45: Property: name

acronym	
Name Type Identifier	acronym DatatypeProperty
Occurrence Constraint Category	optional Availability information
Definition	A short name by which a license model is formally known
Domain Range Cardinality	omv:LicenseModel xsd:string 0:1
OMV version	0.1
Comments	None

Table B.46: Property: acronym

description	
Name Type Identifier	description DatatypeProperty
Occurrence Constraint Category	optional Availability information
Definition	Descriptional free text about a license model
Domain Range Cardinality	omv:LicenseModel xsd:string 0:1
OMV version	0.1
Comments	None

Table B.47: Property: description

2006–2007 © Copyright lies with the respective authors and their institutions.

documentation	
Name Type Identifier	documentation DatatypeProperty
Occurrence Constraint Category	optional Availability information
Definition	URL for further documentation
Domain Range Cardinality	omv:LicenseModel xsd:string 0:1
OMV version	0.2
Comments	None

Table B.48: documentation

specifiedBy	
Name Type Identifier	specifiedBy ObjectProperty, inverseOf(specifiesLicense) Used Identifier for this element.
Occurrence Constraint Category	optional Availability information
Definition	References a party that specified the license model
Domain Range Cardinality	omv:LicenseModel omv:Party 0:n
OMV version	0.6
Comments	None

Table B.49: specifiedBy

B.3.1 Pre-defined license models

Individuals of the class `LicenseModel` refer to well-known license models, such as: [1]

- Academic Free License (AFL)
- Common Public License (CPL)
- Lesser General Public License (LGPL)
- Open Software License (OSL)
- General Public License (GPL)
- Modified BSD License (mBSD)
- IBM Public License (IBM PL)
- Apple Public Source License (APSL)
- INTEL Open Source License (INTEL OSL)

The class can be extended to support additional classifications.

[1] A description of these models can be found in `http://www.gnu.org/licenses/license-list.html`

2006–2007 © Copyright lies with the respective authors and their institutions.

B.4 OntologyEngineeringMethodology

OntologyEngineeringMethodology	
Name Type Identifier	OntologyEngineeringMethodology class
Definition	Information about the ontology engineering methodology
OMV version	0.3
Comments	None

Table B.50: Class: OntologyEngineeringMethodology

name	
Name Type Identifier	name DatatypeProperty
Occurrence Constraint Category	required Other
Definition	The name by which a ontology engineering method is formally known
Domain Range Cardinality	omv:OntologyEngineeringMethodology xsd:string 1:1
OMV version	0.1
Comments	None

Table B.51: Property: name

acronym	
Name Type Identifier	acronym DatatypeProperty
Occurrence Constraint Category	optional Other
Definition	A short name by which a ontology engineering method is known
Domain Range Cardinality	omv:OntologyEngineeringMethodology xsd:string 0:1
OMV version	0.1
Comments	None

Table B.52: Property: acronym

description	
Name Type Identifier	description DatatypeProperty
Occurrence Constraint Category	optional Other
Definition	Free text description of an ontology engineering method
Domain Range Cardinality	omv:OntologyEngineeringMethodology xsd:string 0:1
OMV version	0.6
Comments	None

Table B.53: Property: description

documentation	
Name	documentation
Type	DatatypeProperty
Identifier	
Occurrence Constraint	optional
Category	Other
Definition	URL for further documentation
Domain	omv:OntologyEngineeringMethodology
Range	xsd:string
Cardinality	0:1
OMV version	0.6
Comments	None

Table B.54: documentation

developedBy	
Name	developedBy
Type	ObjectProperty inverseOf(developesOntologyEngineeringMethodology)
Identifier	
Occurrence Constraint	optional
Category	Other
Definition	A party that developed the ontology engineering methodology
Domain	omv:OntologyEngineeringMethodology
Range	omv:Party
Cardinality	0:n
OMV version	0.6
Comments	None

Table B.55: developedBy

2006–2007 © Copyright lies with the respective authors and their institutions.

B.5 OntologyEngineeringTool

OntologyEngineeringTool	
Name Type Identifier	OntologyEngineeringTool class
Definition	A tool used to create the ontology
OMV version	0.3
Comments	None

Table B.56: Class: OntologyEngineeringTool

name	
Name Type Identifier	name DatatypeProperty
Occurrence Constraint Category	required Other
Definition	The name by which a tool is formally known
Domain Range Cardinality	omv:OntologyEngineeringTool xsd:string 1:1
OMV version	0.1
Comments	None

Table B.57: Property: name

acronym	
Name Type Identifier	acronym DatatypeProperty
Occurrence Constraint Category	optional Other
Definition	A short name by which a tool is known
Domain Range Cardinality	omv:OntologyEngineeringTool xsd:string 0:1
OMV version	0.1
Comments	None

Table B.58: Property: acronym

description	
Name Type Identifier	description DatatypeProperty
Occurrence Constraint Category	optional Other
Definition	Free text description of the tool
Domain Range Cardinality	omv:OntologyEngineeringTool xsd:string 0:1
OMV version	0.1
Comments	None

Table B.59: Property: description

documentation	
Name Type Identifier	documentation DatatypeProperty
Occurrence Constraint Category	optional Other
Definition	URL for further documentation.
Domain Range Cardinality	omv:OntologyEngineeringTool xsd:string 0:n
OMV version	0.2
Comments	None

Table B.60: documentation

developedBy	
Name Type Identifier	developedBy ObjectProperty inverseOf(developesOntologyEngineeringTool)
Occurrence Constraint Category	optional Other
Definition	References the tool developer party
Domain Range Cardinality	omv:OntologyEngineeringTool omv:Party 0:n
OMV version	0.4
Comments	None

Table B.61: developedBy

2006–2007 © Copyright lies with the respective authors and their institutions.

