Zur Erlangung des akademischen Grades eines
Doktors der Wirtschaftswissenschaften (Dr. rer. pol.)
von der Fakultät für Wirtschaftswissenschaften
der Universität Karlsruhe (TH)
genehmigte Dissertation.

Metamodel-based Knowledge Representation

von

Licentiaat Informatica Sara Brockmans

Tag der mündlichen Prüfung: 26.07.2007

Referent: Prof. Dr. Rudi Studer
Koreferent: Prof. Dr. Andreas Geyer-Schulz
Prüfer: Prof. Dr. Andreas Oberweis

2007 Karlsruhe

Abstract

In this thesis, we investigate how the engineering of ontologies can be supported by successful methodologies and technologies from software engineering. Thereby, our goal is to support business partners in representing and exchanging their company's knowledge in the form of ontologies.

To facilitate ontology development, we introduce a solution based on the Meta Object Facility (MOF), a metamodeling framework that is successfully used to enable the development and interoperability of model and metadata driven software systems. First, we provide MOF-based metamodels for the ontology languages OWL, SWRL, F-Logic, and for OWL ontology mapping languages. We defined the metamodels in MOF, hence they can be applied in the Model Driven Architecture (MDA) framework. The metamodels reflect the structure of the ontology languages, whereas their instances are referred to as models, namely concrete ontologies and rules. So called language mappings give the semantics of the metamodels by translating the terms of the metamodels to those of the respective formalisms. Second, to ease ontology development, we suggest a UML profile as a visual syntax for OWL ontologies as well as for SWRL rules and OWL ontology mappings. Based on the metamodel, we provide a prototypical implementation of the UML profile and show the practical applicability of our approach through a case study evaluation in the pharmaceutical domain. Since a large array of industrial strength tools is available for UML and MOF, companies are enabled to benefit from ontologies. This facilitates the adoption of semantic technologies and their success in real-life applications.

Our work is a first step to bring the W3C vision of a Semantic Web technology and the OMG vision of a MDA together. While the primary purpose of this work is to enable ontology development and maintenance with MDA technologies and tools, it also serves as a basis for potential future developments that can be created by the flow of capabilities of the semantic web into the software development environment.

Contents

List of Figures

List of Tables

1. Introduction

1.1. Context and Motivation

Ontologies and accompanying rules are an important means for enabling better collaboration and improved communication by enabling machine-understandable semantics of data [CPL^{+}05, BMW^{+}07]. For companies, streamlining the exchange of business information means reduced costs as well as other gains in order processing time, product cycle time, time-to-market, and customer satisfaction. The evidence of these facts brings an increasing number of companies to representing or transforming their knowledge into shared models in the form of (rule-extended) ontologies, for data integration in any kind of application[1]. However, many employees are not familiar with ontology formalisms and need support in building their ontology.

Furthermore, market participants deciding to cooperate and exchange ontologies need them to be aligned. Their ontology formalisms can be the same or, as in most cases, they can be different and then have to be transformed to a common formalism. But even when both trading partners are already using the same ontology language, or when their formalisms have been aligned, typically the ontologies themselves do not directly match, and ontology mappings need to be defined to be able to align them fully.

As a prominent example for this situation, we look at the pharmaceutical sector. More specifically, we examine the cooperation of several Spanish laboratories called Pharmainnova, which has the objective of improving the commercial relations between labs and their customers and suppliers. To decrease time and cost, the consortium aims at introducing electronic interchanges, whereas electronic invoicing is a first axis of technology improvement they aim at. The main steps to introduce the electronic exchange of invoices are secured sending of digital invoices over internet, reception and storage of electronic invoices, connecting the electronic invoice platform to the companies' systems, and the management of the complete invoice cycle. This project involves several thousands of laboratories and their customers and suppliers. First improvements in introducing electronic invoicing were mainly closed systems that were based on a specific data format. Because of the closed system and high implementation costs, these solutions were not frequently used.

Fundamentally problematic in cases such as this use case is the heterogeneity of their

[1]Note that ontologies on its own are very expressive but still not expressive enough for what is needed in most companies. Therefore a rule extension is essential.

models as well as of the representation languages. Currently, technologies do not allow to build generic solutions which can automatically process any type of model or language. Two main possible solutions are applied for electronic invoicing. Some associations agree to define a common invoicing infrastructure in terms of shared platforms, invoice formats, invoice structures and processes. In other cases, a common invoicing infrastructure is identified and for each partner a specific plug-in is implemented to automate invoice exchange. The Pharmainnova partners decided to combine these options by defining a common invoice model but keeping the internal infrastructure of the partners. To handle the heterogeneity of the invoice models, they apply ontologies with which the semantics of invoicing is defined and maintained. From each partner's invoice model a separate ontology is built[2] and consequently mapped to one central reference ontology provided by Pharmainnova. Hence, although most or all partners have different invoice models, the partners do not need to change the invoice model nor the system they are using.

Also cases with just two partners each having an ontology and wanting to cooperate, bring the same issues: partners need to transform their ontology into the same language, and the ontology models itself need to be aligned. Additionally, users need support in building ontologies and ontology mappings.

The following section puts these issues into research questions, and answers the questions with a compact description of our contributions.

1.2. Research Questions and Contributions

When experiencing the issues arising with ontology engineering, and identifying the successful methodologies and technologies in the area of software engineering, an interesting central research question comes up:

Central Question *How can ontology engineering be supported using existing software engineering methodologies and technologies?*

Throughout our work, we divide this central question into two main questions. Our first main question aims at investigating how to rely on success stories in the software engineering field for supporting the use of different ontology languages by different cooperating partners:

Main Question I *How can existing approaches in software engineering be applied to support the use of different ontology languages by different partners?*

Not all partners that want to cooperate utilize the same ontology language and hence their ontologies need to be aligned. We looked at the ontology formalisms currently available to decide on which languages we want to support in our work. In doing so, we chose to support

[2]In case the company already has an ontology available but in a different ontology language than the one used by Pharmainnova, the ontology is transformed into the common language.

the most common languages, taking into account both standardized ontology languages and de facto standards, and so try to support as many users as possible. We want to allow automatic transformations of models in different languages, as a solution to the use of different languages by different partners. That way, partners can incorporate an aligning facility into or between their existing systems without having to worry about the use of a different ontology language. Additionally, when looking ahead and realizing that the range of available ontology languages will be expanding for a while, we aim at an extensible solution.

Thus, to achieve a solution for the first main question, we asked ourselves the following questions:

Question I.1 *Which are the ontology languages to be supported?*

Question I.2 *How can models defined in one language be automatically transformed to another language?*

Question I.3 *How can an extensible solution be provided, supporting the current semantic web area in which it is not in every case clear yet which language will be the (de facto) standard in the future?*

Although the Web Ontology Language (OWL, [Mv03, PSHM07]) is well established as the standard ontology language for representing knowledge on the web, at the same time, the rule language F-Logic has shown its practical applicability in industrial environments. We decided to focus on these two languages with respect to the model transformations.

As an answer on the second and the third question, we profit from the highly extensible Model Driven Architecture framework (MDA, [MKW04, Fra03]) and its Meta Object Facility (MOF, [Obj06, Fra03]) which are very successful in software engineering applications. We provide a metamodel built on MOF supporting rule-extended ontologies in OWL and the Semantic Web Rule Language (SWRL, [HPSB+04])[3], as well as a metamodel built on MOF supporting rule-extended ontologies in F-Logic [KLW95].

The general idea of using a MOF metamodel based approach is depicted in Figure 1.1: A metamodel for ontologies and rules is defined in MOF[4], i.e., it is defined in terms of the MOF meta-metamodel. Constraints in the Object Constraint Language (OCL, [WK04]) make the metamodel more precise. A metamodel refers to a model of a language, whereas the instances of the metamodel are referred to as models. In our case, models thus refer to actual ontologies and rules. Language mappings from the metamodel define the relationship with the particular formalisms by translating the terms of the metamodel to those of the formalisms, and so provide the semantics for the metamodel.

As MOF metamodels are all defined in a standardized way (using the MOF concepts),

[3]Note that SWRL is the rule language built on top of OWL.

[4]Note that, although the figure only shows one metamodel, we do provide two metamodels: one for OWL and SWRL, one for F-Logic.

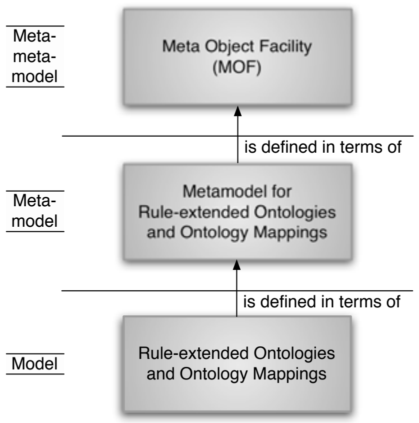

Meta-meta-model
Meta Object Facility (MOF)
is defined in terms of
Meta-model
Metamodel for Rule-extended Ontologies and Ontology Mappings
is defined in terms of
Model
Rule-extended Ontologies and Ontology Mappings

Figure 1.1.: The ontology model as a MOF-metamodel

transformations can be defined between the metamodels and hence automatic model transformations can be provided from F-Logic on the one hand to OWL and SWRL on the other hand, and vice versa. Hence, our work provides a step towards the bridge between F-Logic and OWL.

The second main question we address in our work looks for succesful modeling paradigms in the software engineering field that can be utilized for modeling ontologies, rules and ontology mappings:

Main Question II *How can existing approaches in software engineering be applied to support the modeling of ontologies, rules and ontology mappings?*

Many users are not familiar with logic, or just with textual ontology languages. Even when they are somehow familiar with it, manual work is known to cause more errors as the user is not prevented from writing incorrect statements. Although some approaches already exist to assist the user in modeling ontologies, the user still needs more abstraction from specific aspects of ontology languages. Additionally, it is beneficial to use a paradigm with which the user is familiar, or which is shown to be intuitive.

We divide our second main question into the following partial questions:

Question II.1 *Which requirements do users have that are not familiar with textual syntaxes of ontology languages?*

Question II.2 *How can manual work be avoided, resulting in fewer errors and hence improving overall quality?*

Question II.3 *How can we rely on modeling paradigms with which users are generally familiar?*

Question II.4 *How can shortcomings from existing approaches for ontology engineering be considered?*

Question II.5 *How can be abstracted away from specific ontology language aspects when modeling ontologies?*

As an answer to these questions, we provide a common visual syntax for rule-extended ontologies and ontology mappings, based on the standard Unified Modeling Language (UML, [Fow03, Gro05b]). A UML profile[5] on top of the metamodel for OWL and SWRL[6] and an additional metamodel extension for OWL ontology mappings can employ the extensibility features of UML to allow a visual notation for the modeling of ontologies, especially for people familiar with UML. Leveraging the well-known UML for this purpose is a step towards a model driven approach for modeling and implementing ontologies. The validity of UML models as instances of the metamodel is ensured through the metamodel's OCL constraints. The user is guided in visual modeling in the sense that only valid models can be built. Relying on the standard language UML allows us to make use of existing tool-support as well as of available user experience.

In the answer on the first main question we do not provide a metamodel for OWL ontology mappings. This extension for the metamodel for OWL and SWRL is new and, for lack of any standardized or commonly accepted formalisms, we provide a common metamodel covering the existing OWL ontology mapping formalisms. Constraint extensions provide specific support for the two most well-known formalisms C-OWL [BGvH$^+$03] and DL-Safe Mappings [HM05]. Also for this ontology mapping extension for the metamodel, we provide language mappings defining the relation between the metamodel and the languages. Figure 1.2 shows an overview of the OWL metamodel and its extensions. With 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 ontologies that we support, rules and mappings, 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 the figure demonstrates. MOF allows to add additional modules extending the OWL metamodel when desired. We provide the developed solution as a prototype implementation, on which we conducted an evaluation based on our use case. In this evaluation, the end-users themselves were the participants.

Most of our work has been published before in a number of conference papers. [BVEL04], [BHHS06], [BGSSH06], [BHS06a] and [BHS06b] refer to the most relevant publications for our contributions.

[5]UML profiles, discussed in Section 2.1.3 starting on page 24, are extensions of the UML language to adapt it for specific domains or applications.

[6]Note that we focus on OWL, as decided in answering the first main question.

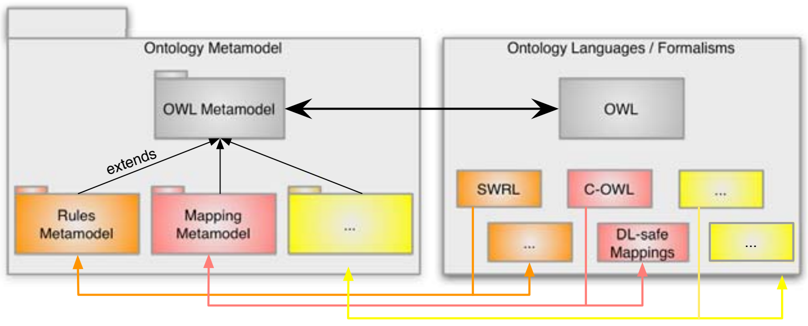


Figure 1.2.: The ontology metamodel and possible language mappings

1.3. Benefits and Overview of the Approach

For easier and more intuitive modeling of ontologies, rules and ontology mappings, we rely on a visual modeling paradigm. Visual syntaxes have shown to bring many benefits that simplify conceptual modeling [Sch02]. In particular visual modeling of ontologies decreases syntactic and semantic errors and increases readability. It makes the modeling and use of ontologies much easier and faster, especially if tools are user-friendly and appropriate modeling languages are applied. The usefulness of a visual syntax for modeling languages has been shown in practice; visual modeling paradigms such as the Entity Relationship model (ER, [Che76]) or UML are used frequently for the purpose of conceptual modeling and software engineering. Consequently, the necessity of a visual syntax for KR languages has been argued frequently in the past [Gai91, Kre98]. Particular representation formalisms such as conceptual graphs [Sow92] or Topic Maps [BBN99], for example, are based on well-defined graphical notations. The absence of a dedicated visual syntax for ontologies[7] has lead to several proposals. [Gai91] proposed a particular visual notation for the CLASSIC description logic. Newer developments have abandoned the idea of a proprietary syntax and proposed to rely on UML class diagrams. [CP99] suggested to directly use UML as an ontology language, whereas [BKK$^+$01] proposed to predefine several stereotypes such that a more detailed mapping from UML to the primitives offered by the DAML+OIL description logic can be achieved. The latter further argues that the UML metamodel should be extended with elements such as property and restriction such that UML is more compatible with KR languages such as OWL.

UML methodology, tools and technology seem to be a feasible approach for supporting the development and maintenance of ontologies together with their rules and mappings. However, an ontology can not be sufficiently represented in UML [HEC$^+$04] and a dedicated visual ontology modeling language is needed. Both representations share a set of core functionalities such as the ability to define classes, class relationships, and relationship

[7]Which can be seen as a direct result of the criticisms about the semantics of early diagrammatic semantic networks [Woo75, Bra79].

cardinalities. But despite this overlap, there are many features which can only be expressed in an ontology language, and others which can only be expressed in UML. Examples for these differences in expressiveness are transitive and symmetric properties in OWL or methods in UML. For a full account of the conceptual differences we refer the reader to [Gro06b]. We rely on UML as a visual approach to our goal.

We need several language transformations, from the visual UML syntax to the logical ontology languages as well as between different ontology languages, and take advantage of the metamodeling features of the MDA. MDA provides the means for the specification and integration of modeling languages in a standardized, platform independent manner. MOF provides the language for creating models of modeling languages, called metamodels, whereas UML defines the language for creating models corresponding to such metamodels. 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. Self-defined 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.

Figure 1.3 depicts an overview of our approach for transforming ontology models in different languages and shows how its fits in the MOF architecture.

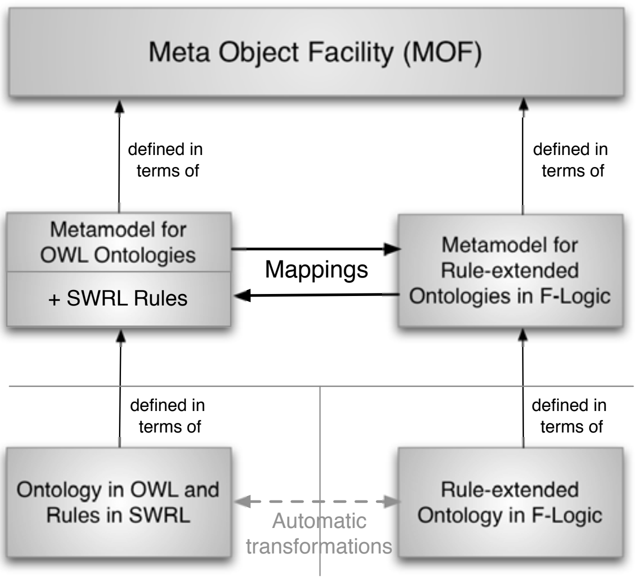

Meta Object Facility (MOF)
defined in terms of
defined in terms of
Metamodel for OWL Ontologies
+ SWRL Rules
Mappings
Metamodel for Rule-extended Ontologies in F-Logic
defined in terms of
defined in terms of
Ontology in OWL and Rules in SWRL
Automatic transformations
Rule-extended Ontology in F-Logic

Figure 1.3.: Overview of the approach for transforming ontology models in different languages

The core of our work is a MOF-based metamodel for OWL ontologies and is displayed left in the figure. The metamodel is grounded in MOF, in the sense that it is defined in terms of the MOF meta-metamodel. Additionally, the core module is extended by a module that provides rule features. The OWL ontology metamodel and its SWRL extension are directly related to the OWL respectively the SWRL language. As is depicted by the box below and its arrow to the metamodel, OWL ontologies and SWRL rules instantiate the metamodel. The constructs of the specific languages have thus a direct correspondence with those of the metamodel.

The right hand side of the figure shows a metamodel for F-Logic. With this metamodel, we provide support for F-Logic as another important ontology language. Ontologies in F-Logic instantiate the metamodel. Based on mappings between the metamodel for F-Logic and the metamodel for OWL and SWRL, automatic structural model transformations between models of both formalisms can be provided. The semantics of the models on both sides of the transformation are defined through the language mapping between the metamodel and the language itself.

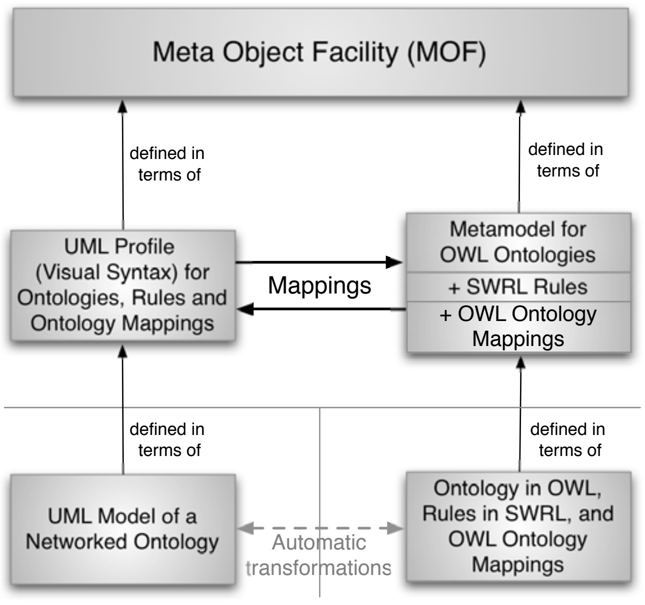

Meta Object Facility (MOF)
defined in terms of
defined in terms of
UML Profile (Visual Syntax) for Ontologies, Rules and Ontology Mappings
Mappings
Metamodel for OWL Ontologies
+ SWRL Rules
+ OWL Ontology Mappings
defined in terms of
defined in terms of
UML Model of a Networked Ontology
Automatic transformations
Ontology in OWL, Rules in SWRL, and OWL Ontology Mappings

Figure 1.4.: Overview of the approach for a visual syntax for modeling ontologies, rules and ontology mappings

Figure 1.4 presents the approach for providing a visual syntax for modeling ontologies, rules and ontology mappings.

An additional extension for the metamodel for OWL and SWRL provides features of ontology mappings. In many application scenarios, only particular aspects are needed and only the relevant modules need to be supported and used. However, the two extensions for rules and mappings build on the core metamodel, and the extension for ontology mappings additionally builds on the SWRL extension, thus they can not stand on their own. Since no consensus exists yet on which OWL mapping language is most suitable, the ontology mapping extension has a generic character in that it is formalism independent and allows a language mapping to different OWL ontology mapping languages. Extensions in the form of OCL constraints can specialize the mapping metamodel for specific languages.

On the left, the figure presents a UML profile which is also grounded in MOF. Our proposed UML profile defines a visual notation for supporting the specification of ontologies, rules and ontology mappings. The graphical constructs for ontologies and rules are specific for OWL and SWRL. For ontology mappings, our goal is to allow the user to specify the 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 figure shows the mappings that are established in both directions between the metamodel and the profile.

The left box at the bottom depicts that specific UML models instantiate the UML profile. Within the MOF framework, the UML models are translated into definitions based on the mappings between the metamodel and the UML profile. In case of ontology mappings, the decision about a concrete mapping formalism is taken in this translation step, which is after the visual modeling of the ontology mappings. This decision is based on the types of the mappings that were modeled. Using the mapping metamodel and the sets of constraints that are defined on it for specific ontology mapping languages, a so-called *constraint checker* could check to which set of constraints, so to which concrete mapping formalism, a model of mappings conforms.

In our case study, we apply our contributions to support the Pharmainnova partners in building their ontologies and mapping them to the reference ontology. Based on the metamodel and its language mappings, as well as on the UML profile and its mappings to the metamodel, a prototype is implemented with which we conducted two experiments to evaluate our approach on usability for the end user.

1.4. Readers' Guide

Part I of this thesis introduces the foundations for our work. At first, the Model Driven Architecture with its two main components, namely the Unified Modeling Language and the Meta Object Facility, is addressed in Chapter 2 starting on page 15. Chapter 3 starting on page 29, introduces the different ontology, rule and ontology mapping languages.

Part II addresses our own contribution. Chapter 4 begins with the description of the metamodel for ontologies in OWL. Appendix A.1 provides a full account of the metamodel, whereas Appendices A.2 provides the language mappings to OWL. Similarly, Chapter 5 and accompanying Appendices A.3 and A.4 introduce the metamodel extension for SWRL. Next, Chapter 6 describes the metamodel for ontologies in F-Logic and addresses the transformations of models in F-Logic to OWL and SWRL. Appendix A.5 gives a detailed overview of the metamodel and Appendix A.6 again provides the language mappings. As the last metamodel part, Chapter 7 and accompanying Appendix A.7 present a common metamodel extension for OWL ontology mappings as well as constraints defining extensions of the common metamodel for specific mapping formalisms, whereas Appendices A.8 and A.9 provide the language mappings for two specific OWL ontology mapping languages

C-OWL and DL-Safe Mappings. Finally, Chapter 8 illustrates a visual syntax for modeling rule-extended ontologies in OWL and SWRL, as well as OWL ontology mappings. Appendix A.10 presents the relationship between the metamodel and the UML profile.

Part III describes the implementation as well as the evaluation of our case study in Chapter 9, followed by a discussion of related work in Chapter 10. Finally, a conclusion with ideas for future work is provided in Chapter 11.

Part I.

Foundations

2. Model Driven Architecture

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. Two of the fundamental components of MDA that are used in our work, are the Unified Modeling Language and the Meta Object Facility. We introduce them in Section 2.1 respectively 2.2. We conclude with a short summary in Section 2.3 on page 28.

2.1. Unified Modeling Language (UML)

The Unified Modeling Language [Fow03, Gro05b][1] driven by the OMG is a fundamental component of MDA. UML is the unification of many existing object-oriented graphical modeling languages of the late 1980s and the early 1990s. As a graphical notation, UML is independent of the methodology for model design. Regardless of the methodology that one uses, UML can be used to express the results. By now, UML has become a well-established and popular standard helping in designing and describing system and domain models.

We mainly apply UML for a visual syntax for ontologies, rules and ontoloy mappings. Additionally, we present our metamodels using UML to make them more understandable for the reader.

People use UML in three different ways. The first and most common way is to use UML for drawing sketches. It helps to communicate some aspects of a system, either by drawing the UML diagram before writing code or by building a UML diagram from code, and hence ease the understanding of the code. In this way, UML is used as a support in communication

[1]The current version of the language is UML2. When mentioning UML throughout this thesis, we always refer to UML2.

and discussion. It is an informal and dynamic use mostly by drawing on a whiteboard, or in documents where selective communication is more important than complete specifications.

Secondly, UML can be used for 'blueprints', when it rather is completeness that matters. In software engineering, the designer draws a detailed design for the programmer, so that the programmer does not need a lot of time to understand what has to be implemented and with which details. With reverse engineering of blueprint UML diagrams, detailed information could be drawn about existing code, to allow developers to understand the details more easily. Clearly, the use of UML for blueprints requires much more sophisticated tools than for sketches.

Thirdly, another way of using UML is to use it as a programming language [MB02]. When used in this form, the whole system is specified in UML. The diagrams are the code, and they are compiled directly to executable code, which makes the UML diagram become the source code. When using UML as a programming language, especially sophisticated tools are needed.

Orthogonal to these three ways of using UML, one can distinguish two types of modeling with UML: conceptual modeling in general and specific software modeling [Fow03]. When modeling conceptually, a modeler applies UML to describe the concepts of a domain. Software modeling is without any doubt the most familiar specific type of conceptual UML modeling, where the elements in a UML diagram map fairly directly to the elements in the specified software system. It can be noticed that UML supports a broad set of goals people can have when modeling. For all this, UML defines thirteen types of diagrams, which can be divided into two categories: structure diagrams and behavior diagrams.

Structure Diagrams emphasize which things must be in the system or the domain being modeled:

- A *class diagram* describes the entity types (classes) with their features as well as the relationships between them;
- a *component diagram* highlights the structure and the connections of the system's components;
- a *composite structure diagram* displays the runtime decomposition of a class;
- a *deployment diagram* models the run-time architecture of a system;
- an *object diagram* displays example configurations of instances;
- a *package diagram* highlights the compile-time hierarchic structure of the model.

Behavior Diagrams emphasize what must happen in the system being modeled:

- An *activity diagram* describes the procedural and parallel behavior of the system;

- a *state machine diagram* highlights how events change an object over its life;
- a *use case diagram* displays how users interact with a system;
- a *communication diagram* describes the interaction between objects while emphasizing links;
- a *sequence diagram* describes the interaction between objects while emphasizing the sequence of the interactions;
- a *timing diagram* describes the interaction between objects while emphasizing timing;
- an *interaction overview diagram* is a combination of a sequence diagram and an activity diagram.

It is often allowed to use an element from a certain diagram type in another diagram. The UML standard specifies which elements are generally used in certain diagram types, but it is not a prescription. UML has rather descriptive instead of prescriptive rules (at least when not used as programming language). Moreover, UML does not only consist of the graphical notation, but provides a MOF metamodel[2] as well, defining all concepts of the language. From a MDA standpoint, UML has some great strengths. Firstly, it is not one fixed language but allows to define extensions. Furthermore, UML allows to raise the abstraction level for software development and domain modeling.

Of all UML diagram types, the class diagram is the most relevant for our work because of the specific objective to represent domain models. We introduce UML class diagrams in Section 2.1.1. Next, Section 2.1.2 starting on page 22 addresses the possibility to augment UML diagrams with constraints to achieve precision. Finally, Section 2.1.3 starting on page 24 discusses the possibilities to extend UML for specific domains.

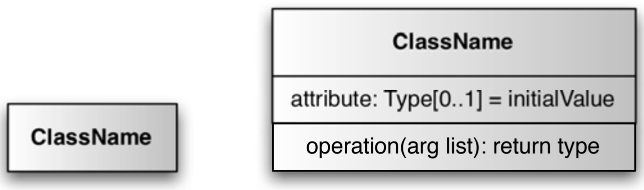
2.1.1. UML Class Diagrams

The class diagram notation is the backbone of UML. This type of static structure diagram describes the structure of a domain by showing the types of objects together with their properties and the relationships between them. Class diagrams display which elements interact but not what happens when an interaction takes place. We introduce the available constructs for UML class diagrams.

Class As one can tell from the name, the central element in UML class diagrams is the class construct. A class represents a type of entity in the domain and encapsulates information about instances of that entity type. Figure 2.1(a) shows how UML depicts a

[2]For this section it suffices to know that a MOF metamodel is a model of a language, specifying the constructs in the language.

class as a box carrying the name of the entity type. Figure 2.1(b) demonstrates optional compartments defining features of the class. Attributes, which appear in one of the two optional compartments, contain the properties, the structural features of the entity type. Each such property contains a name and optionally a type, an indication whether the property is mandatory (cardinality 0 or 1) and an initial value. A "/" foregoing the attribute's name can indicate that the value of the attribute is derived from other attribute values. The other primary optional compartments is not really used in domain models but rather in models of software systems. It defines operations, called methods in programming terms, through their name, optional parameters and the return type. Other compartments may be defined as well, for example to define constraints that are defined on the class. When a class is defining an entity type of which no instances exist, for example when it is only defined to generalize other classes, this is called an *abstract class* which is commonly illustrated by an italicized class name.


ClassName
ClassName
attribute: Type[0..1] = initialValue
operation(arg list): return type

(a) Without compartments (b) With compartments

Figure 2.1.: UML class

The other principal elements in UML class diagrams are relationships. We highlight the primary kinds of relationships in UML class diagrams.

Association Association is probably the best-known type of relationship in UML class diagrams. Although it is often used, it actually means exactly the same thing as an attribute in a class box, being just an alternative way of representation. Usually attributes are used when the attribute value is rather something simple, like a boolean or an integer. Otherwise, it is common to use an association. Additionally, an association can be bidirectional, representing a pair of properties. Figure 2.2 shows that associations are graphically represented as a solid line between the classes. The arrow indicates the direction of the relationship. At the target end of the line, a property name can be shown as the role of the class on the other side. However, this is not mandatory; if a name is needed, for example in a constraint, the target class name with lower case is used. We explained that a "/" can forego an attribute's name in case of a derived attribute value. The same counts for associations with derived property values. An association can also carry specific property-strings between curly brackets to indicate additional properties. A property-string that is often used is {*ordered*}, indicating that the elements at the end of the association have a specific order.

At the beginning as well as at the end of an association, multiplicities can indicate the number of objects that participate in the association. The most common multiplicities, which we present in Figure 2.3, are 1 (exactly one), 0..1 (0 or 1), 1..* (at least one, without upper bound), and * (0 or some, without upper bound).

Lastly, an association class can be defined for adding certain extra features to the association. Figure 2.4 shows how a third class is connected to the middle of the association to represent an association class.

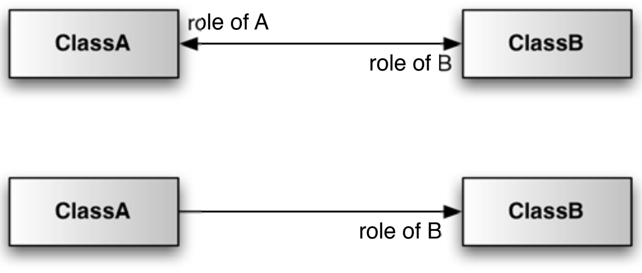


Figure 2.2.: UML association, representing two properties or just one

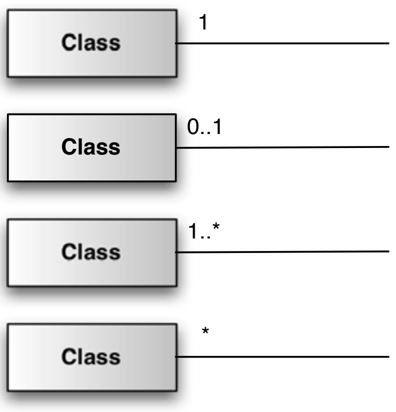


Figure 2.3.: UML multiplicities

Generalization When several entity types have similarities but also differences, this can be represented by generalizing them into a *superclass*. A class which is a special kind of that

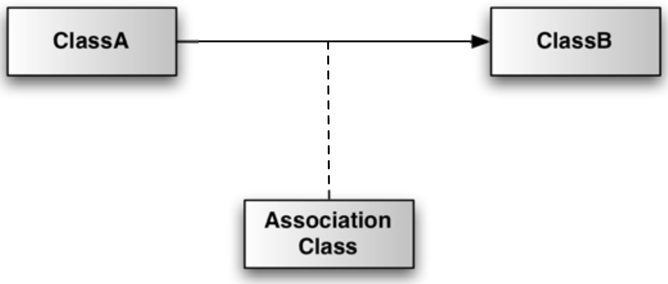


Figure 2.4.: UML association class

class, is called a *subclass* of that class. In software engineering, this setting is known as *inheritance*. In case of such relationship between a superclass and its subclasses, everything which is said about the superclass, also counts for the subclasses (but not the other way around). Any instance of a subclass is an instance of the superclass, too. Figure 2.5 demonstrates how this relationship in UML class diagrams is depicted by a clear triangle shape at the side of the superclass. A solid line connects the triangle with one or more subclasses.

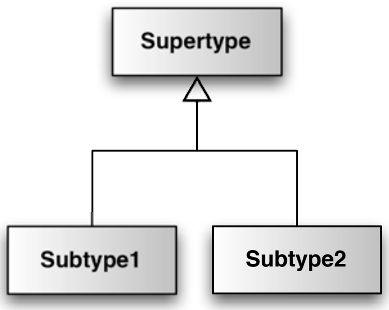


Figure 2.5.: UML generalization

Aggregation A UML aggregation can be seen as an association with a part-of relationship, representing a collection or container. Figure 2.6 shows how aggregations are represented as a clear diamond shape at the side of the containing class.

Composition A composition relationship in UML is similar to an aggregation, but with composition, destroying the container implies destroying the contents too. Consider, for example the relationship of a company to its departments. Both company and department are

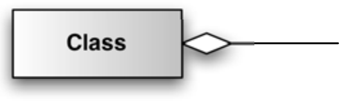


Figure 2.6.: UML aggregation

modeled as classes, and a department cannot exist if the company does not exist. Figure 2.7 demonstrates how a black diamond shape represents a composition relationship in UML.

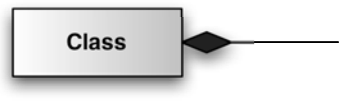


Figure 2.7.: UML composition

Dependency A dependency is a relationship between two elements where changes to one element may cause changes to the other element. Dependencies can be specified between all sorts of UML elements. Many UML relationships essentially imply a dependency, e.g. some navigable associations between classes. Figure 2.8 presents how UML depicts a dependency as a dashed arrow pointing from the dependent to the independent element. Additionally, dependencies can be mutual.

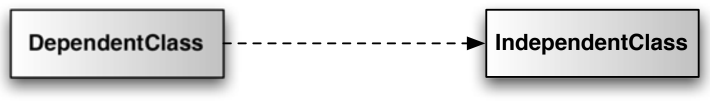


Figure 2.8.: UML dependency

Comment Comments in the form of notes can be added to UML class diagrams, as to any other UML diagram. Figure 2.9 shows how notes are drawn. They can be on their own next to the diagram, or connected to a specific element via a dashed line. In contrast to constraints, which are depicted in curly brace brackets and represent rules that the model must follow, a note is just additional information for reference.

Package UML provides the package construct to allow better readibility of class diagrams, grouping similar or related classes and their relationships. UML packages can be nested,

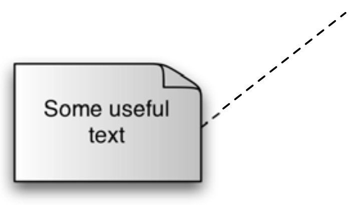


Figure 2.9.: UML comment

and elements belonging to the same package can never carry the same name. Figure 2.10 demonstrates how a tapped folder represents a package, which can additionally be related to other packages, similar to relationships between classes. The access to an element from another package is carried out via its pathname, composed of the name of the other package, a twofold colon and the name of the element.

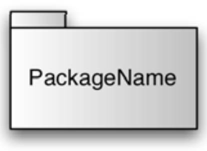


Figure 2.10.: UML package

Object Finally, although strictly seen not belonging to the class diagrams, we represent the UML notation for objects as it is important for our work. Figure 2.11 shows how an object is presented underlined in a class box, together with the optional name of the class to which it belongs.

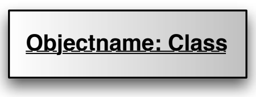


Figure 2.11.: UML object

2.1.2. Object Constraint Language

To add precision to UML diagrams, one can express constraints that must hold for every model. For this, UML allows to use any language, with the only requirement that the constraints must be put inside braces ({}) when they are depicted directly in the UML diagram. A modeler can choose to use natural language or a programming language, or UML's formal Object Constraint Language (OCL, [WK04, Gro06a]) based on predicate calculus. Although the formal OCL notation demands the modeler to be familiar with the language, it lowers the risk of misinterpretation, which could exist in case of ambiguous natural language. In our work, we utilize OCL to specify the constraints on the metamodels.

OCL is a declarative language defining restrictions. Constraints defined in OCL return only a value without causing any side effects (for instance when the constraint is violated). OCL provides some built in types like integers, strings or collections, and types can be constructed from the entities of a UML model. A context definition in front of an OCL expression can specify the model element for which the expression is defined. Typically, UML diagrams express OCL expressions within notes or in the definitions of classes in a UML diagram. We present them separate from the diagrams. Four types of constraints are supported:

- An *invariant* is a constraint that must hold true for all instances of a particular element, e.g. all objects that belong to a given class;
- a *precondition* is a constraint that must hold true when the execution of a particular operation is about to begin;
- a *postcondition* is a constraint that must hold true when the execution of an operation has just ended;
- a *guard* is a constraint that must be true before a state transition fires.

Throughout our work, we only need invariant constraints. Examples 1 and 2 illustrate the use of them.

Example 1 *Imagine an example model containing a class 'Customer' which has an attribute 'age'. The following is a simple attribute invariant constraint which could be defined to restrict the attribute's value:*
context Customer inv :
age ≥ 18

Example 2 *Imagine another example model of a class 'Treatment' which has an association 'includes' to a class 'Medicament', and an association 'helps' to a class 'Person'. Additionally, the class 'Medicament' has an attribute 'hasMinimumAge' and the class 'Person' has an attribute 'hasAge', both having an integer as value. Now, one could define the following invariant constraint on this model expressing that the age of the person using a treatment (an instance of the class 'Treatment'), must not be lower than the minimum age of the medicaments of the treatment:*

context Treatment inv :
helps.hasAge ≥ *includes.hasMimimumAge*

2.1.3. UML Profiling Mechanism

UML has the ability, via the profiling mechanism [Gro05b], to add features so that additional meanings can be adduced, or to confine the language if only specific concepts are to be used. The UML profile mechanism tailors UML to specific application areas, allowing to define UML dialects. By means of general-purpose UML-tools, one can define and use additional or restricted modeling constructs in the form of UML profiles.

The available adaptation constructs for UML profiles are stereotypes. Stereotypes can be attached to any element of the UML vocabulary to indicate that the element is different from other elements of that type. A stereotype consists of a keyword surrounded by guillemets (<< >>). For instance in a UML diagram of a software system, classes can be classified as representing objects of the problem domain (<<entity>>), objects of the user interface (<<boundary>>) or controlling functions in the application (<<control>>).

A stereotype extending the UML metamodel element *Class* is added right above the class name in the class box. For attributes, stereotypes can be prepended to the attribute's declaration in the class box. Additionally, icons can be specified as an alternative notation to represent a stereotyped model element.
Additionally, the definition of a stereotype can include attributes that act as tags. Such tags are written as *tag name* '=' *tagged value*. The modeler adds the values of such tags to instances of the extended UML model elements. For example, for the previously introduced stereotype <<entity>>, the tagged value *baseClass* = *Class* could be defined.