B.6 OntologySyntax

OntologySyntax	
Name Type Identifier	OntologySyntax class
Definition	Information about the syntax used in an OI
OMV version	0.3
Comments	None

Table B.62: Class: OntologySyntax

name	
Name Type Identifier	name DatatypeProperty
Occurrence Constraint Category	required Format information
Definition	The name by which an ontology syntax is formally known
Domain Range Cardinality	omv:OntologySyntax xsd:string 1:1
OMV version	0.1
Comments	None

Table B.63: Property: name

acronym	
Name Type Identifier	acronym DatatypeProperty
Occurrence Constraint Category	optional Format information
Definition	A short name by which an ontology syntax is known
Domain Range Cardinality	omv:OntologySyntax xsd:string 0:1
OMV version	0.1
Comments	None

Table B.64: Property: acronym

description	
Name Type Identifier	description DatatypeProperty
Occurrence Constraint Category	optional Format information
Definition	Free text description of the used syntax
Domain Range Cardinality	omv:OntologySyntax xsd:string 0:1
OMV version	0.6
Comments	None

Table B.65: Property: description

documentation	
Name Type Identifier	documentation DatatypeProperty
Occurrence Constraint Category	optional Format information
Definition	URL for further documentation.
Domain Range Cardinality	omv:OntologySyntax xsd:string 0:1
OMV version	0.6
Comments	None

Table B.66: documentation

developedBy	
Name Type Identifier	developedBy ObjectProperty inverseOf(developedOntologySyntax)
Occurrence Constraint Category	optional Format information
Definition	The party who developed the used syntax
Domain Range Cardinality	omv:OntologySyntax omv:Party 0:n
OMV version	0.6
Comments	None

Table B.67: developedBy

2006–2007 © Copyright lies with the respective authors and their institutions.

B.6.1 Pre-defined ontology syntaxes

Individuals of the class `OntologySyntax` refers to well-known ontology syntax standards, such as:

- OWL-XML
- RDF/XML

The class can be extended to support additional classifications.

B.7 OntologyLanguage

OntologyLanguage	
Name Type Identifier	OntologyLanguage class
Definition	Information about the language in which the ontology is implemented
OMV version	0.3
Comments	None

Table B.68: Class: OntologyLanguage

name	
Name Type Identifier	name DatatypeProperty
Occurrence Constraint Category	required Format information
Definition	The name by which an ontology language is formally known
Domain Range Cardinality	omv:OntologyLanguage xsd:string 1:1
OMV version	0.1
Comments	None

Table B.69: Property: name

acronym	
Name Type Identifier	acronym DatatypeProperty
Occurrence Constraint Category	optional Format information
Definition	A short name by which an ontology language is known
Domain Range Cardinality	omv:OntologyLanguage xsd:string 0:1
OMV version	0.1
Comments	None

Table B.70: Property: acronym

description	
Name Type Identifier	description DatatypeProperty
Occurrence Constraint Category	optional Format information
Definition	Free text description of an ontology language.
Domain Range Cardinality	omv:OntologyLanguage xsd:string 0:1
OMV version	0.6
Comments	None

Table B.71: Property: description

2006–2007 © Copyright lies with the respective authors and their institutions.

documentation	
Name Type Identifier	documentation DatatypeProperty
Occurrence Constraint Category	optional Format information
Definition	URL for further documentation
Domain Range Cardinality	omv:OntologyLanguage xsd:string 0:1
OMV version	0.6
Comments	None

Table B.72: Property: documentation

developedBy	
Name Type Identifier	developedBy ObjectProperty, inverseOf(developesOntologyLanguage)
Occurrence Constraint Category	optional Format information
Definition	References the party who developed the language
Domain Range Cardinality	omv:OntologyLanguage omv:Party 0:n
OMV version	0.6
Comments	None

Table B.73: Property: developedBy

supportsRepresentationParadigm	
Name Type Identifier	supportsRepresentationParadigm ObjectProperty
Occurrence Constraint Category	optional Format information
Definition	Specifies the representation paradigm supported by the ontology language
Domain Range Cardinality	omv:OntologyLanguage omv:Party 0:n
OMV version	0.6
Comments	None

Table B.74: Property: supportsRepresentationParadigm

hasSyntax	
Name Type Identifier	hasSyntax ObjectProperty
Occurrence Constraint Category	optional Format information
Definition	References the syntactical alternatives of the language
Domain Range Cardinality	omv:OntologyLanguage omv:OntologySyntax 0:n
OMV version	0.6
Comments	None

Table B.75: Property: hasSyntax

B.7.1 Pre-defined ontology languages

Individuals of the class `OntologyLanguage` refer to well-known ontology language standards, such as:

- OWL
- OWL-DL
- OWL-Lite
- OWL-Full
- DAML-OIL
- RDF(S)

The class can be extended to support additional classifications.

2006–2007 © Copyright lies with the respective authors and their institutions.

B.8 KnowledgeRepresentationParadigm

KnowledgeRepresentationParadigm	
Name Type Identifier	KnowledgeRepresentationParadigm class
Definition	Information about a knowledge representation paradigm a particular language adheres to
OMV version	0.9.1
Comments	E. g. Description Logics, Frames

Table B.76: Class: KnowledgeRepresentationParadigm

name	
Name Type Identifier	name DatatypeProperty
Occurrence Constraint Category	required Format information
Definition	The name by which a KR paradigm is formally known
Domain Range Cardinality	omv:KnowledgeRepresentationParadigm xsd:string 1:1
OMV version	0.9.1
Comments	None

Table B.77: Property: name

acronym	
Name Type Identifier	acronym DatatypeProperty
Occurrence Constraint Category	optional Format information
Definition	A short name by which a kR paradigm is known
Domain Range Cardinality	omv:KnowledgeRepresentationParadigm xsd:string 0:1
OMV version	0.9.1
Comments	None