As an alternative to adapting UML via profiles, it can also be extended by adapting its metamodel, since UML is defined via MOF. However, when using the profile mechanism one can benefit from generic UML tools that support UML profiles, whereas in case of the heavyweight extension via the metamodel, a modeler could not easily make use of existing tools. This is clearly the main advantage of the profile extension alternative. However, the disadvantage of UML profiles is that the extension possibilities are restricting, especially in comparison to the heavyweight alternative. For our work, the available extension possibilities of the profile alternative suffice, so we take the big advantage of benefitting from existing generic UML tools.

2.2. Meta Object Facility (MOF)

When describing a universe of discourse in the real world, objects are being *classified* according to their common attributes. Furthermore, when refering 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 classified 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 2.2.1. Next, Section 2.2.2 starting on page 26 introduces the available MOF constructs. Finally, we address the transformation of models and metamodels in MOF in Section 2.2.3 starting on page 27.

2.2.1. 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 2.12.

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

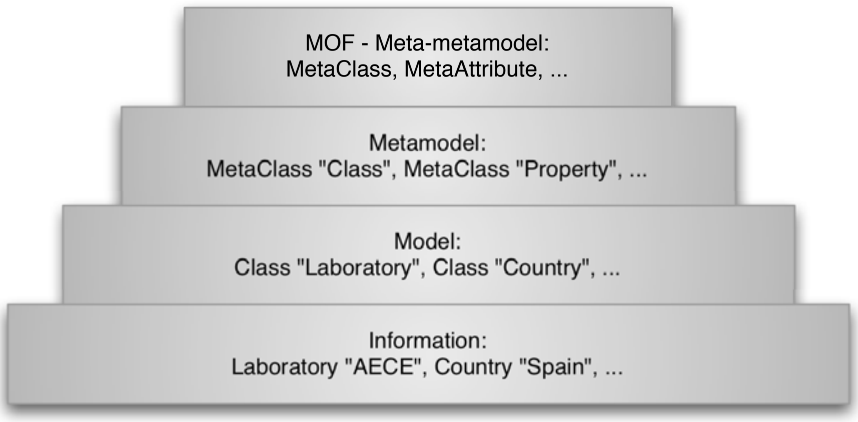

MOF - Meta-metamodel:
MetaClass, MetaAttribute, ...
Metamodel:
MetaClass "Class", MetaClass "Property", ...
Model:
Class "Laboratory", Class "Country", ...
Information:
Laboratory "AECE", Country "Spain", ...

Figure 2.12.: OMG's four-layer metamodel hierarchy

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 2.12 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.

2.2.2. 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 meta-concepts in Figure 2.12 and can define the abstract syntax of a language:

- types (classes, primitive types, and enumerations),
- generalization,
- attributes,

- associations, and
- operations.

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 ([Gro05b], [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.

2.2.3. Model and Metamodel Transformation in MOF

Recall our use case of Chapter 1 where trading partners want to build and exchange ontologies. Whenever they have built an ontology but used another language than their partners, they have to transform their ontology to be able to cooperate. In such cases, an automatic model transformation can reduce the necessary efforts drastically, especially if adapting the internal system is too costly and the company decides to stick to the language being used internally.

For the model transformations to be executed automatically, general mappings that are applicable to all models of a particular language must be defined beforehand. To allow this, one needs to understand the modeling languages in which both models are expressed. Here comes the MDA with its standards for representing models, metamodels as well as metametamodels into play. MOF defines a set of modeling constructs and captures the structure and semantics of arbitrary metamodels. This infrastructure allows the specification of the mappings on the metamodel layer, and in this way the automatic transformations of models. Additionally, MOF brings the benefit that we can not only transform metamodels that we defined ourselves but also other MOF-based metamodels.

A mapping is a collection of rules that define how a particular mapping between two MOF-based metamodels works. As it is defined against the metamodels of the participating models, it applies to all models that conform to the same metamodel and hence is not specific to a single model. Mapping rules specify very specific transformations between elements or combinations of elements in the one (graphical or textual) language to elements or combinations of elements in the other (graphical or textual) language.

In order to automatically execute a model transformation, rules have to be specified in a machine-readable manner, just as models must be formalized so that they can be consumed and produced by a machine. To formalize the rules, many appropriate formalisms exist and the user is free to select a suitable formalism. However, quite recently OMG developed QVT (Query / Views / Transformations, [Gro05a]) especially for MOF, aiming at providing some standards for defining MOF metamodel transformations. Two main approaches exist for specifying mappings, and are defined by QVT. Firstly, specifying mappings in an imperative way consists of defining how to query the data in one metamodel, how to transform it, and how to specify it. Secondly, in a declarative approach it is specified what is to be produced, not how this is done. The QVT language is based on the OCL standard and supports most OCL constructs.

Several QVT implementations already exist. To write our transformations in QVT, we used ATL[3][JK06] which is an open-source Eclipse plugin that implements the QVT and has a large user community and an open source library of transformations. An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models.

2.3. Conclusion

UML is a well-established graphical notation for designing and describing system and domain models. It defines thirteen different diagram types of which the class diagram, describing the structure of a domain, is the most relevant for our work. On top of any UML model, textual constraints in any language, for instance OCL, can be defined to add precision to the model. For specific application areas, UML allows to define and use additional or restricted modeling constructs in the form of UML profiles.
The UML methodology, tools and technology together with its extension mechanism seem to be a feasible approach for supporting the development and maintenance of ontologies. In our work, we apply UML to graphically represent ontologies, rules and ontology mappings, as well as to represent our metamodels graphically.

MOF describes the modeling constructs available for the specification of metamodels. These metamodels describe the modeling constructs of a certain language available for users for the description of a domain (i.e. the creation of models). The fact that all MOF metamodels are defined using MOF and so have a commonly understood meaning, allows to specify mapping functions on the metamodel layer, and hence achieve automatic model transformations. As UML is closely related to MOF, it makes it very suitable as a graphical notation for MOF metamodels. Although no real standard for a textual syntax exists yet, Ecore has emerged as the de facto standard.

[3]http://wiki.eclipse.org/index.php/ATL.

3. 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 and provide a first solution for trading partners using F-Logic to transform their models into OWL.

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 3.1 describes OWL, after which Section 3.2 introduces SWRL. Subsequently, the language F-Logic is discussed in Section 3.3. Finally, OWL ontology mapping languages are addressed in Section 3.4. We conclude with a short discussion on how these languages relate to each other, and how they are supported in our work in Section 3.5.

3.1. 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 thesis, 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 allows to give web content even more semantics than XML [BPSM+06], RDF and RDF Schema [BG04] by supporting additional vocabulary with a formal semantics. We list the different languages from the bottom up:

XML (Extensible Markup Language) provides a syntax for structured data, without imposing semantical constraints.

XML Schema allows to describe the structure of XML documents, and to restrict the content of elements and attributes to specific datatypes.

RDF (Resource Description Framework) provides simple semantics for objects, called resources, and their relationships. RDF models can be represented in an XML syntax.

RDF Schema additionally provides vocabulary for resource properties and classes, as well as semantics for generalization-hierarchies.

OWL (Web Ontology Language) provides even more vocabulary to describe properties and classes. Examples are descriptions of relations between classes, enumerated classes, cardinalities, or property typing.

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]

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.

[1] Some language primitives are shortcuts for combinations of primitives in the logic.

[2] The work in this thesis is based on the OWL 1.1 version available at the end of February, 2007.

[3] Throughout the rest of this thesis, we are talking about OWL 1.1 whenever we refer to OWL.

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].

3.2. 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.

3.3. 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, FHK$^+$97].

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 6.

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.

3.4. 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 [SSW05] 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 7, where we present their aspects along with the introduction of the metamodel.

3.5. Conclusion

The ontology language OWL allows to build, publish and share ontologies. Since its standardization in 2004, it has become the most-accepted ontology language in the semantic web community, and an increasing range of applications became available. By specifying concepts and relationships of a domain formally, computers and software are able to process the data. Although OWL is very expressive, it can not express arbitrary axioms. SWRL is one of the most prominent proposals for an extension of OWL to overcome this restriction, proposing to use Horn-like rules together with OWL axioms.

F-Logic is another language which is applied to build ontologies, although it is intentionally not built for semantics and the web. F-Logic additionally allows to define declarative rules which infer new information from the available information.

The core language of our work is OWL. On top of a (MOF-based) metamodel for OWL, we provide an extension for SWRL. Next to this metamodel, we provide a metamodel for F-Logic to allow automatic transformations between models in the different languages and hence bring the languages more closely together.

To allow interoperability between different heterogenous OWL ontologies, mappings can be defined on top of the ontologies. No de facto standard exists yet for representing such mappings, hence we rely on existing research in which general aspects of OWL ontology mappings were identified. These general aspects cover the different languages C-OWL, DL-forII, OIS and DL-Safe Mappings. We provide an extension of the OWL metamodel for these languages by covering the general aspects. Specific support for two of the languages, C-OWL and DL-Safe Mappings is given through constraints on the metamodel.

In addition, we provide a UML profile as a visual syntax for ontologies in OWL, rules in SWRL and OWL ontology mappings. This profile is based on the metamodel and hence the graphical models can be mapped to the metamodel and so to the languages itself, and vice versa, automatically.

Part II.

Providing Metamodel-based Support for Ontologies

4. Metamodel Descriptions for Ontologies in OWL

This chapter introduces the core of our work, 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 in Section 4.1. To simplify the understanding of the metamodel, we add accompanying UML diagrams to our discussion. Appendix A.1 starting on page 163 provides a full account of the metamodel with a particular description for each metaclass, including an informal description of the element, a listing of attributes, associations and generalizations, as well as a listing of the OCL constraints applicable to that class. We refer the reader to this Appendix for specific details of the metamodel. Additionally, we provide a listing of the mappings between the OWL language constructs and the metamodel in Appendix A.2 starting on page 189. We conclude with a short summary in Section 4.2.

4.1. A MOF-based Metamodel for OWL

This section presents the MOF-based metamodel for OWL 1.1 in eight subsections: Section 4.1.1 starts with ontologies and annotations, after which Section 4.1.2 presents entities and data ranges. Next, Section 4.1.3 demonstrates class descriptions and Section 4.1.4 presents OWL axioms. Then, Section 4.1.5 gives class axioms, after which Section 4.1.6 presents

[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.

object property axioms. Finally, Section 4.1.7 presents data property axioms and Section 4.1.8 presents facts.

4.1.1. 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 3). 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 3 *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."))

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[3]. 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.

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 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:

[3]Note that the '+'-sign in front of a property name denotes its public visibility. Although in our models everything is just public, in software models elements are often characterized as private or protected.

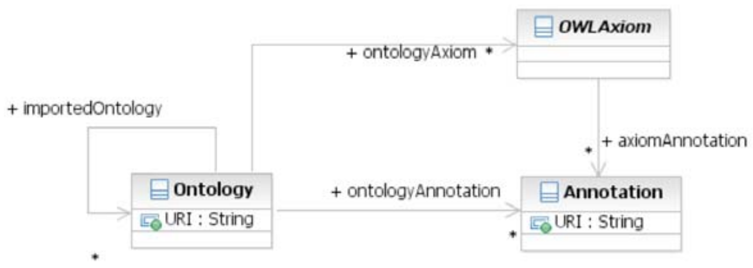


Figure 4.1.: OWL metamodel: ontologies

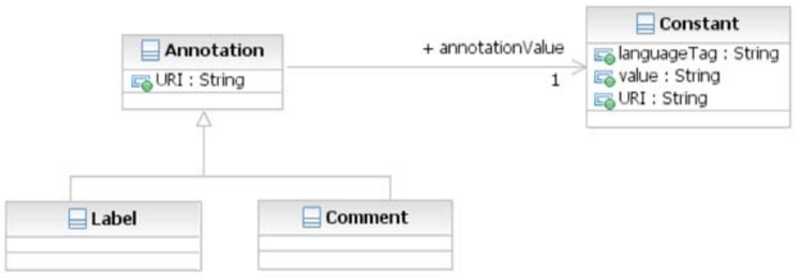


Figure 4.2.: OWL metamodel: annotations

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:
   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* [4]:

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)
   ```

4.1.2. 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[5] (see Example 4), 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 4 *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*[6], *ObjectProperty*, *DataProperty* and *Individual*. An attribute *URI* of the abstract metaclass *OWLEntity* is inherited by all subclasses to identify the entity.

Just like ontologies and axioms, also entities can be annotated (see Example 5). 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.

Example 5 *The following example illustrates the annotation of an entity:*
EntityAnnotation(OWLClass(Medication) Annotation(CreationDate "Created in March 2007."))

[4]Note that the metamodel specifies the attribute *value* as mandatory for any constant. Such cardinalities can be found in the metamodel descriptions in Appendix A.1 starting on page 163.

[5]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).

[6]Note that a simple class, in contrast to the other entities, has the prefix 'OWL' in conformity with the OWL 1.1 specifications.

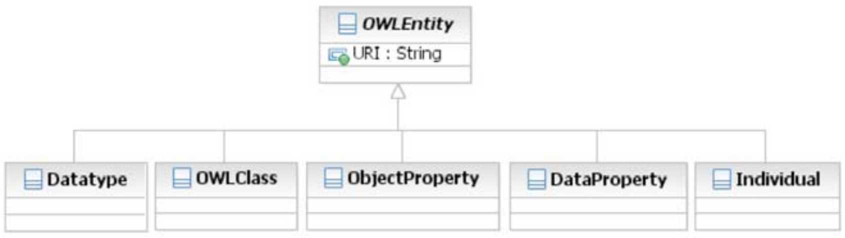


Figure 4.3.: OWL metamodel: entities

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*

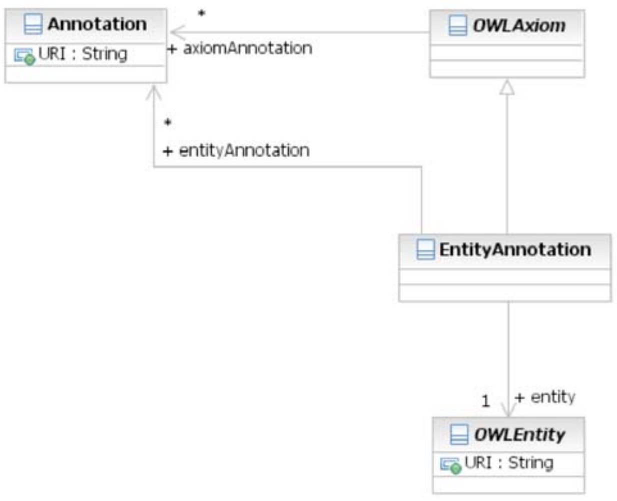


Figure 4.4.: OWL metamodel: entity annotations

to the annotation. As the number of 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'.

Additionally, OWL provides axioms to declare entities (see Example 6). 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

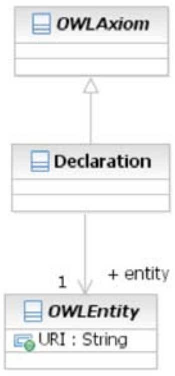


Figure 4.5.: OWL metamodel: declarations

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

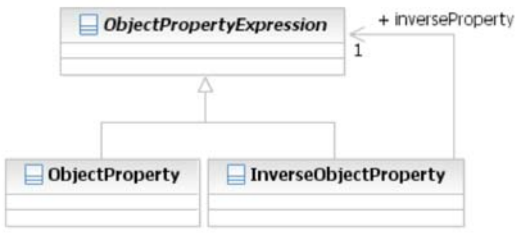


Figure 4.6.: OWL metamodel: object property expressions

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 7). 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 based on any, normal or inverse, object property, hence the association *inverseProperty* from the

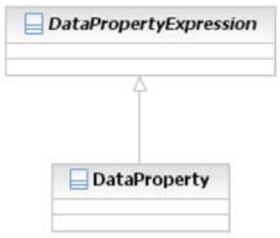


Figure 4.7.: OWL metamodel: data property expressions

metaclass *InverseObjectProperty* is connected to the superclass *ObjectPropertyExpression*.

Example 7 *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.

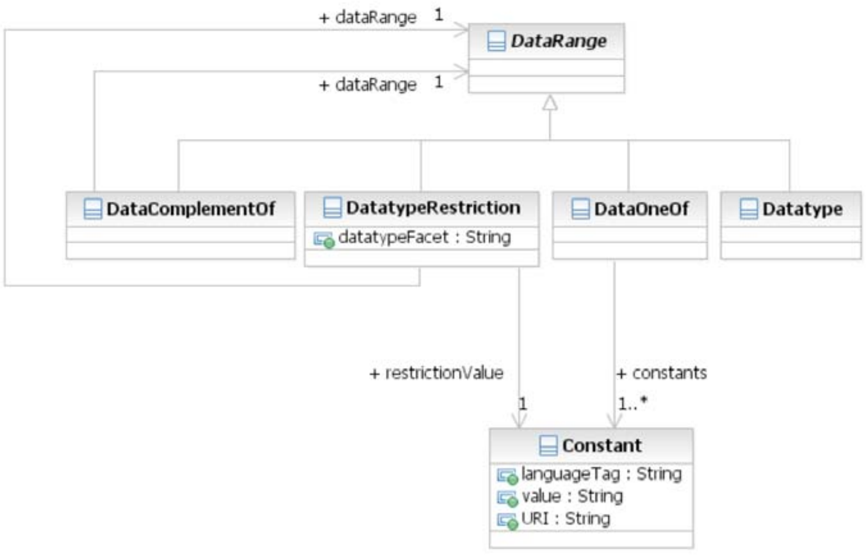


Figure 4.8.: OWL metamodel: data ranges

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'
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 8). The enumerated data values are represented as constants.

Example 8 *The following example illustrates the enumeration of data values:*
DataOneOf(”5”^^xsd:integer ”15”^^xsd:integer)

4.1.3. 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[7]. 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 9). They both have an association called *classes* specifying the involved classes[8]. 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 9 *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))

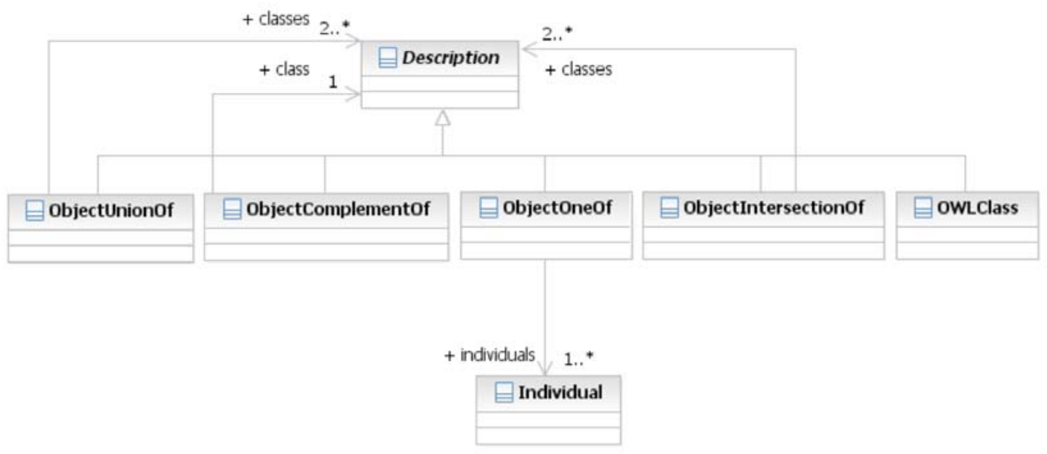


Figure 4.9.: OWL metamodel: propositional connectives

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 set of all objects which have only objects from a certain other class description as value for a specific object property (see Example 10).

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

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

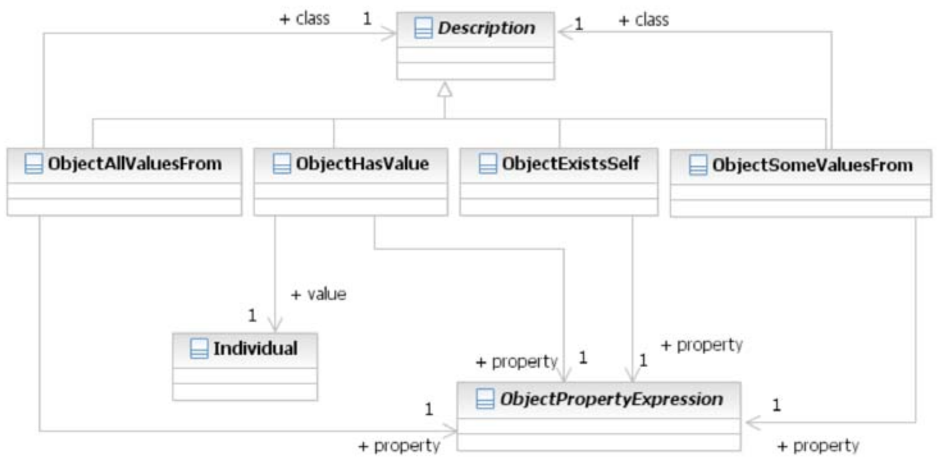


Figure 4.10.: OWL metamodel: object property restrictions

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 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 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 11).

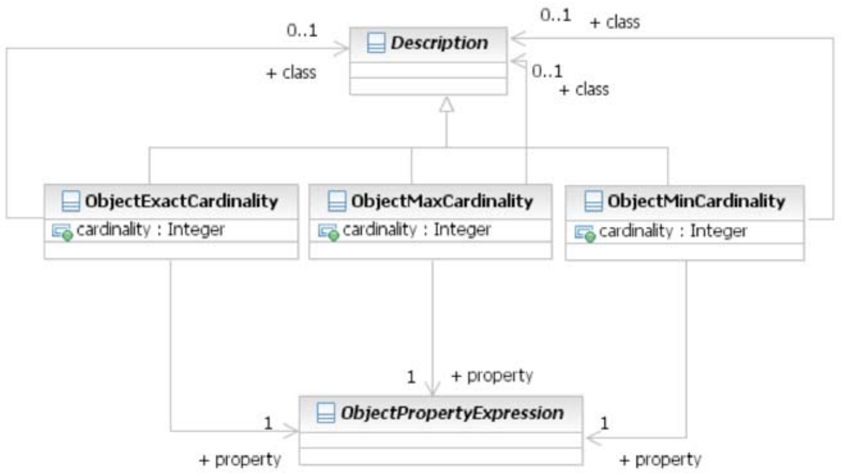


Figure 4.11.: OWL metamodel: object property cardinality restrictions

Example 11 *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 *ObjectExact-Cardinality* defines a class of which all individuals have exactly N different individuals of a certain class as values for the specified object property. 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[9]:

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

2. The cardinality must be nonnegative:
   context ObjectMaxCardinality inv:

[9]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.

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[10].

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 12).

Example 12 *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))

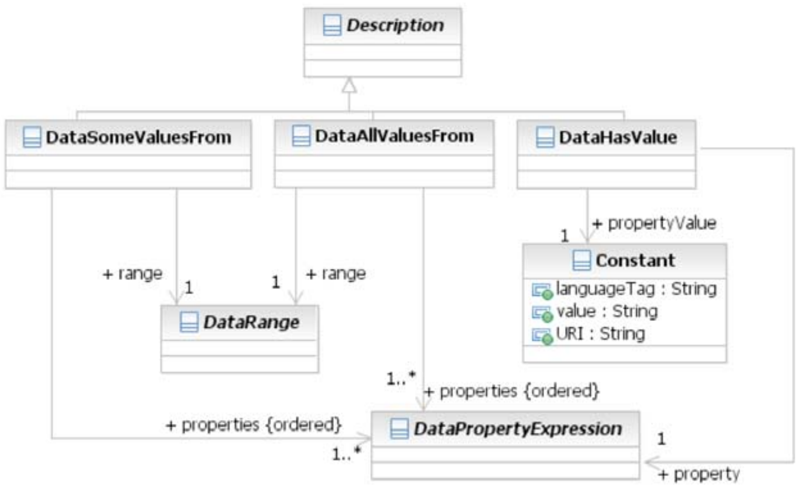


Figure 4.12.: OWL metamodel: data property restrictions

Figures 4.12 represents the metaclasses for these constructs, *DataSomeValuesFrom*, *DataAllValuesFrom* and *DataHasValue*. The meaning of the corresponding constructs is

[10]In the case where it is not explicitly defined, called an unqualified cardinality restriction, the description is owl:Thing.

the same as with these restrictions on object properties. The value of a data property belongs to a data range, represented by the metaclass *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.

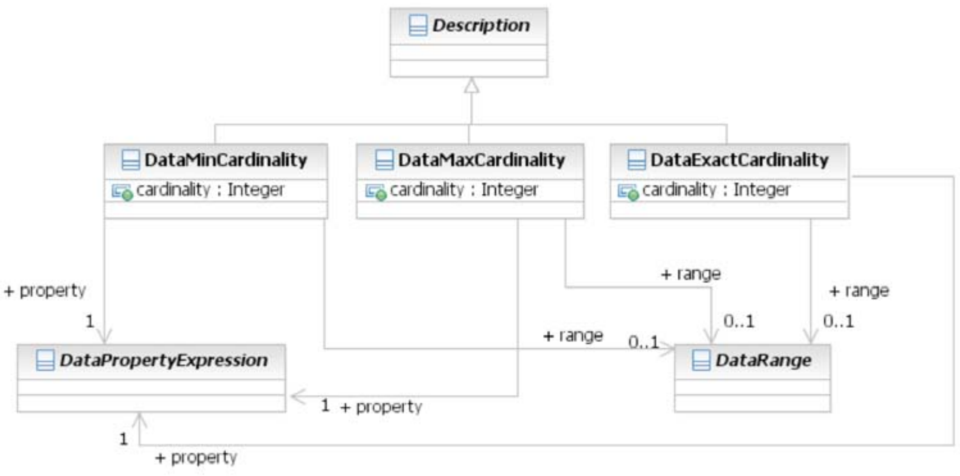


Figure 4.13.: OWL metamodel: data property cardinality restrictions

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

4.1.4. OWL Axioms

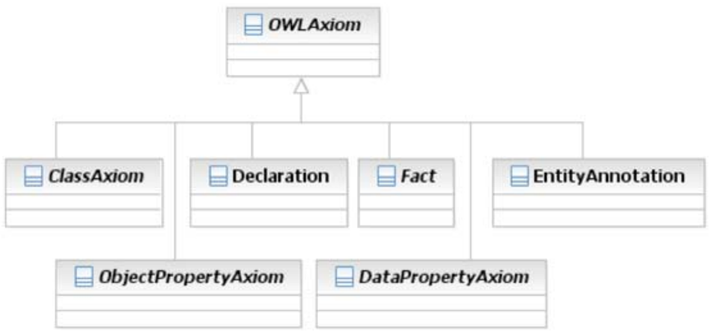


Figure 4.14.: OWL metamodel: axioms

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[11].

4.1.5. 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 13). An association *disjointClasses*, respectively *equivalentClasses* connects the metaclasses to the related class descriptions.

Example 13 *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.

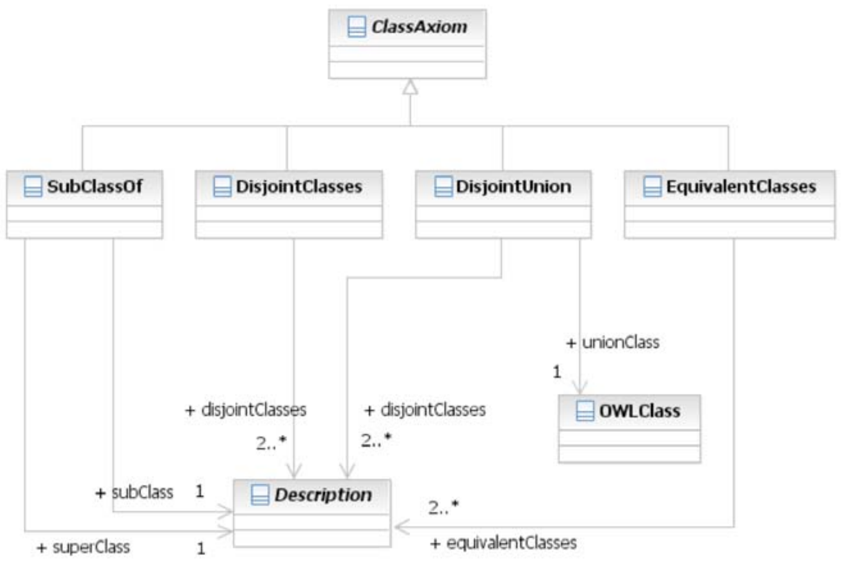


Figure 4.15.: OWL metamodel: class axioms

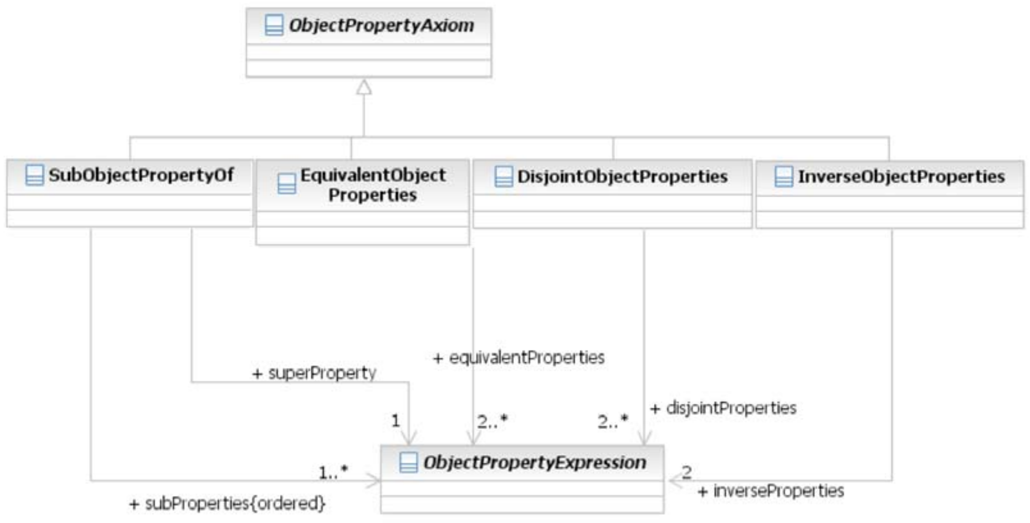


Figure 4.16.: OWL metamodel: object property axioms - part 1

4.1.6. Object Property Axioms

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.

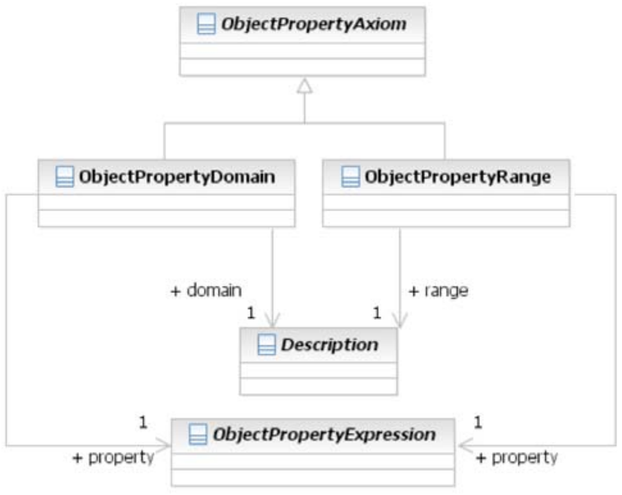


Figure 4.17.: OWL metamodel: object property axioms - part 2

To define the class to which the subjects of an object property belong, OWL provides the

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

domain concept (see Example 14), 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 14 *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, 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.

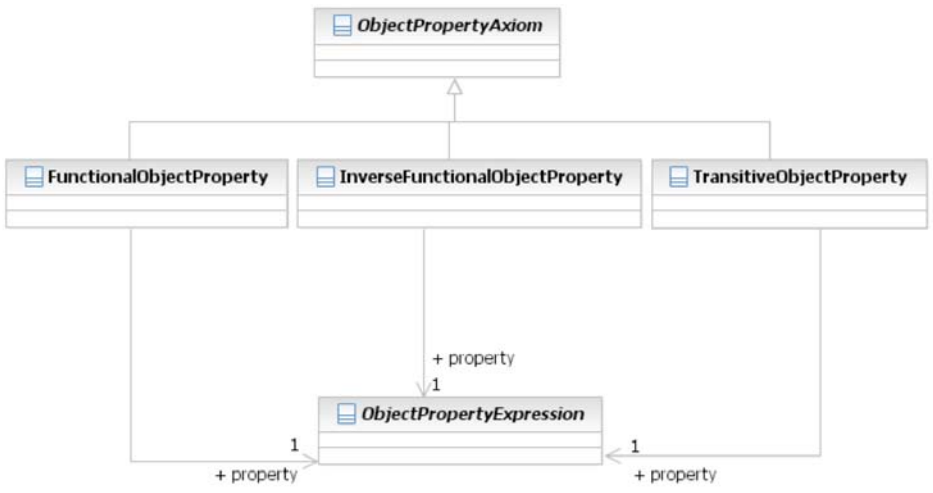


Figure 4.18.: OWL metamodel: object property axioms - part 3

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[12] (see Example 15).

Example 15 *The following example illustrates how an object property is defined to be antisymmetric:*
AntisymmetricObjectProperty(madeFromIngredient)

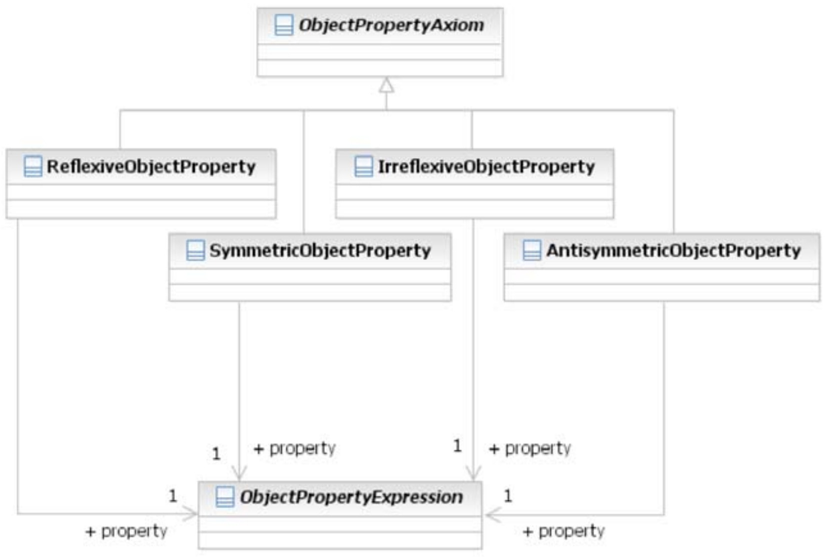


Figure 4.19.: OWL metamodel: object property axioms - part 4

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.

4.1.7. Data Property Axioms

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

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

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

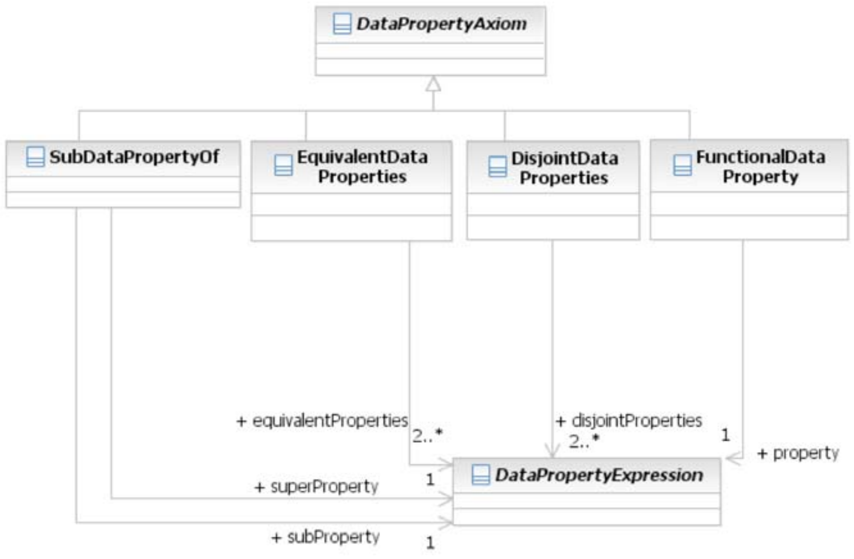


Figure 4.20.: OWL metamodel: data property axioms - part 1

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

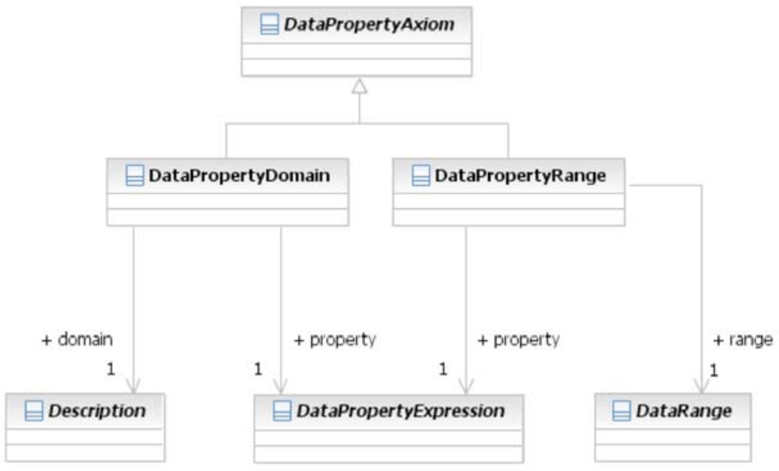


Figure 4.21.: OWL metamodel: data property axioms - part 2

Example 16).

Example 16 *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.

4.1.8. Facts

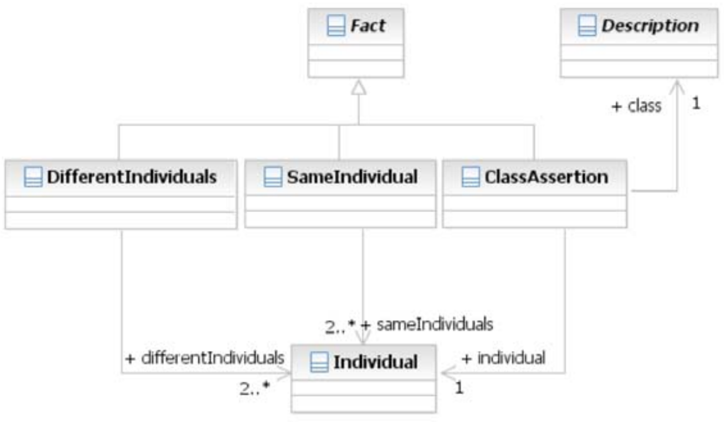


Figure 4.22.: OWL metamodel: facts - part 1

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 17) and is defined in the metamodel as the metaclass *ClassAssertion* with an association *individual* to *Individual* and an association *class* to *Description*.

Example 17 *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 18).

Example 18 *The following example illustrates how the value of an individual under an object property, is defined:*
ObjectPropertyAssertion(madeFromIngredient aspirin acetylsalicylicacid)

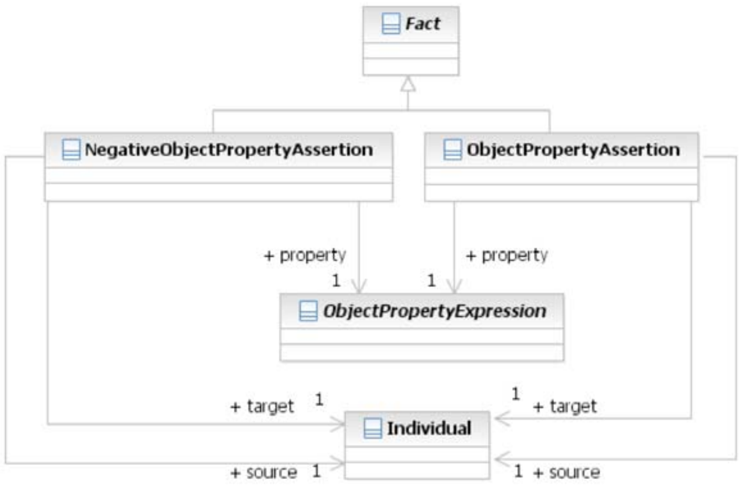


Figure 4.23.: OWL metamodel: facts - part 2

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 *NegativeDat-*

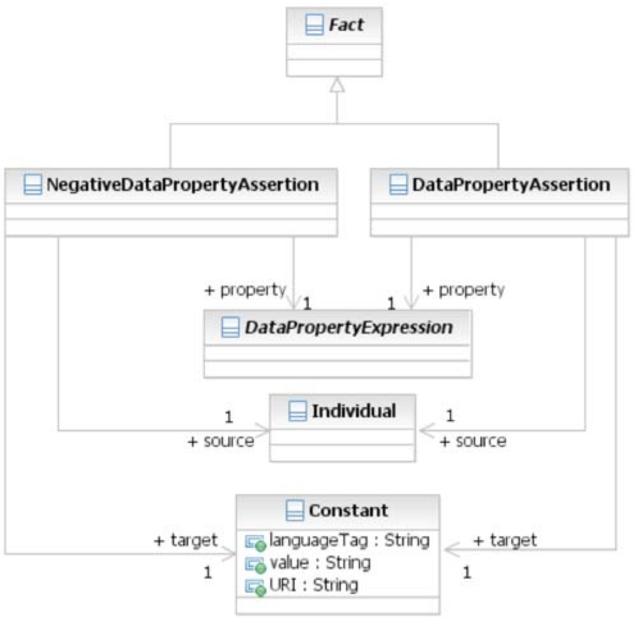


Figure 4.24.: OWL metamodel: facts - part 3

aPropertyAssertion have three associations connecting them to the property, an individual as the subject of the property assertion, and a constant as the property value.

4.2. Conclusion

In this chapter and the accompanying Appendices A.1 starting on page 163 and A.2 starting on page 189, we have presented a MOF-based metamodel for OWL 1.1. and ensured the validity of its instances through various OCL constraints. This metamodel is the first part of our work towards MOF-support for ontology languages. On top of this metamodel, we build a metamodel extension supporting SWRL rules, in Chapter 5 starting on page 59. Next to this metamodel for OWL and SWRL, we present a metamodel for F-Logic in Chapter 6 starting on page 65. Based on these two metamodels, we are able to provide some first possible (automatic) transformations from F-Logic models to OWL models, and vice versa. Moreover, we provide another extension for the metamodel of OWL and SWRL to support mappings between heterogenous OWL ontologies in Chapter 7 starting on page 91. Additionally, directly related to the metamodel we just presented and its extension for OWL mappings, we provide a UML profile in Chapter 8 starting on page 101 as a visual syntax to allow model-driven development of ontologies, rules and ontology mappings.

5. Metamodel Descriptions 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 (see Section 3.2 starting on page 32). 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 in Section 5.1.[1] [2] Appendix A.3 starting on page 196 gives a complete overview with details of the metamodel. In doing so, for every metaclass of the metamodel an informal description, a listing of attributes, associations and generalizations, as well as the constraints, is provided. Additionally, Appendix A.4 starting on page 202 provides the mappings between SWRL and the metamodel. Finally, Section 5.2 starting on page 63 finalizes the Chapter with a short conclusion.

5.1. A MOF-based Metamodel Extension for SWRL

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 39). 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

[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.

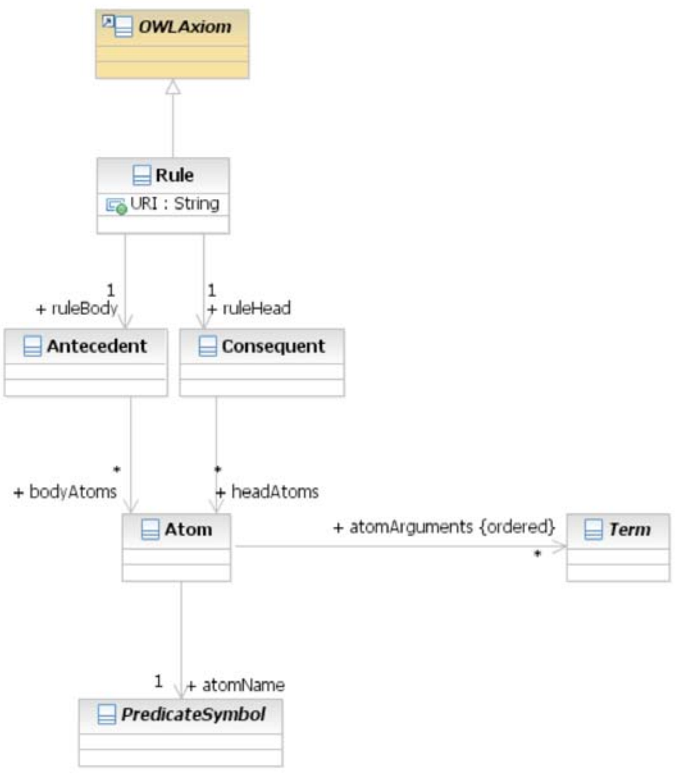


Figure 5.1.: SWRL metamodel extension: rules

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

[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.

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.

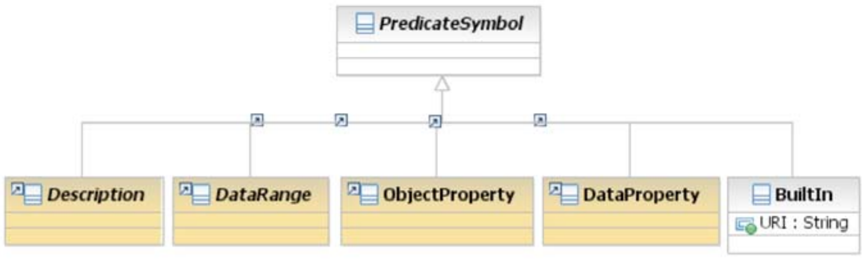


Figure 5.2.: SWRL metamodel extension: predicate symbols

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].

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

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.

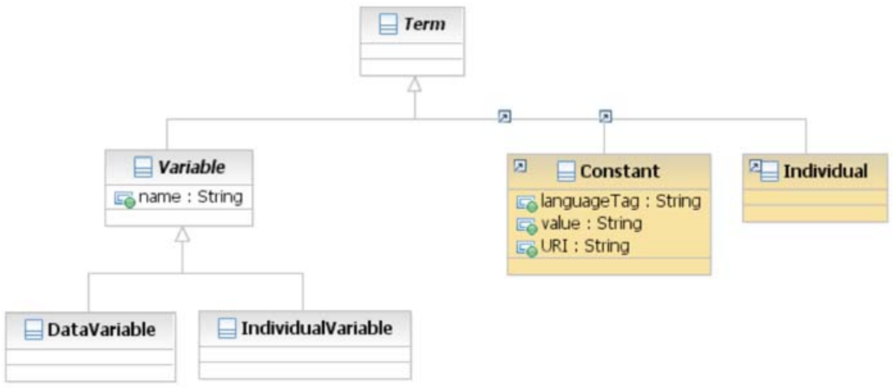


Figure 5.3.: SWRL metamodel extension: terms

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
   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. Conclusion

We introduced an extension for the MOF-based metamodel for OWL to support SWRL rules in this chapter and Appendices A.3 starting on page 196 and A.4 starting on page 202. In the next chapter, after presenting a MOF-based metamodel for rule-extended ontologies in F-

Logic, we show how automatic transformations from F-Logic ontologies to OWL and SWRL could be provided based on the MOF-based metamodels.

Additionally, after introducing an OWL metamodel extension supporting OWL ontology mappings in Chapter 7 starting on page 91, Chapter 8 starting on page 101 introduces a UML profile based on the OWL metamodel and its extensions for rules and mappings.

6. Metamodel Descriptions for Rule-Extended Ontologies in F-Logic

We want to allow automatic transformations between models in F-Logic, and models in OWL and SWRL, and vice versa. We rely on the MDA possibilities to achieve this and define MOF-based metamodels for both formalisms. This chapter presents a metamodel for the language F-Logic as it is described in Section 3.3 starting on page 32. 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 in Section 6.1 with introducing the various metaclasses with their properties and constraints, while explaining the F-Logic constructs they represent[1]. UML diagrams accompany this discussion for the sake of clarity. Appendix A.5 starting on page 205 provides an entire listing of the metaclasses, consisting of an informal description, a listing of all its attributes, associations and generalizations, as well as a listing of the constraints defined on it. The specification of the metamodel is completed with the mapping between the metamodel and F-Logic, presented in Appendix A.6 starting on page 218.

The open world semantic of OWL does not fit together with the F-Logic semantics, and unrestricted transformations between both languages will not be possible. However, possible transformations in a certain extent are desirable. Section 6.2 starting on page 79 addresses transformations between the metamodel for F-Logic, and the metamodel for OWL and SWRL. In doing so, we provide a first step towards automatic transformation of models in F-Logic to OWL and SWRL, and vice versa.

We summarize the chapter in Section 6.3 starting on page 90.

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

6.1. A MOF-based Metamodel for F-Logic

Eight subsections present the metamodel: Section 6.1.1 starts with F-Logic programs, after which Section 6.1.2 presents terms. Next, Section 6.1.3 presents Formulas and Section 6.1.4 presents rules and queries. Then, Section 6.1.5 gives logical connectives and Section 6.1.6 presents logical quantifiers. Finally, Section 6.1.7 demonstrates F-atoms and F-molecules, whereas Section 6.1.8 P-atoms.

6.1.1. 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 6.1 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*.

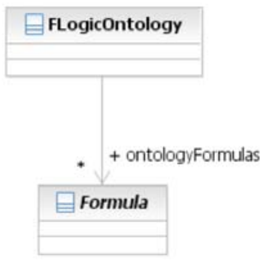


Figure 6.1.: F-Logic metamodel: ontologies

6.1.2. 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.

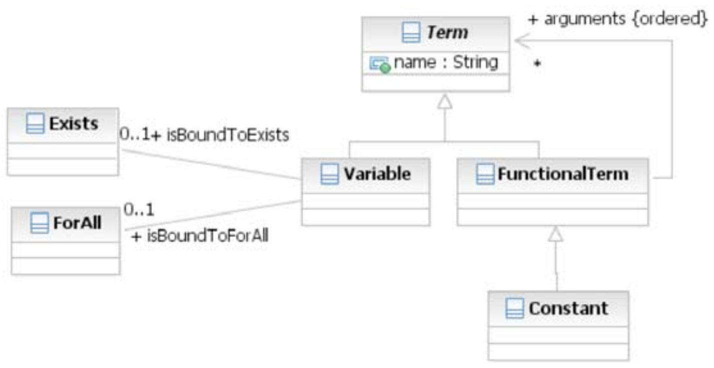


Figure 6.2.: F-Logic metamodel: terms

Figure 6.2 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', 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)

6.1.3. 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 6.3 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.

6.1.4. Rules and Queries

Rules (see Example 19) infer new information from available facts and so extend the object base intensionally. A rule consists of a head (postcondition), an implication sign ('$\longleftarrow$'), and a body (precondition). A rule defines that *when* the formula (or combination of formulas) in

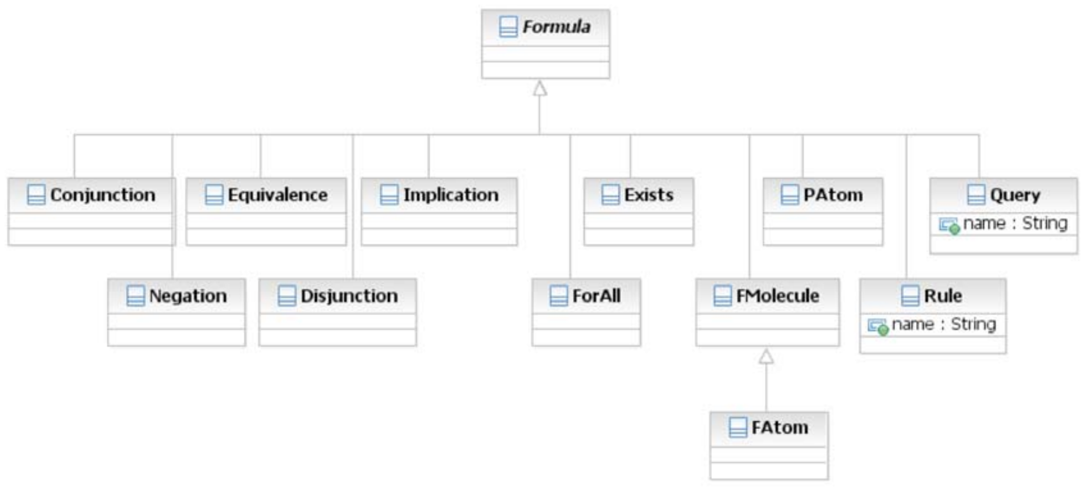


Figure 6.3.: F-Logic metamodel: formulas

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

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 appear anywhere in the rule body.

From the rules and facts in an F-Logic program, a model is computed on which queries (see Example 20) 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 20 *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:
$\forall X, Y, Z \longleftarrow X : \mathsf{Medication}[\mathsf{madeFromIngredient} \rightarrow \mathsf{acetylsalicylicacid}, \mathsf{helps} \rightarrow Y] \wedge Y[\mathsf{hasDisease} \rightarrow Z]$

Figure 6.4 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'[2].

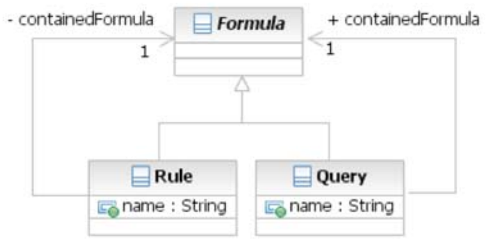


Figure 6.4.: F-Logic metamodel: rules and queries

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)[3]. The following constraint defines this for rules without a universal quantifier:
   context Rule inv:
   self.containedFormula.oclIsTypeOf(Implication) implies

[2]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.

[3]Note that another fact construct in F-Logic, F-atom, is defined as a subclass of *FMolecule* and thus is also included.

self.containedFormula.oclAsType(Implication).consequent.oclIsTypeOf(Conjunction)
or self.containedFormula.oclAsType(Implication).consequent.oclIsTypeOf(FMolecule)
or self.containedFormula.oclAsType(Implication).consequent.oclIsTypeOf(PAtom)

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 specify 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)
   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)

6.1.5. 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 symbol in logic, the serialization in F-Logic, and the representation in the metamodel for the logical connectives in F-Logic:

Logical symbol	**Serialization in F-Logic**	**Metamodel element**
$\wedge$	AND	class *Conjunction*
$\equiv$	$\leftrightarrow$	class *Equivalence*
$\sqsupset$	$\rightarrow$	class *Implication*
$\neg$	NOT	class *Negation*
$\vee$	OR	class *Disjunction*

Figure 6.5 details logical connectives[4]. Each of these subclasses is connected to the class *Formula*, specifying the formulas that are combined. *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

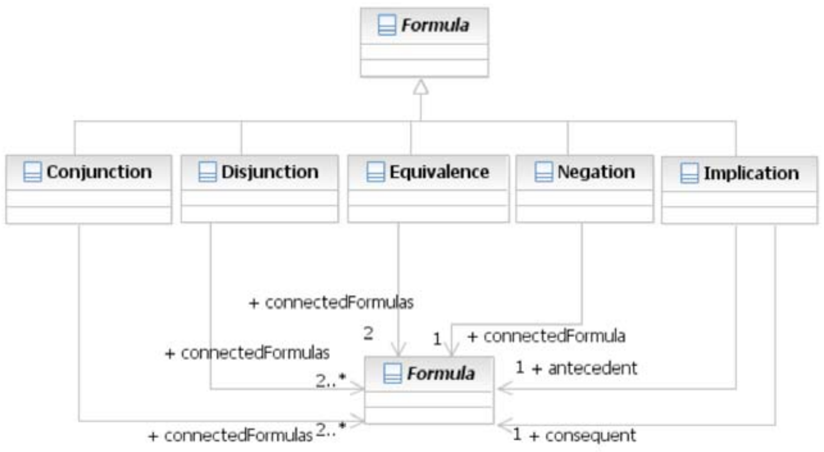


Figure 6.5.: F-Logic metamodel: logical connectives

connectives cannot be used in combining rules or queries:

[4]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.

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

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))

6.1.6. Logical Quantifiers

F-Logic allows the two logical quantifiers ∀ and ∃, binding variables in a formula. Figure 6.6 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 *contained-Formula*, 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 *boundVari-ables* from both to the metaclass *Variable*. To make sure that a variable that is connected to a quantifier through the 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))

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)

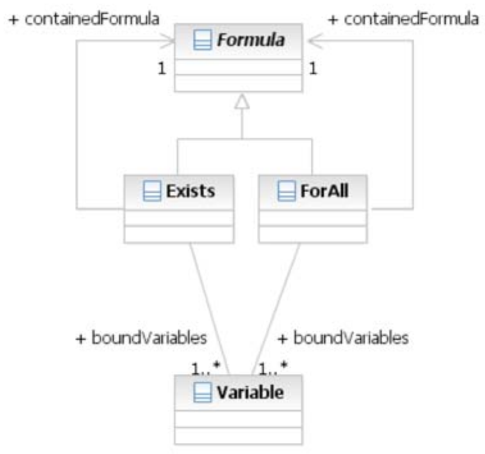


Figure 6.6.: F-Logic metamodel: logical quantifiers

6.1.7. 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[5]. For example

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

is equivalent to

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

Figure 6.7 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:

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

(self.host.oclIsTypeOf(Term) and self.methods→size()=1) or
(not self.host.oclIsTypeOf(Term) and self.methods→size()=0)

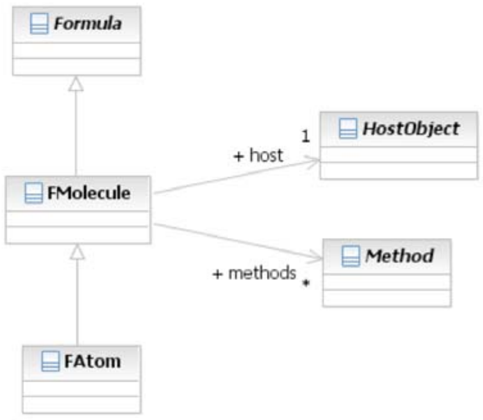


Figure 6.7.: F-Logic metamodel: F-molecules

Figure 6.8 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 6.9 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*: *SingleValuedSignature*, *SingleValuedApplication*, *MultiValuedSignature* and *MultiValuedApplication*. Every type of method has a name, which is represented through the association *name* of their superclass. Additionally, 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 val-

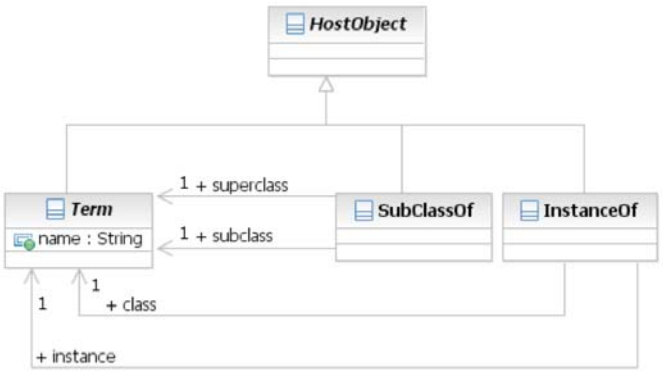


Figure 6.8.: F-Logic metamodel: F-molecule host objects

ues, the association *value* between *MultiValuedApplication* and *MethodValue* has multiplicity 'one to many'[6].

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*.

6.1.8. 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 parameter.

Figure 6.10 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

[6]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.

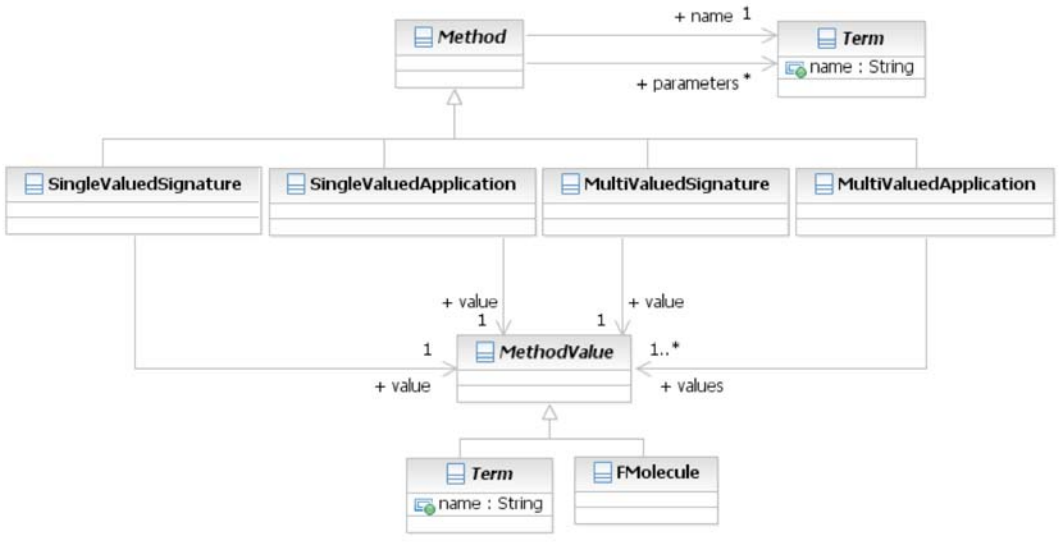


Figure 6.9.: F-Logic metamodel: methods

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

More detail about the F-Logic metamodel can be found in Appendices A.5 starting on page 205 and A.6 starting on page 218.

6.2. Model Transformations for Rule-Extended Ontologies

Recall our use case where trading partners want to build and exchange ontologies. Here, we experience the situation where most companies do not have ontologies yet. However, some companies have already modeled their knowledge as ontologies earlier. As no specific ontology language such as OWL was existing yet, companies often used F-Logic, although it is not intentionally built for modeling ontologies. To let such companies switch to OWL and get full advantage from their ontologies, an automatic model transformation is very beneficial.

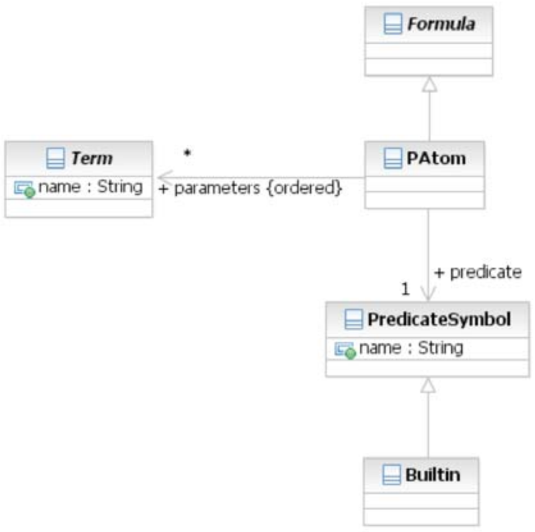


Figure 6.10.: F-Logic metamodel: P-atoms

Transformations can be defined on the level of MOF metamodels to transform models based on the F-Logic or OWL metamodel. To this extent the correspondences between the metamodels must be established.

The semantic correspondence between F-Logic and OWL is heavily being researched [MHRS06, MR07]. Our goal is to provide a structural transformation from the core part of an F-Logic ontology to OWL. This core ontology part consists of the taxonomy, which is the concept hierarchy, and instantiations.

A taxonomy and instantiations in F-Logic are modeled using F-atoms and F-molecules, including class assertions, instance specifications, as well as single- and multivalued methods. As every F-molecule can be split up in several F-atoms, it suffices to define transformations for F-atoms to provide an automatic transformation from a core ontology part in F-Logic to OWL. Below, we define the transformations for the six possible forms of F-atoms[7]: instances of classes, subclasses, and the different method constructs, which concerns single valued method signature, multi valued method signature, single valued method application and multi valued method application.

We represent each transformation first in a table which we believe is more readable than the QVT syntax, which is very close to OCL as it uses many OCL constructs. The corresponding elements in the MOF metamodel for F-Logic and the corresponding elements in the MOF metamodel for OWL, defined through the mapping 'σ', are defined. In doing so, we sometimes rely on the OCL syntax to define elements. Elements that are indented, are

[7]Remember that the interpretation of the models is not always identical, and not all transformations yield semantically equivalent statements.

attributes of the element above. The variables s_1, s_2 and s_3 represent strings. After each table, we give the corresponding QVT syntax.

The F-Logic construct for defining an individual as an instance of a certain class, is written as follows: s_1 *:* s_2. The corresponding construct in OWL is the *ClassAssertion* construct: *ClassAssertion(*s_1 s_2*)*.
As an example, *aspirin* could be defined as an instance of *Medication*, represented in F-Logic as: *aspirin : Medication* and in OWL as: *ClassAssertion(aspirin Medication)*.
The table and QVT syntax below define this transformation.

MOF-element in Metamodel for F-Logic	**MOF-element in Metamodel for OWL = σ(element in Metamodel for F-Logic)**

FAtom
host = InstanceOf
instance = Term
name = s_1
class = Term
name = s_2
	ClassAssertion
	individual = Individual
	URI = s_1
	class = OWLClass
	URI = s_2

```
rule InstanceOf2ClassAssertion {
from
        f : FLogic!FAtom (
                f.host.oclIsTypeOf(FLogic!InstanceOf)
                )
        using {
                s_1 : String = f.host.instance.name;
                s_2 : String = f.host.class.name;
        }
to
        c : OWL!ClassAssertion (
                individual <- i,
                class <- o
        ),
        i : OWL!Individual(
                URI <- s_1
        ),
        o : OWL!OWLClass(
                URI <- s_2
        )
}
```

The second type of F-atom in F-Logic is the subclass-construct, defining that one concept is a subconcept of another one: $s_1 :: s_2$. The corresponding construct in OWL is the *SubClassOf* construct: *SubClassOf(*s_1 s_2*)*.
An example could define a class *FluidMedication* as a subclass of the class *Medication*, presented in F-Logic as follows: *FluidMedication :: Medication* and in OWL as follows: *SubClassOf(FluidMedication Medication)*.
The corresponding transformation is presented in the following table and QVT syntax.

MOF-element in Metamodel for F-Logic	**MOF-element in Metamodel for OWL = σ(element in Metamodel for F-Logic)**

FAtom
host = SubClassOf
subclass = Term
name = s_1
superclass = Term
name = s_2
	SubClassOf
	subClass = OWLClass
	URI = s_1
	superClass = OWLClass
	URI = s_2