Table B.78: Property: acronym

description	
Name Type Identifier	description DatatypeProperty
Occurrence Constraint Category	optional Format information
Definition	Free text description of the knowledge representation paradigm
Domain Range Cardinality	omv:KnowledgeRepresentationParadigm xsd:string 0:1
OMV version	0.9.1
Comments	None

Table B.79: Property: description

documentation	
Name	documentation
Type	DatatypeProperty
Identifier	
Occurrence Constraint	optional
Category	Format information
Definition	URL for further documentation
Domain	omv:KnowledgeRepresentationParadigm
Range	xsd:string
Cardinality	0:1
OMV version	0.9.1
Comments	None

Table B.80: documentation

specifiedBy	
Name	specifiedBy
Type	ObjectProperty inverseOf specifiesKnowledgeRepresentationParadigm
Identifier	
Occurrence Constraint	optional
Category	Format information
Definition	Author of the KR paradigm
Domain	omv:KnowledgeRepresentationParadigm
Range	omv:Party
Cardinality	0:n
OMV version	0.1
Comments	None

Table B.81: Property: specifiedBy

2006–2007 © Copyright lies with the respective authors and their institutions.

B.8.1 Pre-defined knowledge representation paradigms

In this version we foresee two main classes of `KnowledgeRepresentationParadigm`s:

- Description Logics
- Frames

B.9 FormalityLevel

FormalityLevel	
Name Type Identifier	FormalityLevel class
Definition	The level of formality of an ontology
OMV version	0.9.1
Comments	According to classifications in the OE literature

Table B.82: Class: FormalityLevel

name	
Name Type Identifier	name DatatypeProperty
Occurrence Constraint Category	required Applicability information
Definition	The name by which this element is formally known
Domain Range Cardinality	omv:FormalityLevel xsd:string 1:1
OMV version	0.9.1
Comments	None

Table B.83: Property: name

description	
Name Type Identifier	description DatatypeProperty
Occurrence Constraint Category	optional Format information
Definition	Free text description of the formality level
Domain Range Cardinality	omv:FormalityLevel xsd:string 0:1
OMV version	0.9.1
Comments	None

Table B.84: Property: description

B.9.1 Pre-defined formality levels

The pre-defined values for the formality level are based on the work presented in [LM01], which classifies ontologies in a spectrum of definitions according to the detail in their specification as: catalog, glossary, thesauri, taxonomy, frames and properties, value restrictions, disjointness, general logic constraints.

2006–2007 © Copyright lies with the respective authors and their institutions.

B.10 OntologyTask

OntologyTask	
Name Type Identifier	OntologyTask class
Definition	Information about the task the ontology was intended to be used for
OMV version	0.9.1
Comments	Super-class of classes modelling typical ontology-related tasks

Table B.85: Class: OntologyTask

name	
Name Type Identifier	taskName DatatypeProperty
Occurrence Constraint Category	required Applicability information
Definition	The name by which an ontology task is formally known
Domain Range Cardinality	omv:OntologyTask xsd:string 1:1
OMV version	0.9.1
Comments	None

Table B.86: Property: name

acronym	
Name Type Identifier	acronym DatatypeProperty
Occurrence Constraint Category	optional Applicability information
Definition	A short name by which an ontology task is known
Domain Range Cardinality	omv:OntologyTask xsd:string 0:1
OMV version	0.9.1
Comments	None

Table B.87: Property: acronym

description	
Name Type Identifier	description DatatypeProperty
Occurrence Constraint Category	optional Applicability information
Definition	Free text description of the ontology task
Domain Range Cardinality	omv:OntologyTask xsd:string 0:1
OMV version	0.9.1
Comments	None

Table B.88: Property: description

documentation	
Name Type Identifier	`documentation` `DatatypeProperty`
Occurrence Constraint Category	`optional` Applicability information
Definition	URL for further documentation

Domain
Range
Cardinality
	`omv:OntologyTask`
	`xsd:string`
	0:1

OMV version	0.9.1
Comments	None

Table B.89: documentation

2006–2007 © Copyright lies with the respective authors and their institutions.

B.10.1 Pre-defined ontology tasks

Individuals of the class `OntologyTask` refer to particular application scenarios for ontologies, in which the benefits of using ontologies are widely acknowledged. We differentiate among the following tasks:

AnnotationTask : the ontology is used as a controlled vocabulary to annotate Semantic Web resources. This task includes the usage of a semantically rich ontology for representing arbitrarily complex annotation statements on these resources. The task can be performed manually or (semi-)automatically.

ConfigurationTask : the ontology is designed to provide a controlled and unambiguous means to represent valid configuration profiles in application systems. As the aim of the ontology is to support the operationalization of particular system-related processes; this task is performed automatically in that the ontology is processed in an automatic manner by means of reasoners or APIs.

FilteringTask : the task describes at a very general level how ontologies are applied to refine the solution space of a certain problem, such as information retrieval or personalization. The task is targeted at being performed semi-automatically or automatically.

IndexingTask : in this scenario, the goal of the ontology is to provide a clearly defined classification and browsing structure for the information items in a repository. Again, the task can be performed manually by domain experts or as part of an application in an automatic or semi-automatic way.

IntegrationTask : the task characterizes how ontologies provide an integrating environment, an interlingua, for information repositories or software tools. In this scenario the ontology is applied (semi-)automatically to merge between heterogeneous data pools in the same or in adjacent domains.

MatchingTask : the goal of matching is to establish links between semantically similar data items in information repositories. In contrast to the previous task, matching does not include the production of a shared final schema/ontology as a result of aggregating the matched source elements to common elements. W.r.t. the automatization level the range varies from manual to fully-automatical execution.