```
rule FLogicSubClassOf2OWLSubClassOf {
from
        f : FLogic!FAtom (
                f.host.oclIsTypeOf(FLogic!SubClassOf)
        )
        using {
                s_1 : String = f.host.subclass.name;
                s_2 : String = f.host.superclass.name;
        }
to
        s_3 : OWL!SubClassOf (
                subClass <- s_4,
                superClass <- s_5
        ),
        s_4 : OWL!OWLClass(
                URI <- s_1
        ),
        s_5 : OWL!OWLClass(
                URI <- s_2
        )
}
```

F-Logic provides four other types of F-atoms, which define methods over classes. OWL distinguishes between data properties and object properties. Our transformation presents all properties as object properties. Further processing could detect the specific type of property from an F-Logic method construct.

The methods in F-Logic can be single- or multi-valued, and can be either method signatures or method applications. We first address the method signatures.

The general form of a single-valued method signature in F-Logic is as follows: $s_1[s_2 \Rightarrow s_3]$. By transforming this F-Logic construct into OWL, two constructs are built. Firstly, one construct defines an OWL description using an *ObjectAllValuesFrom* restriction to denote the fact that the application of the specified method on the OWL description has instances of the specified class as value: *SubClassOf(*s_1 *ObjectAllValuesFrom(*s_2 s_3*))*. Secondly, as we additionally have to denote that the F-Logic construct denotes a single-valued method, an *ObjectMaxCardinality* restriction with maximum cardinality 1 is defined over the same description[8] [9]: *SubClassOf(*s_1 *ObjectMaxCardinality(1* s_2 s_3*))*.
An example single-valued method signature in F-Logic could define that the method *producedBy* applied on instances of the class *Medication* returns instances of the class *Laboratory*: *Medication[producedBy* $\Rightarrow$ *Laboratory]* . The corresponding OWL constructs for this example are: *SubClassOf(Medication ObjectAllValuesFrom(producedBy Laboratory))* and *SubClassOf(Medication ObjectMaxCardinality(1 producedBy Laboratory))*.
The following table and QVT syntax present the transformation.

[8]Note that a single-valued method in F-Logic possibly has no value, wich means that it does not denote an exact cardinality of 1 but a maximum cardinality of 1.

[9]Note that we can not define the property as a functional property instead of defining the cardinality restriction, as that would be defined globally on the property. A cardinality restriction is defined locally on the property for the concerning class, as a method in F-Logic does as well.

MOF-element in Metamodel for F-Logic	**MOF-element in Metamodel for OWL = σ(element in Metamodel for F-Logic)**

FAtom
  host = Term
    name = s_1
  methods =
  SingleValuedSignature
    name = Term
      name = s_2
    value = Term
      name = s_3
	SubClassOf
	  subClass = OWLClass
	    URI = s_1
	  superClass = ObjectAllValuesFrom
	    property = ObjectProperty
	      URI = s_2
	    class = OWLClass
	      URI = s_3
	SubClassOf
	  subClass = OWLClass
	    URI = s_1
	  superClass = ObjectMaxCardinality
	    property = ObjectProperty
	      URI = s_2
	    class = OWLClass
	      URI =s_3
	    cardinality = 1

```
rule SingleValuedSignature2Property {
from
        f: FLogic!Fatom (
                f.host.oclIsTypeOf(FLogic!Constant) and
                f.methods.
                oclIsTypeOf(FLogic!SingleValuedSignature) and
                f.methods.name.oclIsTypeOf(FLogic!Constant) and
                f.methods.value.oclIsTypeOf(FLogic!Constant)
        )
        using {
                s_1 : String = f.host.name;
                s_2 : String = f.methods.name.name;
                s_3 : String = f.methods.value.name;
        }
to
        s_4 : OWL!SubClassOf(
                subClass <- s_5,
                superClass <- s_6
        ),
        s_5 : OWL!OWLClass(
                URI <- s_1
        ),
        s_6 : OWL!ObjectAllValuesFrom(
                property <- o_1,
                class <- d_1
        ),
        o_1 : OWL!ObjectProperty(
                URI <- s_2
        ),
        d_1 : OWL!OWLClass(
                URI <- s_3
        ),
        s_7 : OWL!SubClassOf(
                subClass <- s_8,
                superClass <- s_9
        ),
        s_8 : OWL!OWLClass(
                URI <- s_1
        ),
        s_9 : OWL!ObjectMaxCardinality (
                property <- o_2,
                class <- d_2,
                cardinality <- 1
        ),
        o_2 : OWL!ObjectProperty(
                URI <- s_2
        ),
        d_2 : OWL!OWLClass(
                URI <- s_3
        )
}
```

The transformation of an F-Logic multi-valued method signature is quite similar to the single-valued method signature but results in only one OWL construct, as it does not pose any restrictions on the cardinality[10]. Generally, a multi-valued method signature in F-Logic is presented as follows: *s_1[s_2 $\Rightarrow\!\!\!\Rightarrow$ s_3]* . The corresponding OWL construct for this is: *SubClassOf(s_1 ObjectAllValuesFrom (s_2 s_3)).*
The following example defines that when the method *treats* is applied on the class *Medication*, a set of values from the class *Disease* is returned: *Medication[treats $\Rightarrow\!\!\!\Rightarrow$ Disease].* In

[10]Note that F-Logic defines multi-valued methods as methods that return sets of values, although no minimum number of elements for such sets is specified.

OWL, this statement would look like: *SubClassOf(Medication ObjectAllValuesFrom (treats Disease)).*
The table and QVT syntax below define the corresponding transformation from the F-Logic metamodel to the OWL metamodel.

MOF-element in Metamodel for F-Logic	**MOF-element in Metamodel for OWL = σ(element in Metamodel for F-Logic)**

FAtom
host = Term
name = s_1
methods = MultiValuedSignature
name = Term
name = s_2
value = Term
name = s_3
	SubClassOf
	subClass = OWLClass
	URI = s_1
	superClass = ObjectAllValuesFrom
	property = ObjectProperty
	URI = s_2
	class = OWLClass
	URI = s_3

```
rule MultiValuedSignature2Property {
from
        f : FLogic!FAtom (
                f.host.oclIsTypeOf(FLogic!Constant)        and
                f.methods.oclIsTypeOf(FLogic!MultiValuedSignature)
                and
                f.methods.name.oclIsTypeOf(FLogic!Constant) and
                f.methods.value.oclIsTypeOf(FLogic!Constant)
        )
        using{
                s_1 : String = f.host.name;
                s_2 : String = f.methods.name.name;
                s_3 : String = f.methods.value.name;
        }
to
        s_4 : OWL!SubClassOf(
                subClass <- s_5,
                superClass <- s_6
        ),
        s_5 : OWL!OWLClass(
                URI <- s_1
        ),
        s_6 : OWL!ObjectAllValuesFrom(
                property <- o_1 ,
                class <- d_1
        ),
        o_1  : OWL!ObjectProperty(
                URI <- s_2
        ),
        d_1  : OWL!OWLClass(
                URI <- s_3
        )
}
```

Finally, we present the transformations of the two remaining F-Logic constructs that make up the core part of the ontology consisting of the taxonomy and the instantiations: the method applications, which instantiate method signatures on class instances. Again, F-Logic distinguishes between methods returning scalar values and methods returning sets of values. The general form of a single-valued method application in F-Logic is as follows: $s_1[s_2 \rightarrow s_3]$. The corresponding construct in OWL is an *ObjectPropertyAssertion*: *ObjectPropertyAssertion(*s_2 s_1 s_3*)*.
An example statement of a single-valued method application could specify that the application of the method *producedBy* on the instance *aspirin* returns the instance *bayer*, represented in F-Logic as: *aspirin[producedBy → bayer]* and in OWL as *ObjectPropertyAssertion(producedBy aspirin bayer)*.
We define the transformation of the corresponding MOF elements in the F-Logic metamodel to the corresponding elements in the OWL metamodel in the following table and QVT syntax.

MOF-element in Metamodel for F-Logic	**MOF-element in Metamodel for OWL = σ(element in Metamodel for F-Logic)**

FAtom
  host = Term
    name = s_1
  methods =
  SingleValuedApplication
    name = Term
      name = s_2
    value = Constant
      name = s_3
	ObjectPropertyAssertion
	  property = ObjectProperty
	    URI = s_2
	  source = Individual
	    URI = s_1
	  target = Individual
	    URI = s_3

```
rule SingleValuedApplication2Property{
from
        f : FLogic!FAtom(
                f.host.oclIsTypeOf(FLogic!Constant) and
                f.methods.
                oclIsTypeOf(FLogic!SingleValuedApplication) and
                f.methods.name.oclIsTypeOf(FLogic!Constant) and
                f.methods.value.oclIsTypeOf(FLogic!Constant)
        )
        using{
                s_1 : String = f.host.name;
                s_2 : String = f.methods.name.name;
                s_3 : String = f.methods.value.name;
        }
to
        o_1 : OWL!ObjectPropertyAssertion(
                property <- o_2,
                source <- i_1,
                target <- i_2
        ),
        o_2 : OWL!ObjectProperty(
                URI <- s_2
        ),
        i_1 : OWL!Individual(
                URI <- s_1
        ),
        i_2 : OWL!Individual(
                URI <- s_3
        )
}
```

At last, we define the transformation for multi-valued method applications in F-Logic: $s_1[s_2 \twoheadrightarrow s_3]$. The corresponding construct in OWL is the same as with single-valued method applications as OWL does not make the same distinction for property instantiations as F-Logic does for method applications: *ObjectPropertyAssertion(s_2 s_1 s_3)*.
An example could define that the application of the method *treats* on the instance *aspirin* returns the instance *headache* as value. In F-Logic, this is presented as: *aspirin[treats ↠ headache]*, whereas the corresponding OWL construct is: *ObjectPropertyAssertion(treats aspirin headache)*.
The table and QVT syntax below specify the transformation of the corresponding elements in the MOF metamodel for F-Logic to the corresponding elements in the MOF metamodel for OWL.

MOF-element in Metamodel for F-Logic	**MOF-element in Metamodel for OWL = σ(element in Metamodel for F-Logic)**

FAtom
host = Term
name = s_1
methods =
MultiValuedApplication
name = Term
name = s_2
value = Term
name = s_3
	ObjectPropertyAssertion
	property = ObjectProperty
	URI = s_2
	source = Individual
	URI = s_1
	target = Individual
	URI = s_3

```
rule MultiValuedApplication2Property{
from
        f : FLogic!FAtom(
                f.host.oclIsTypeOf(FLogic!Constant) and
                f.methods.
                oclIsTypeOf(FLogic!MultiValuedApplication) and
                f.methods.name.oclIsTypeOf(FLogic!Constant) and
                f.methods.value.oclIsTypeOf(FLogic!Constant)
        )
        using{
                s_1 : String = f.host.name;
                s_2 : String = f.methods.name.name;
                s_3 : String = f.methods.value.name;
        }
to
        o_1 : OWL!ObjectPropertyAssertion(
                property <- o_2,
                source <- i_1,
                target <- i_2
        ),
        o_2 : OWL!ObjectProperty(
                URI <- s_2
        ),
        i_1 : OWL!Individual(
                URI <- s_1
        ),
        i_2 : OWL!Individual(
                URI <- s_3
```

```
            )
}
```

6.3. Conclusion

In this chapter and the accompanying appendices, we presented a MOF-based metamodel for ontologies and rules in F-Logic, along with OCL constraints that give it the right precision. After we also presented a MOF-based metamodel for ontologies in OWL in Chapter 4 starting on page 37, we could define a transformation from the metamodel of F-Logic to the metamodel of OWL. This transformation on metamodel level allows us to automatically transform a taxonomy and its instantiations in F-Logic to OWL. Hence, we provide a first step of support for companies that are already using F-Logic but want to benefit from the advantages of OWL.

7. Metamodel Descriptions for OWL Ontology Mappings

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 7.1 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] For a full account on the metamodel, we refer the reader to Appendix A.7 starting on page 224, which lists each metaclass with an informal description, its attributes, associations and generalizations, and applicable OCL constraints.

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 7.2 starting on page 96 presents the extension for C-OWL mappings, consisting of a set of constraints. Similarly, Section 7.3 starting on page 97 presents the extension for DL-Safe Mappings.

Additionally, Appendix A.8 starting on page 228 provides the mapping between the metamodel and the language C-OWL, whereas Appendix A.9 starting on page 230 provides the mapping for DL-Safe Mappings.

Finally, Section 7.4 starting on page 99 concludes the chapter with a little summary.

[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.

7.1. 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 7.1.1 presents mappings, after which Section 7.1.2 presents queries.

7.1.1. 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 made the following choices:

- A mapping is a set of mapping assertions that consist of a semantic relation between mappable elements in different ontologies. Figure 7.1 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 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.

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 [SSW05]. 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

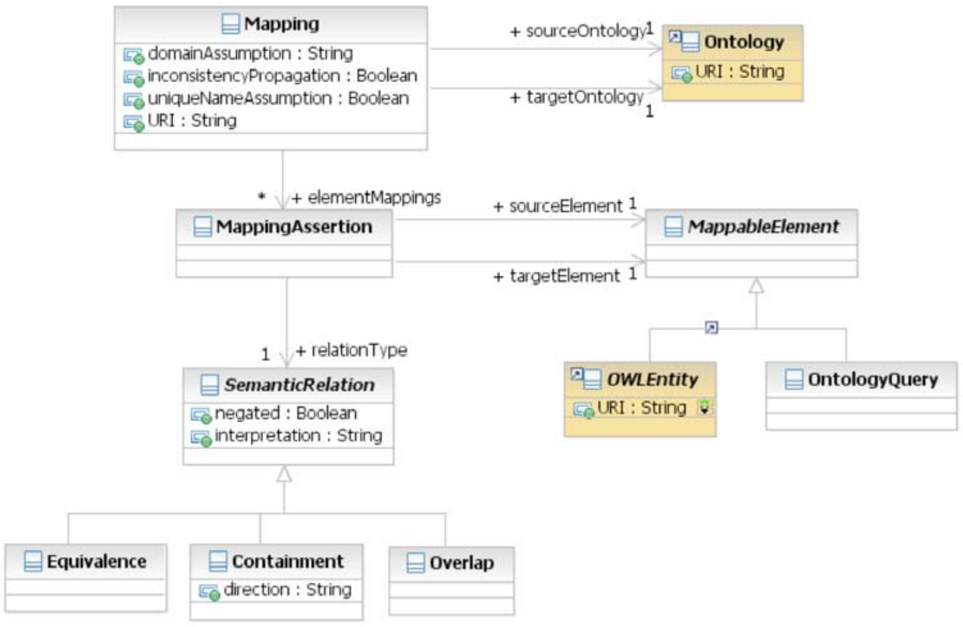


Figure 7.1.: OWL mapping metamodel: mappings

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.

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.

7.1.2. Queries

A mapping assertion can take a query as mappable element. Figure 7.2 demonstrates the class *Query* that reuses constructs from the SWRL metamodel.

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 zero, *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 defines 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)))
   ```

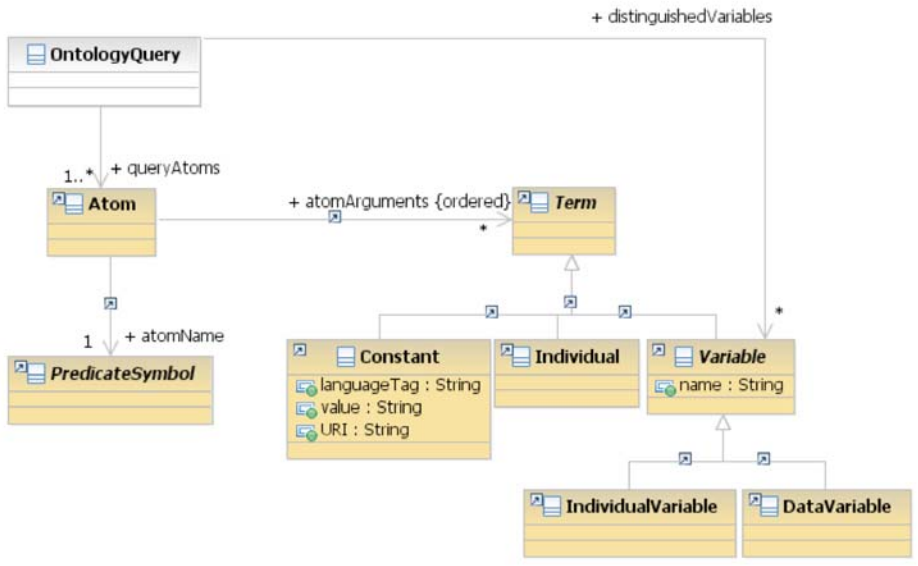


Figure 7.2.: OWL mapping metamodel: queries

7.2. A Metamodel Extension 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]where 'inv' stands for the constraint type *invariant*.

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

7.3. A Metamodel Extension 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)

7.4. Conclusion

We introduced an extension for the MOF-based metamodel for OWL and SWRL to support OWL ontology mappings. A common extension for general mapping aspects covers several existing OWL mapping languages. Constraints on this define concrete extensions for the specific languages C-OWL and DL-Safe Mappings. A constraint checker could check which metamodel is actually instantiated by a certain model of mappings, so to which concrete formalism the model conforms. Such a model could be modeled using the UML profile that we define in Chapter 8 starting on page 101, which allows graphical modeling of ontologies, rules and ontology mappings in a UML-like syntax based on the metamodel.

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 UML and ontology languages. 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 115 gives an overview of the profile. The profile extensions for rules and ontology mappings are represented similarly in Section 8.2 starting on page 117 and Section 8.3 starting on page 121. We complete the specifications of the profile with the mappings between the metamodel for OWL, SWRL and ontology mappings on the one hand and the profile on the other hand, given in Appendix A.10 starting on page 232.

8.1. A UML Profile for 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.

8.1.1. UML Syntax for Ontologies

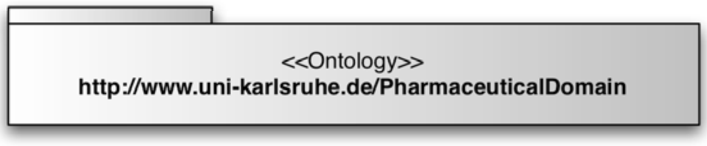


Figure 8.1.: Ontology(http://www.uni-karlsruhe.de/PharmaceuticalDomain)

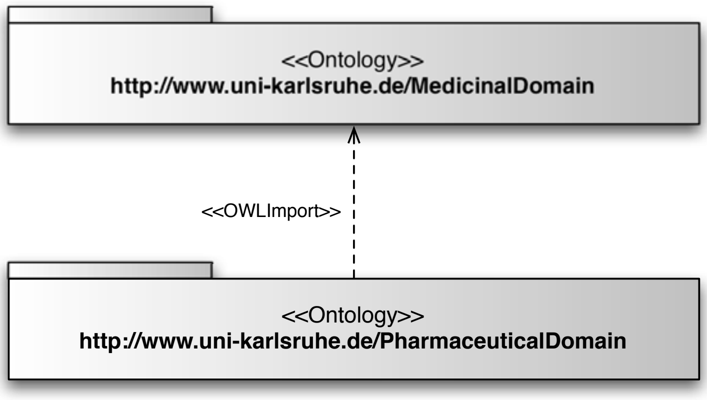

<<Ontology>>
http://www.uni-karlsruhe.de/MedicinalDomain
<<OWLImport>>
<<Ontology>>
http://www.uni-karlsruhe.de/PharmaceuticalDomain

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

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 *MedicinalDomain*. 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.

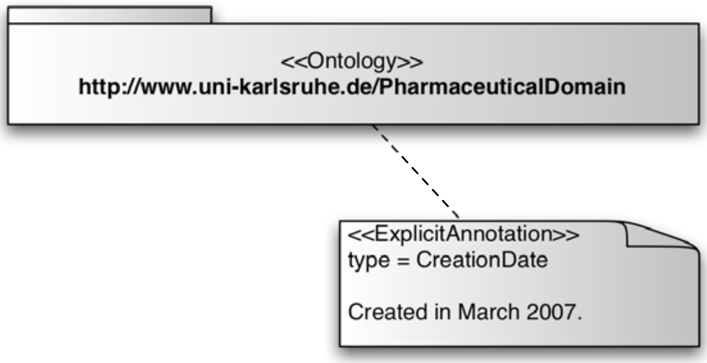


Figure 8.3.: Ontology(PharmaceuticalDomain Annotation(CreationDate "Created in March 2007."))

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

Medication

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.

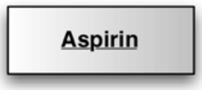
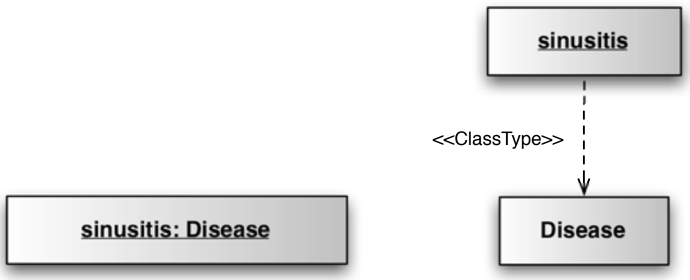


Figure 8.5.: Individual(Aspirin)

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.

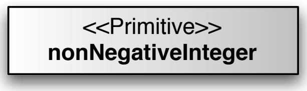
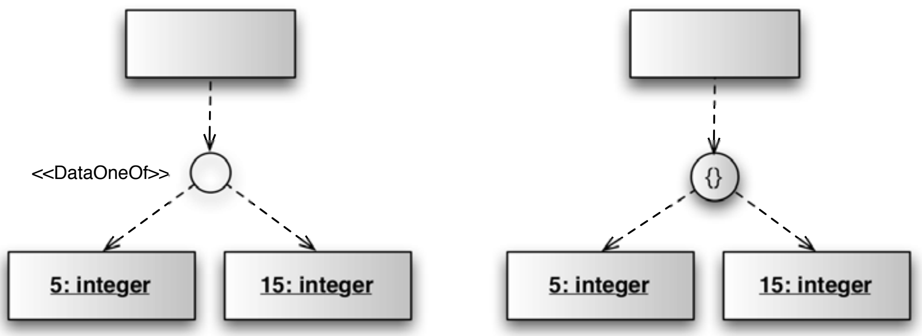


Figure 8.7.: Datatype(xsd:nonNegativeInteger)

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 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.

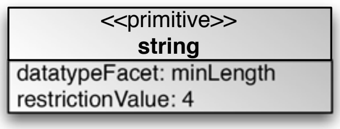
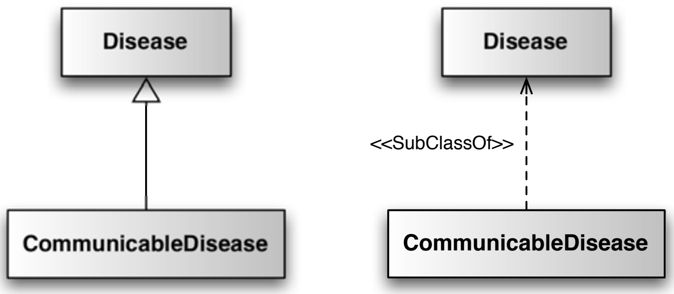


Figure 8.9.: DatatypeRestriction(xsd:string minLength 4)

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 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.



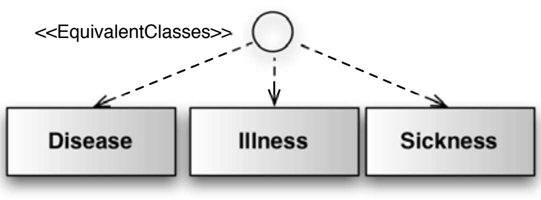
(a) Generalization (b) Dependency

Figure 8.10.: SubClassOf(CommunicableDisease Disease)

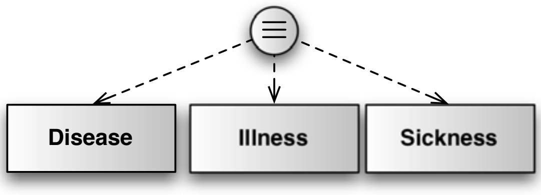
OWL 1.1 allows more than two classes in the construct for equivalent classes. Although the specific notation for two classes is most useful and intuitive for this particular case, an elaborated notation for the construct with more than two classes is necessary. For this we need a certain construct connected 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.

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



(a) White dot



(b) Icon

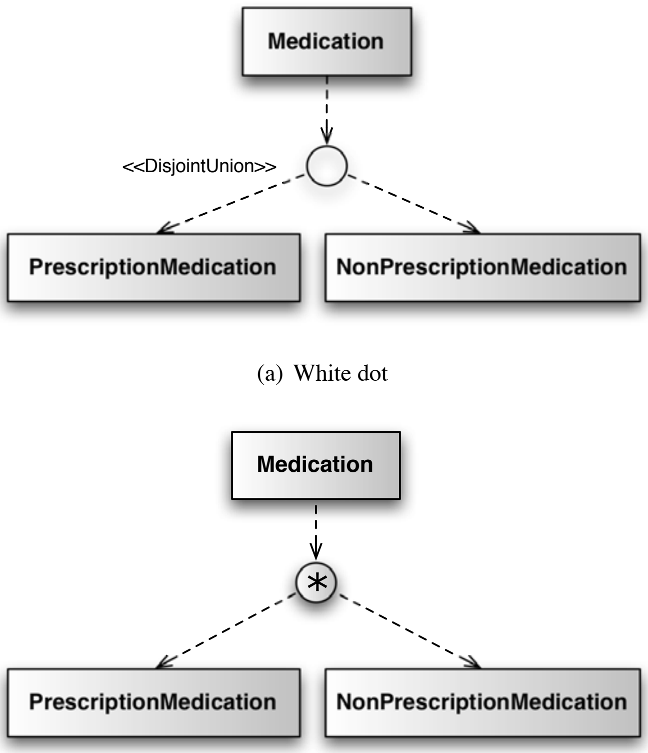
Figure 8.11.: EquivalentClasses(Disease Illness Sickness)

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 used on their own, but are used in axioms. To give the reader a clear idea of how the constructs would be used, we present 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 be-



(a) White dot

(b) Icon

Figure 8.12.: DisjointUnion(Medication PrescriptionMedication NonPrescriptionMedication)

fore 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 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 cardi-

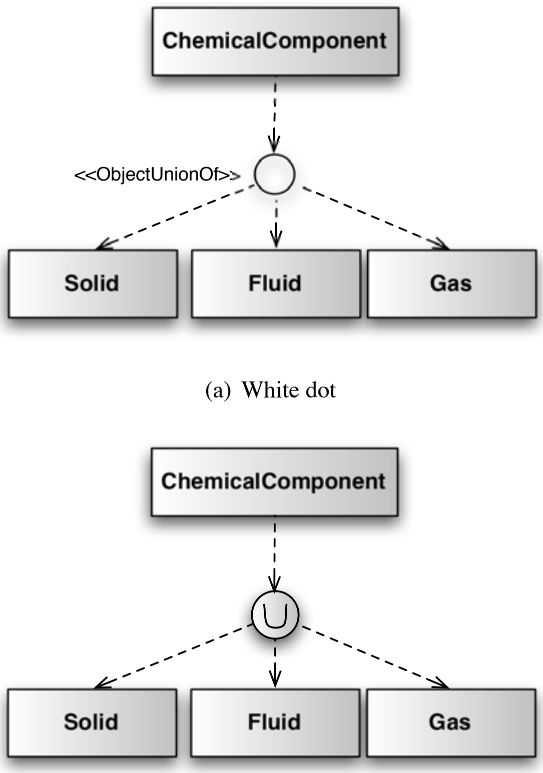


Figure 8.13.: SubClassOf(ChemicalComponent ObjectUnionOf(Solid Fluid Gas))

nality. Figure 8.14 shows how UML multiplicities are applied for an OWL minimum cardinality on an object property. The notation for maximum and exact cardinalities, as well as for these cardinalities on data properties, are similar.

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

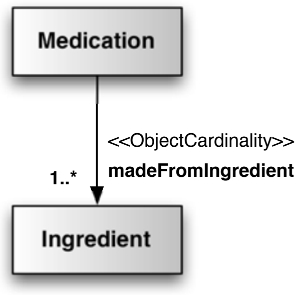


Figure 8.14.: SubClassOf(Medication ObjectMinCardinality(1 madeFromIngredient Ingredient))

same as for object properties, and utilizes the stereotypes <<*DataSomeValuesFrom*>> and <<*DataAllValuesFrom*>>.

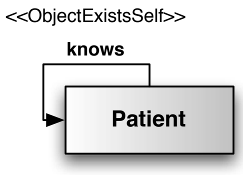


Figure 8.15.: SubClassOf(Patient ObjectExistsSelf(knows))

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.

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

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

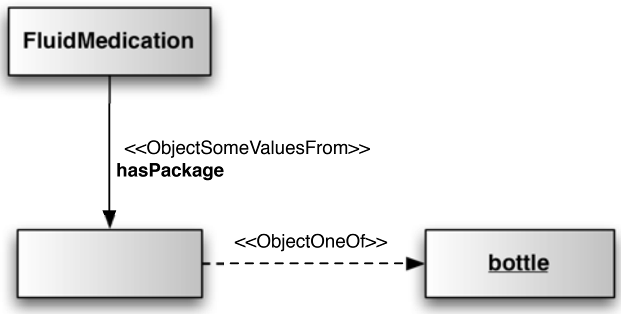


Figure 8.16.: SubClassOf(FluidMedication ObjectHasValue(hasPackage bottle))

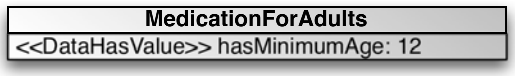


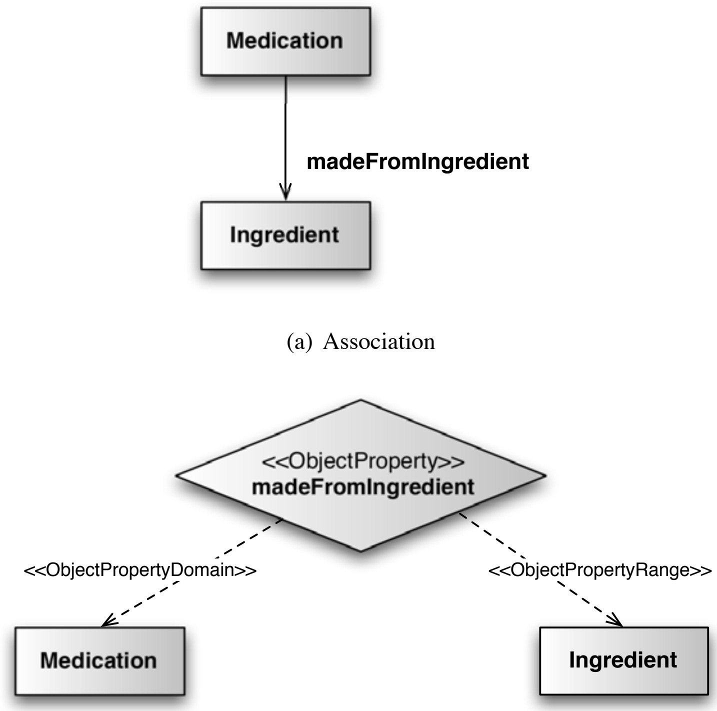
Figure 8.17.: SubClassOf(MedicationForAdults DataHasValue(hasMinimumAge ”12”^^xsd:integer))

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



(a) Association

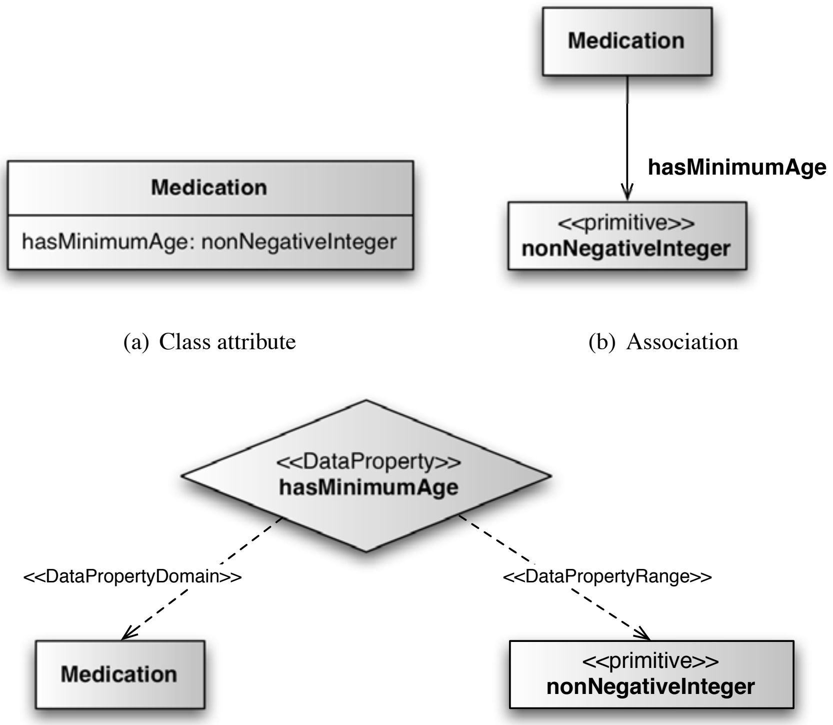
(b) Diamond class

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

properties. To be able to distinguish them from normal 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.