MediationTask : the ontology is built to reduce the ambiguities between communicating human or machine agents. It can act as a normative model which formally and clearly defines the meaning of the terms employed in agent interactions. In the context of programmed agents, the task is envisioned to be performed automatically.

QueryFormulationTask : the ontology is used in information retrieval settings as a controlled vocabulary for representing user queries. Usually the task is performed automatically in that the concepts of the ontology is are listed in a query formulation front-end in order to allow users to specifies their queries.

QueryRewritingTask : complementary to the query formulation dimension, this task applies ontologies to semantically optimize query expressions by means of the domain knowledge (constraints, subsumption relations etc.) The task can be interpreted as a particular art of filtering information. The task is performed automatically; however, it assumes the availability of patterns describing the transformations at query level.

PersonalizationTask : the ontology is used mainly for providing personalized access to information resources. Individual user preferences w.r.t. particular application settings are formally specified by means of an ontology, which, in conjunction with appropriate reasoning services, can be directly integrated to a personalization component for filtering purposes. The usage of ontologies in personalization tasks might be carried out in various forms, from a direct involvement of the user who manually specifies ontological concepts which optimally describe his preferences, to the ontological modelling of user profiles.

SearchTask : the task characterizes how ontologies are used to refine common keyword-based search algorithms using domain knowledge in form of subsumption relations. Ontology-driven search is usually performed automatically by means of reasoning services handling particular aspects of an ontology representation language.

2006–2007 © Copyright lies with the respective authors and their institutions.

B.11 OntologyDomain

OntologyDomain	
Name Type Identifier	OntologyDomain `OntologyDomain`
Definition	Typically, the domain refers to established topic hierarchies such as the general purpose topic hierarchy DMOZ or the domain specific topic hierarchy ACM for the computer science domain
Comments	None

Table B.90: Class: OntologyDomain

URI	
Name Type Identifier	URI `DatatypeProperty`
Occurrence Constraint Category	`required` General information
Definition	The URI of the ontology domain

Domain
Range
Cardinality
	`omv:OntologyDomain`
	`xsd:string`
	1:1

OMV version	2.1
Comments	None

Table B.91: Property: URI

name	
Name Type Identifier	name `DatatypeProperty`
Occurrence Constraint Category	`required` General information
Definition	The name by which an ontology domain is formally known

Domain
Range
Cardinality
	`omv:Ontology`
	`xsd:string`
	1:1

OMV version	2.1
Comments	None

Table B.92: Property: name

isSubDomainOf	
Name Type Identifier	isSubDomainOf `ObjectProperty`
Occurrence Constraint Category	`optional` Applicability information
Definition	Specifies the domain topic of which this domain topic is a sub domain
Domain Range Cardinality	`omv:OntologyDomain` `omv:OntologyDomain` 0:n
OMV version	0.8
Comments	Typically, the domain can refer to established topic hierarchies such as the general purpose topic hierarchy DMOZ or the domain specific topic hierarchy ACM for the computer science domain

Table B.93: Property: isSubDomainOf

B.12 Party

Party	
Name Type Identifier	Party `class`
Definition	A party is a person or an organisation
OMV version	0.4
Comments	None

Table B.94: Class: Party

isLocatedAt	
Name Type Identifier	isLocatedAt `ObjectProperty`
Occurrence Constraint Category	`optional` Availability Information
Definition	The geographical location of a party

Domain
Range
Cardinality
	`omv:Party`
	`omv:Location`
	0:n

OMV version	0.9
Comments	None

Table B.95: Property: isLocatedAt

developesOntologyEngineeringTool	
Name Type Identifier	developesOntologyEngineeringTool `ObjectProperty, inverseOf(toolDeveloper)`
Occurrence Constraint Category	`optional` Provenance information
Definition	References a tool developed by a party

Domain
Range
Cardinality
	`omv:Party`
	`omv:OntologyEngineeringTool`
	0:n

OMV version	0.6
Comments	None

Table B.96: Property: developesOntologyEngineeringTool

developesOntologyLanguage	
Name Type Identifier	developesOntologyLanguage `ObjectProperty, inverseOf(languageDeveloper)`
Occurrence Constraint Category	`optional` Provenance information
Definition	References an ontology language developed by a party

Domain
Range
Cardinality
	`omv:Party`
	`omv:OntologyLanguage`
	0:n

OMV version	0.6
Comments	None

Table B.97: Property: developeaOntologyLanguage

2006–2007 © Copyright lies with the respective authors and their institutions.

developesOntologySyntax	
Name	developesOntologySyntax
Type	`ObjectProperty, inverseOf(syntaxDeveloper)`
Identifier	
Occurrence Constraint	`optional`
Category	Provenance information
Definition	References an ontology syntax developed by a party
Domain	`omv:Party`
Range	`omv:OntologySyntax`
Cardinality	0:n
OMV version	0.6
Comments	None

Table B.98: Property: developesOntologySyntax

specifiesKnowledgeRepresentationParadigm	
Name	specifiesKnowledgeRepresentationParadigm
Type	`ObjectProperty inverseOf(knowledgeRepresentationParadigmSpecifiedBy)`
Identifier	
Occurrence Constraint	`optional`
Category	Provenance information
Definition	References an KR paradigm specified or defined by a party
Domain	`omv:Party`
Range	`omv:KnowledgeRepresentationParadigm`
Cardinality	0:n
OMV version	0.6
Comments	None

Table B.99: Property: specifiesKnowledgeRepresentationParadigm

definesOntologyType	
Name	definesOntologyType
Type	`ObjectProperty, inverseOf(definedBy)`
Identifier	
Occurrence Constraint	`optional`
Category	Provenance information
Definition	Reference an ontology type defined by a party
Domain	`omv:Party`
Range	`omv:OntologyType`
Cardinality	0:n
OMV version	0.6
Comments	None

Table B.100: Property: definesOntologyType

developesOntologyEngineeringMethodology	
Name	developesOntologyEngineeringMethodology
Type	`ObjectProperty`
Identifier	
Occurrence Constraint	`optional`
Category	Provenance information
Definition	References a ontology engineering method developed by a party
Domain	`omv:Party`
Range	`omv:OntologyEngineeringMethod`
Cardinality	0:n
OMV version	0.6
Comments	None

Table B.101: Property: developesOntologyEngineeringMethodology

specifiesLicense	
Name	specifiesLicense
Type	`ObjectProperty, inverseOf(licenseSpecifiedBy)`
Identifier	
Occurrence Constraint	`optional`
Category	Provenance information
Definition	References a license specified by a party
Domain	`omv:Party`
Range	`omv:LicenseModel`
Cardinality	0:n
OMV version	0.6
Comments	None

Table B.102: Property: specifiesLicense

hasAffiliatedParty	
Name	hasAffiliatedParty
Type	`ObjectProperty`
Identifier	
Occurrence Constraint	`optional`
Category	Provenance information
Definition	References another party that is affiliated with the instance
Domain	`omv:Party`
Range	`omv:Party`
Cardinality	0:n
OMV version	0.2
Comments	None

Table B.103: Property: hasAffiliatedParty

createsOntology	
Name	`createsOntology`
Type	`ObjectProperty, inverseOf(ontologyCreator)`
Identifier	
Occurrence Constraint	`optional`
Category	Provenance information
Definition	References an ontology created by a party
Domain	`omv:Party`
Range	`omv:Ontology`
Cardinality	0:n
OMV version	0.7
Comments	None

Table B.104: Property: createsOntology

contributesToOntology	
Name	`contributesToOntology`
Type	`ObjectProperty inverseOf(ontologyContributor)`
Identifier	
Occurrence Constraint	`optional`
Category	Provenance information
Definition	References an ontology a party made contributions to
Domain	`omv:Party`
Range	`omv:Ontology`
Cardinality	0:n
OMV version	0.7
Comments	None

Table B.105: Property: contributesToOntology

2006–2007 © Copyright lies with the respective authors and their institutions.

B.13 Person

Person	
Name Type Identifier	Person `class`
Definition	A named individual
OMV version	0.1
Comments	Represents an individual responsible for the creation, or contribution to an ontology

Table B.106: Class: Person

lastName	
Name Type Identifier	lastName `DatatypeProperty`
Occurrence Constraint Category	`required` Provenance information
Definition	The surname of a person

Domain
Range
Cardinality
	`omv:Person`
	`xsd:string`
	1:1

OMV version	0.2
Comments	None

Table B.107: lastName

firstName	
Name Type Identifier	firstName `DatatypeProperty`
Occurrence Constraint Category	`required` Provenance information
Definition	The first name of a person

Domain
Range
Cardinality
	`omv:Person`
	`xsd:string`
	1:n

OMV version	0.2
Comments	None

Table B.108: firstname

eMail	
Name Type Identifier	email `DatatypeProperty`
Occurrence Constraint Category	`required` Provenance information
Definition	The email address of a person

Domain
Range
Cardinality
	`omv:Person`
	`xsd:string`
	1:n

OMV version	0.1
Comments	None

Table B.109: eMail

phoneNumber	
Name Type Identifier	phoneNumber DatatypeProperty
Occurrence Constraint Category	optional Provenance information
Definition	The phone number of a person
Domain Range Cardinality	omv:Person xsd:string 0:n
OMV version	0.1
Comments	None

Table B.110: Property: phoneNumber

faxNumber	
Name Type Identifier	faxNumber DatatypeProperty
Occurrence Constraint Category	optional Provenance information
Definition	The fax number of a person
Domain Range Cardinality	omv:Person xsd:string 0:n
OMV version	0.1
Comments	None

Table B.111: faxNumber

isContactPerson	
Name Type Identifier	isContactPerson ObjectProperty, inverseOf(contactPerson)
Occurrence Constraint Category	optional Provenance information
Definition	Instance is contact person of an organisation
Domain Range Cardinality	omv:Person omv:Organisation 0:n
OMV version	0.7
Comments	None

Table B.112: isContactPerson

2006–2007 © Copyright lies with the respective authors and their institutions.

B.14 Organisation

Organisation	
Name Type Identifier	Organisation `class, subclassOf(Party)`
Definition	An organisation of some kind
OMV version	0.6
Comments	Represents social institutions such as universities, companies, societies etc.

Table B.113: Class: Organisation

name	
Name Type Identifier	name `DatatypeProperty`
Occurrence Constraint Category	`required` Provenance information
Definition	The name by which an organisation is formally known

Domain
Range
Cardinality
	`omv:Organisation`
	`xsd:string`
	1:1

OMV version	0.1
Comments	None

Table B.114: Property: name

acronym	
Name Type Identifier	acronym `DatatypeProperty` Used Identifier for this entity.
Occurrence Constraint Category	`required` Provenance information
Definition	A short name by which an organisation is known

Domain
Range
Cardinality
	`omv:Organisation`
	`xsd:string`
	1:1

OMV version	0.1
Comments	None

Table B.115: Property: acronym

hasContactPerson	
Name Type Identifier	`hasContactPerson` `ObjectProperty, inverseOf(ishasContactPerson)`
Occurrence Constraint Category	`optional` Provenance information
Definition	A contact person in the organisation

Domain
Range
Cardinality
	`omv:Organisation`
	`omv:Person`
	0:n

OMV version	0.6
Comments	None

Table B.116: hasContactPerson

B.15 Location

Location	
Name Type Identifier	Location class
Definition	A location.
OMV version	0.9
Comments	The geographical location of a party. To keep things simple we use only DatatypeProperties instead of introducing classes for country, street, etc.

Table B.117: Class: Location

state	
Name Type Identifier	state DatatypeProperty
Occurrence Constraint Category	optional Provenance information
Definition	The state of a country.
Domain Range Cardinality	omv:Location xsd:string 0:1
OMV version	0.9
Comments	None

Table B.118: Property: state

country	
Name Type Identifier	country DatatypeProperty
Occurrence Constraint Category	optional Provenance information
Definition	The name of the country
Domain Range Cardinality	omv:Location xsd:string 0:1
OMV version	0.9
Comments	Changed the name from land to country

Table B.119: Property: country

city	
Name Type Identifier	city DatatypeProperty
Occurrence Constraint Category	optional Provenance information
Definition	Name of the city (and zip code).
Domain Range Cardinality	omv:Location xsd:string 0:1
OMV version	0.9
Comments	None

Table B.120: Property: city

2006–2007 © Copyright lies with the respective authors and their institutions.

street	
Name Type Identifier	street DatatypeProperty
Occurrence Constraint Category	optional Provenance information
Definition	Name of the street and number (address).
Domain Range Cardinality	omv:Location xsd:string 0:1
OMV version	0.9
Comments	None

Table B.121: Property: street

Bibliography

[ABdB+05] J. Angele, H. Boley, J. de Bruijn, D. Fensel, P. Hitzler, M. Kifer, R. Krummenacher, H. Lausen, A. Polleres, and R. Studer. *Web Rule Language (WRL)*. World Wide Web Consortium, September 2005. W3C Member Submission, `http://www.w3.org/Submission/WRL/`.

[BCH06a] J. Bao, D. Caragea, and V. Honavar. A distributed tableau algorithm for package-based description logics. In *the 2nd International Workshop On Context Representation And Reasoning (CRR 2006), co-located with ECAI 2006*. 2006.

[BCH06b] J. Bao, D. Caragea, and V. Honavar. Modular Ontologies - A Formal Investigation of Semantics and Expressivity. In *R. Mizoguchi, Z. Shi, and F. Giunchiglia (Eds.): Asian Semantic Web Conference 2006, LNCS 4185*, pages 616–631, 2006.

[BCH06c] J. Bao, D. Caragea, and V. Honavar. On the semantics of linking and importing in modular ontologies. In *I. Cruz et al. (Eds.): ISWC 2006, LNCS 4273 (In Press)*, pages 72–86. 2006.

[BCH06d] J. Bao, D. Caragea, and V. Honavar. Towards collaborative environments for ontology construction and sharing. In *International Symposium on Collaborative Technologies and Systems (CTS 2006)*, pages 99–108. IEEE Press, 2006.

[BCM+03] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, New York, NY, USA, 2003.

[Bec04] D. Beckett. RDF/XML Syntax Specification. Technical report, World Wide Web Consortium (W3C), February 2004. Internet: http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/.

[BG04] D. Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. Technical report, World Wide Web Consortium (W3C), February 2004. Internet: http://www.w3.org/TR/rdf-schema/.

[BGS+03] F. Budinsky, T. J. Grose, D. Steinberg, R. Ellersick, E. Merks, and S. A. Brodsky. *Eclipse Modeling Framework: a developer's guide*. Addison Wesley Professional, 2003.

[BGvH+03] P. Bouquet, F. Giunchiglia, F. van Harmelen, L. Serafini, and H. Stuckenschmidt. C-OWL: Contextualizing Ontologies. In *Second International Semantic Web Conference ISWC'03*, volume 2870 of *LNCS*, pages 164–179. Springer, 2003.

[BKK+01] K. Baclawski, M. Kokar, P. Kogut, L. Hart, J. Smith, W. Holmes, J. Letkowski, and M. Aronson. Extending UML to Support Ontology Engineering for the Semantic Web. In *4th Int. Conf. on UML (UML 2001)*, Toronto, Canada, October 2001.

[BKK+02] K. Baclawski, M. M. Kokar, P.A. Kogut, L. Hart, J. Smith, J. Letkowski, and P. Emery. Extending the Unified Modeling Language for Ontology Development. *Software and Systems Modeling*, 1(2):142–156, 2002.

2006–2007 © Copyright lies with the respective authors and their institutions.

[BLFM98] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (uri): Generic syntax, 1998.

[Bro07] S. Brockmans. *Metamodel-based Knowledge Representation*. Phd thesis, University of Karlsruhe (TH), Karlsruhe, Germany, July 2007.

[BS02] Alexander Borgida and Luciano Serafini. Distributed description logics: Directed domain correspondences in federated information sources. In *CoopIS/DOA/ODBASE*, pages 36–53, 2002.

[BS03] Alexander Borgida and Luciano Serafini. Distributed description logics: Assimilating information from peer sources. *J. Data Semantics*, 1:153–184, 2003.

[CDL01] D. Calvanese, G. De Giacomo, and M. Lenzerini. A Framework for Ontology Integration. In *Proceedings of the Semantic Web Working Symposium*, pages 303–316, Stanford, CA, 2001.

[CDL02] D. Calvanese, G. De Giacomo, and M. Lenzerini. Description Logics for Information Integration. In A. Kakas and F. Sadri, editors, *Computational Logic: Logic Programming and Beyond*, volume 2408 of *Lecture Notes in Computer Science*, pages 41–60. Springer, 2002.

[CP99] S. Cranefield and M. Purvis. UML as an Ontology Modelling Language. In *Proceedings of the Workshop on Intelligent Information Integration*, volume 23 of *CEUR Workshop Proceedings*, Stockholm, Sweden, July 1999.

[dSM06] Mathieu d'Aquin, Marta Sabou, and Enrico Motta. Modularization: a key for the dynamic selection of relevant knowledge components. In *Workshop on Modular Ontologies*, 2006.

[FHK97] J. Frohn, R. Himmeröder, and P.-Th. Kandzia. FLORID - A Prototype for F-Logic. Technical report, Institut für Informatik, Universität Freiburg, Germany, 1997.

[Fra03] D. S. Frankel. *Model Driven Architecture*. Wiley Publishing, Inc., Indianapolis, Indiana, 2003.

[GKM05] B. Grosof, M. Kifer, and D. L. Martin. Rules in the Semantic Web Services Language (SWSL): An Overview for Standardization Directions. In *Proceedings of the W3C Workshop on Rule Languages for Interoperability, 27-28 April 2005, Washington, DC, USA*, 2005.

[GM06] B. C. Grau and B. Motik. OWL 1.1 Web Ontology Language - Model-Theoretic Semantics. Technical report, World Wide Web Consortium (W3C), November 2006. Internet: http://owl1-1.cs.manchester.ac.uk/semantics.html.

[GM07] B. Cuenca Grau and B. Motik. OWL 1.1 Web Ontology Language - Mapping to RDF Graphs. Technical report, World Wide Web Consortium (W3C), February 2007. Internet: http://owl1-1.cs.manchester.ac.uk/rdf_mapping.html.

[GMPS07] B. Cuenca Grau, B. Motik, and P. Patel-Schneider. OWL 1.1 Web Ontology Language - XML Syntax. Technical report, World Wide Web Consortium (W3C), February 2007. Internet: http://owl1-1.cs.manchester.ac.uk/xml_syntax.html.

[GPFLC03] A. Gómez-Pérez, M. Fernández-López, and O. Corcho. *Ontological Engineering*. Springer, 2003.

[GPS99] Aldo Gangemi, Domenico M. Pisanelli, and Geri Steve. An overview of the ONIONS project: Applying ontologies to the integration of medical terminologies. *Data Knowledge Engineering*, 31(2):183–220, 1999.

[GPS04] Bernardo Cuenca Grau, Bijan Parsia, and Evren Sirin. Working with multiple ontologies on the semantic web. In *International Semantic Web Conference*, pages 620–634, 2004.

[GPSK05] Bernardo Cuenca Grau, Bijan Parsia, Evren Sirin, and Aditya Kalyanpur. Automatic partitioning of owl ontologies using -connections. In *Description Logics*, 2005.

[Gro05] Object Management Group. Unified Modeling Language: Superstructure. Technical report, Object Management Group, July 2005. Internet: http://www.omg.org/docs/formal/05-07-04.pdf.

[Gua98] N. Guarino. Formal Ontology and Information Systems. In *Proceedings of the FOIS'98*, pages 3–15, 1998.

[HEC+04] L. Hart, P. Emery, B. Colomb, K. Raymond, S. Taraporewalla, D. Chang, Y. Ye, and M. Dutra E. Kendall. OWL full and UML 2.0 compared, March 2004. `http://www.itee.uq.edu.au/$\sim$colomb/Papers/UML-OWLont04.03.01.pdf`.

[HKS06] I. Horrocks, O. Kutz, and U. Sattler. The Even More Irresistable SROIQ. In *Proceedings of the 10th International Conference on Principles of Knowledge Representation and Reasoning (KR 2006)*. AAAI Press, 2006.

[HM05] P. Haase and B. Motik. A Mapping System for the Integration of OWL-DL Ontologies. In *In Proceedings of the ACM-Workshop: Interoperability of Heterogeneous Information Systems (IHIS05)*, November 2005.

[HP05] J. Hartmann and R. Palma. OMV - Ontology Metadata Vocabulary for the Semantic Web, 2005. v. 1.0, available at `http://omv.ontoware.org/`.

[HPPSH05] I. Horrocks, B. Parsia, P. F. Patel-Schneider, and J. A. Hendler. Semantic Web Architecture: Stack or Two Towers? In F. Fages and S. Soliman, editors, *PPSWR*, volume 3703 of *Lecture Notes in Computer Science*, pages 37–41. Springer, 2005.

[HPSB+04] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Technical report, World Wide Web Consortium (W3C), May 2004. W3C Member Submission, `http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/`.

[HT00] I. Horrocks and S. Tessaris. A conjunctive query language for description logic aboxes. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 399–404. AAAI Press / The MIT Press, 2000.

[KC04] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. Technical report, World Wide Web Consortium (W3C), February 2004. Internet: http://www.w3.org/TR/rdf-concepts/.

[KF01] M. Klein and D. Fensel. Ontology versioning for the semantic web, 2001.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the Association for Computing Machinery*, 1995.

[KLWZ03] Oliver Kutz, Carsten Lutz, Frank Wolter, and Michael Zakharyaschev. E-connections of description logics. In *Description Logics Workshop, CEUR-WS Vol 81*, 2003.

[Kre98] R. Kremer. Visual Languages for Knowledge Representation. In *Proc. of 11th Workshop on Knowledge Acquisition, Modeling and Management (KAW'98)*, Voyager Inn, Banff, Alberta, Canada, April 1998. Morgan Kaufmann. `http://ksi.cpsc.ucalgary.ca/KAW/KAW98/kremer/`.

[LM01] O. Lassila and D. McGuinness. The role of frame-based representation on the semantic web. KSL Tech Report Number KSL-01-02, 2001.

2006–2007 © Copyright lies with the respective authors and their institutions.

[LTGP04] A. Lozano-Tello and A. Gomez-Perez. ONTOMETRIC: A Method to Choose the Appropriate Ontology. *Journal of Database Management*, 15(2), 2004.

[McG02] D. L. McGuinness. Ontologies Come of Age. In *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*. MIT Press, 2002.

[MKW04] S. J. Mellor, S. Kendall, and A. Uhl D. Weise. *MDA Distilled*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[MSS04] B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with Rules. In *International Semantic Web Conference*, pages 549–563, 2004.

[Mv03] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. Technical report, World Wide Web Consortium (W3C), August 2003. Internet: http://www.w3.org/TR/owl-features/.

[Obj06] Object Management Group. Meta Object Facility (MOF) Core Specification. Technical report, Object Management Group (OMG), January 2006. `http://www.omg.org/docs/formal/06-01-01.pdf`.

[Ont06] Ontoprise. F-Logic-Programs. Technical report, Ontoprise, January 2006.

[Org04] National Information Stadards Organization. Understanding metadata. NISO Press, 2004.

[PBMT05] E. Paslaru Bontas, M. Mochol, and R. Tolksdorf. Case Studies on Ontology Reuse. In *Proceedings of the IKNOW05 International Conference on Knowledge Management*, 2005.

[PM01] H. S. Pinto and J. P. Martins. A methodology for ontology integration. In *Proc. of the International Conf. on Knowledge Capture K-CAP01*, 2001.

[PSHM07] P. F. Patel-Schneider, I. Horrocks, and B. Motik. OWL 1.1 Web Ontology Language - Structural Specification and Functional-Style Syntax. Technical report, World Wide Web Consortium (W3C), February 2007. Internet: http://owl1-1.cs.manchester.ac.uk/syntax.html.

[PSZ06] J.Z. Pan, L. Serafini, and Y. Zhao. Semantic import: An approach for partial ontology reuse. In *Workshop on Modular Ontologies*, 2006.

[RVMS99] T. Russ, A. Valente, R. MacGregor, and W. Swartout. Practical experiences in trading off ontology usability and reusability, 1999.

[SK03] Heiner Stuckenschmidt and Michel C. A. Klein. Integrity and change in modular ontologies. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 900–908, 2003.

[SK04] Heiner Stuckenschmidt and Michel C. A. Klein. Structure-based partitioning of large concept hierarchies. In *International Semantic Web Conference*, pages 289–303, 2004.

[SP04] Evren Sirin and Bijan Parsia. Pellet: An owl dl reasoner. In *Description Logics*, 2004.

[SR06] Julian Seidenberg and Alan Rector. Web ontology segmentation: Analysis, classification and use. In *Proceedings of the World Wide Web Conference (WWW)*, Edinburgh, June 2006.

[SS89] M. Schmidt-Schauss. Subsumption in KL-ONE is Undecidable. In *Proceedings of the First International Conference on the Principles of Knowledge Representation and Reasoning (KR-89)*, pages 421–431. Morgan Kaufmann, Los Altos, 1989.

[SSW05a] L. Serafini, H. Stuckenschmidt, and H. Wache. A Formal Investigation of Mapping Languages for Terminological Knowledge. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence - IJCAI05*, Edinburgh, UK, August 2005.

[SSW05b] Heiner Stuckenschmidt, Luciano Serafini, and Holger Wache. Reasoning about ontology mappings. Technical report, Department for Mathematics and Computer Science, University of Mannheim ; TR-2005-011, 2005.

[ST05] Luciano Serafini and Andrei Tamilin. Drago: Distributed reasoning architecture for the semantic web. In *European Semantic Web Conference - ESWC*, pages 361–376, 2005.

[Stu06] Heiner Stuckenschmidt. Towards multi-viewpoint reasoning with OWL ontologies. In *European Semantic Web Conference*, 2006.

[SU05] H. Stuckenschmidt and M. Uschold. Representation of Semantic Mappings. In Yannis Kalfoglou, Marco Schorlemmer, Amit Sheth, Steffen Staab, and Michael Uschold, editors, *Semantic Interoperability and Integration. Dagstuhl Seminar Proceedings*, volume 04391, Germany, 2005. IBFI, Schloss Dagstuhl.

[TBMM04] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures Second Edition. Technical report, World Wide Web Consortium (W3C), October 2004. Internet: http://www.w3.org/TR/xmlschema-1/.

[TF05] Sergio Tessaris and Enrico Franconi. Rules and queries with ontologies: a unifying logical framework. In *Description Logics*, 2005.

[TV56] A. Tarski and R. Vaught. Arithmetical Extensions of Relational Systems. *Compositio Mathematica*, 13:81–102, 1956.

[UHW+98] M. Uschold, M. Healy, K. Williamson, P. Clark, and S. Woods. Ontology Reuse and Application. In *Proc. of the Int. Conf. on Formal Ontology and Information Systems FOIS98*, 1998.

[Ull88] J. D. Ullman. *Principles of Database & Knowledge-Base Systems Volume 1: Classical Database Systems*. W.H. Freeman & Company, 1988.

[vHPSH01] F. van Harmelen, P. F Patel-Schneider, and I. Horrocks. Reference Description of the DAML+OIL Ontology Markup Language. Technical report, World Wide Web Consortium (W3C), March 2001. Internet: http://www.daml.org/2001/03/reference.html.

[Vol04] R. Volz. *Web Ontology Reasoning with Logic Databases*. Phd thesis, University of Karlsruhe (TH), Karlsruhe, Germany, http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=2004/wiwi/2, February 2004.

[VOM02] R. Volz, D. Oberle, and A. Maedche. Towards a Modularized Semantic Web. In *Semantic Web Workshop*, Hawaii, 2002.

[W3C05a] *Accepted Papers of the W3C Workshop on Rule Languages for Interoperability, 27-28 April 2005, Washington, DC, USA*, 2005. http://www.w3.org/2004/12/rules-ws/accepted.

[W3C05b] W3C. Rule Interchange Format Working Group Charter. `http://www.w3.org/2005/rules/wg/charter`, 2005.

[WW02] Y. Wand and R. Weber. Information Systems and Conceptual Modelling: A Research Agenda. *Information Systems Research*, 13(4), 2002.

2006–2007 © Copyright lies with the respective authors and their institutions.