(a) Class attribute (b) Association

(c) Diamond class

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

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,

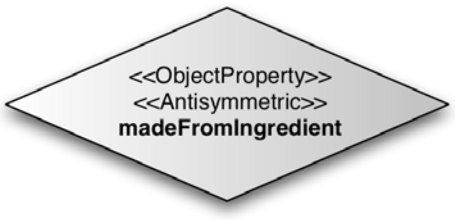


Figure 8.20.: AntisymmetricObjectProperty(madeFromIngredient)

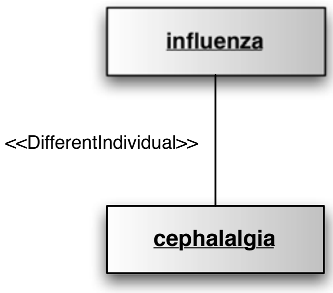


Figure 8.21.: DifferentIndividual(influenza cephalalgia)

<<*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*>>.

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.

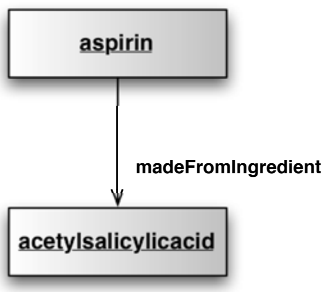


Figure 8.22.: ObjectPropertyAssertion(madeFromIngredient aspirin acetylsalicylicacid)

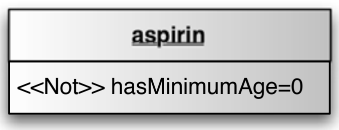


Figure 8.23.: NegativeDataPropertyAssertion(hasMinimumAge aspirin 0)

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.

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
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

Stereotype	Tags	UML metamodel elements
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
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 starting on page 59.

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 *name* 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.

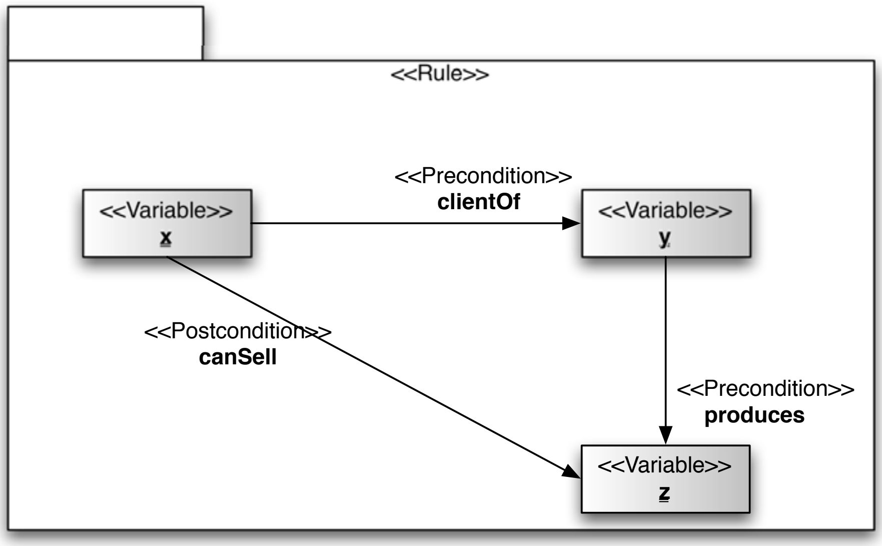


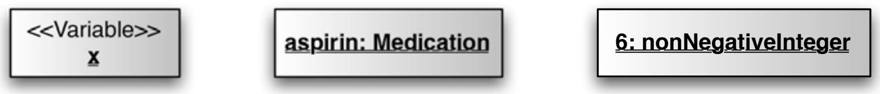
Figure 8.24.: canSell$(x, z) \leftarrow$ clientOf$(x, y) \wedge$ produces(y, z)

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

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

each type of term: a variable *x*, an individual *aspirin* belonging to the class *Medication*, and a data value *6* which is a *nonNegativeInteger*.



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

Figure 8.25.: Rule terms

8.2.3. UML Syntax for Predicate Symbols in Atoms

8.2.3.1. 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 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.

8.2.3.2. 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*.

8.2.3.3. 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.

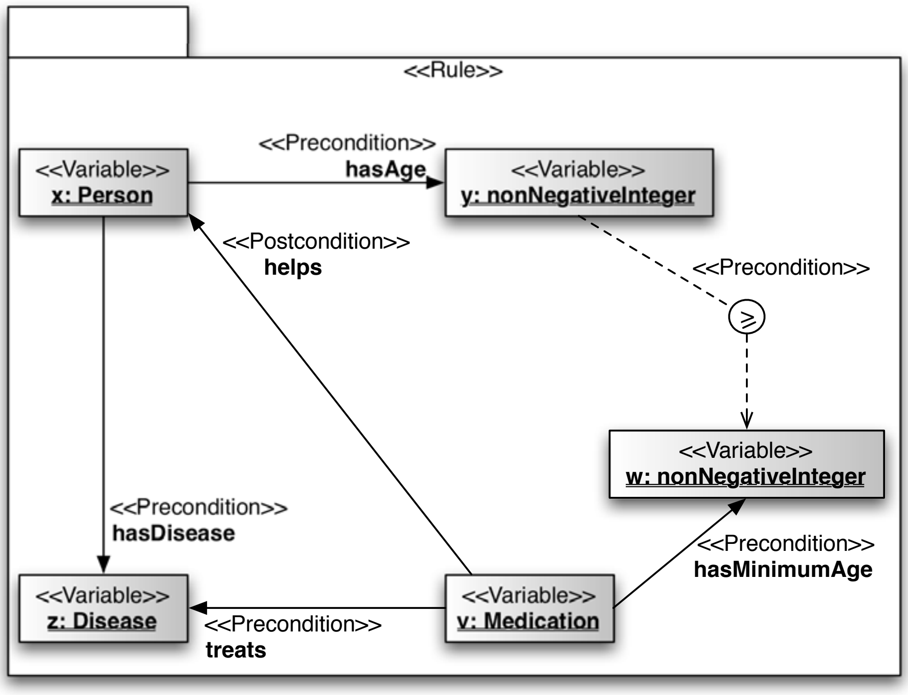


Figure 8.26.: helps(v, x) ← Person(x), nonNegativeInteger(y), hasAge(x, y), Disease(z), hasDisease(x, z), Medication(v), treats(v, z), nonNegativeInteger(w), hasMinimumage(v, w), swrlb:greaterThanOrEqual(y, w)

8.2.3.4. 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*.

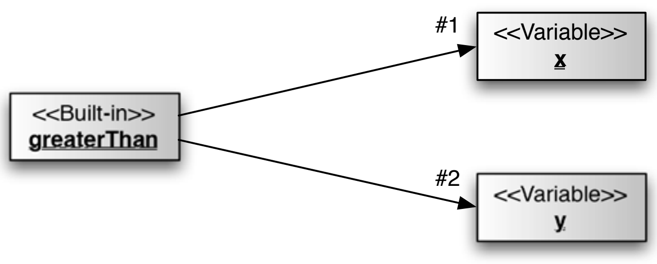


Figure 8.27.: Built-in predicates

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 115.

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 125 shows an overview of the profile extension.

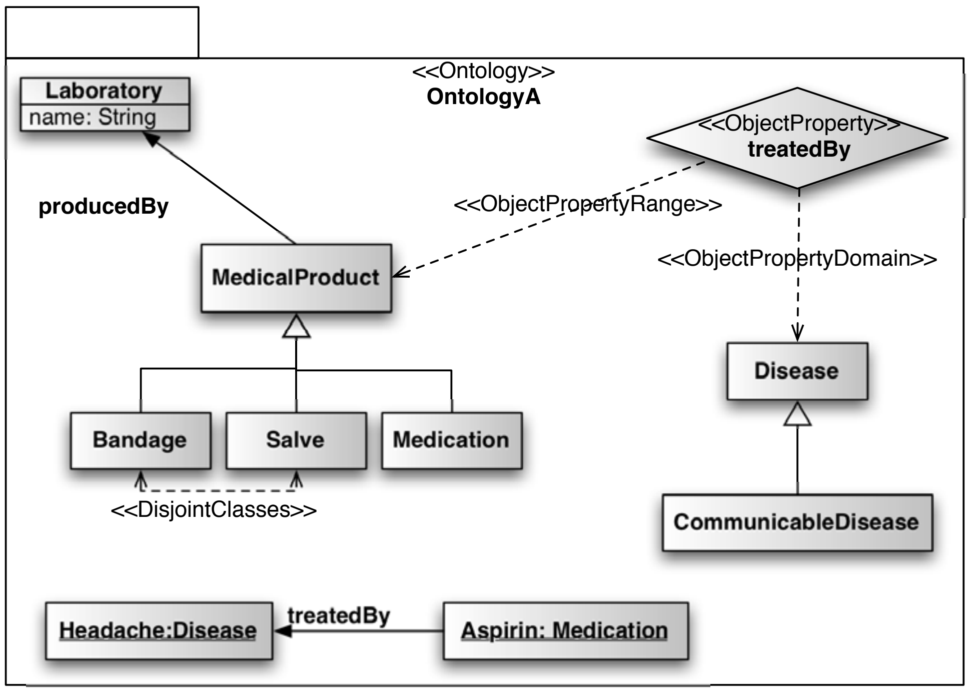


Figure 8.28.: A first sample ontology depicted using the UML profile

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 meta-model, 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.

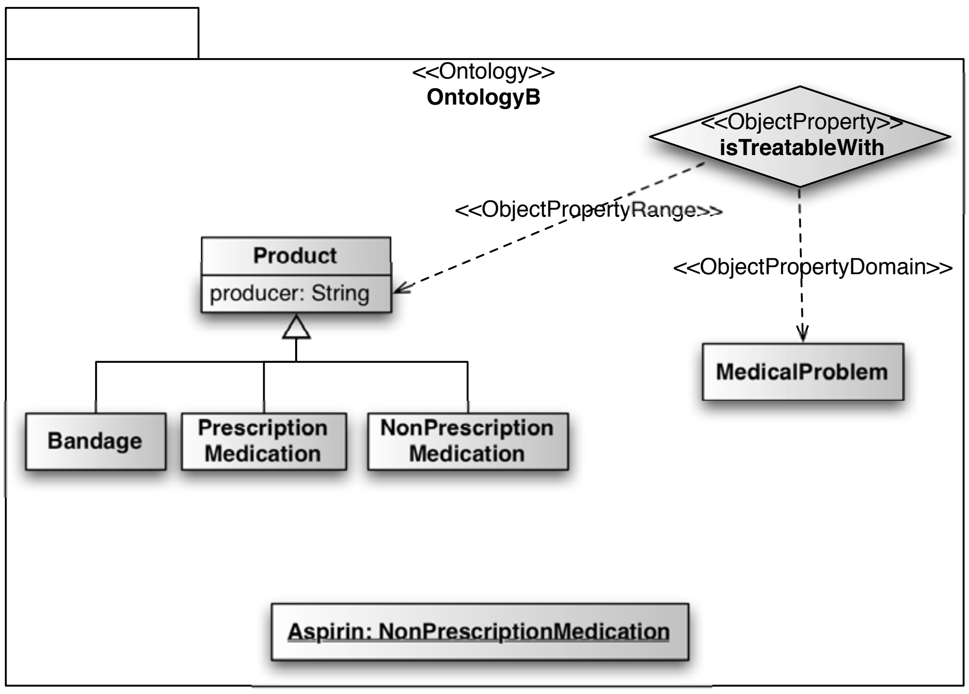


Figure 8.29.: A second sample ontology depicted using the UML profile

Dependencies with the stereotype <<Mapping>> link the mapping definition to the defined mapping assertions. 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 *Medical-Product* 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 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 these 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,

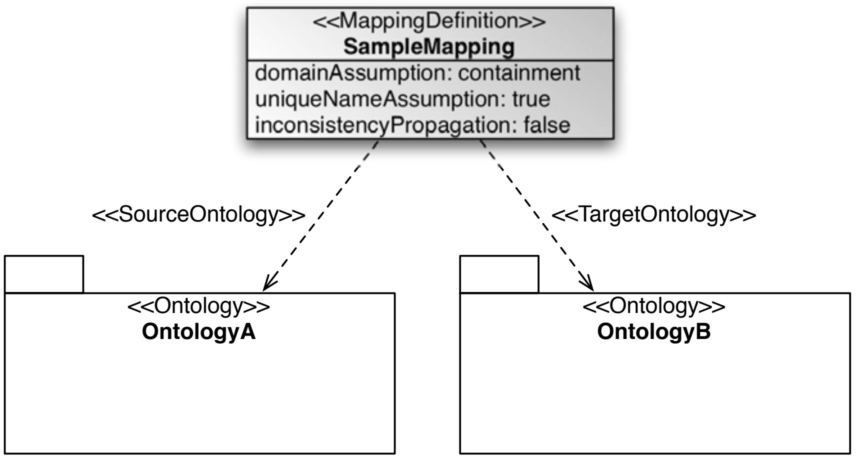


Figure 8.30.: Sample mapping definition between two ontologies

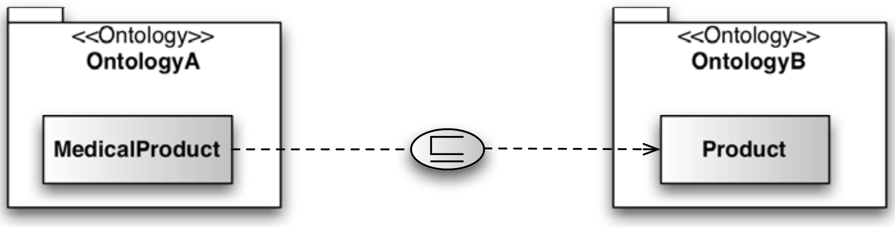


Figure 8.31.: Sample sound containment relation between two concepts

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

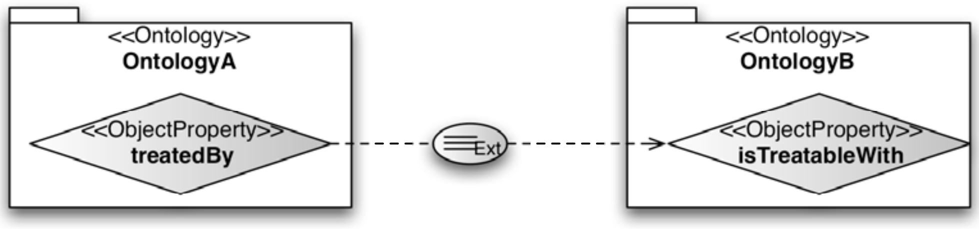


Figure 8.32.: Sample extensional equivalence relation between two properties

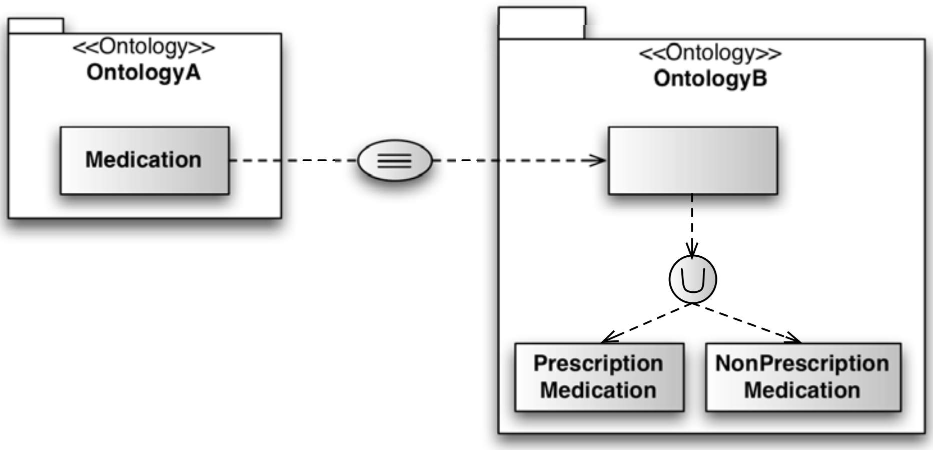


Figure 8.33.: Sample equivalence relation between complex class descriptions

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.

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 115 and 8.2.4 on page 121.

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

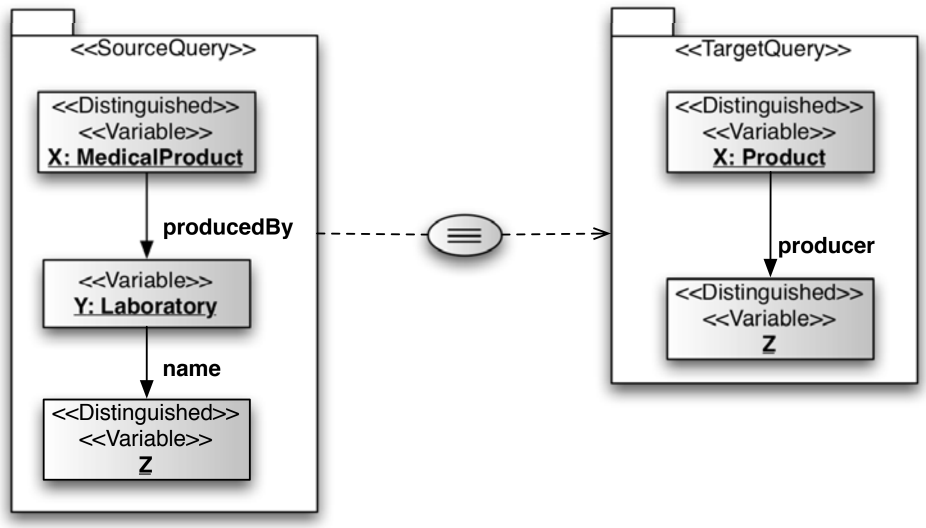

<<SourceQuery>>
<<Distinguished>> <<Variable>> X: MedicalProduct
producedBy
<<Variable>> Y: Laboratory
name
<<Distinguished>> <<Variable>> Z
<<TargetQuery>>
<<Distinguished>> <<Variable>> X: Product
producer
<<Distinguished>> <<Variable>> Z

Figure 8.34.: Sample equivalence relation between two queries

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

8.4. Conclusion

In this chapter we introduced a visual syntax for ontologies, rules and ontology mappings, based on UML. We can rely on the existing experience of users with the UML-paradigm as well as rely on available tool support. Mappings between the metamodel and the profile, as presented in Appendix A.10 starting on page 232, define the relationship between the two. Based on this and on the mapping function between the metamodel and the logical languages (presented in Appendices A.2, A.4, A.8 and A.9), we can provide automatic transformations between visual models and OWL ontologies, SWRL rules and ontology mappings in C-OWL or DL-Safe Mappings, and vice versa.

Part III.

Finale

9. Implementation and Case Study Evaluation

To evaluate our approach for visually modeling ontologies, we implemented a prototype with which we conducted two experiments. This chapter explains the architecture of the implementation in Section 9.1, and discusses the general setup of the evaluation in Section 9.2. Consequently, Sections 9.3 and 9.4 represent details and results of both experiments. Finally, Section 9.5 concludes the chapter.

9.1. Prototype Implementation

We implemented an editor based on a framework called Eclipse[1], which is extensible via plug-ins. We provide a plug-in called OntoModel, based on our metamodel and corresponding UML profile to support the graphical development of OWL ontologies and ontology mappings.

OntoModel provides the following functionalities:

- Load (import) an ontology in OWL.
- Graphically model ontologies and mappings between ontologies using a UML-syntax.
- Save (export) a model in the appropriate syntax (OWL or DL-Safe Mappings).

Figure 9.1 shows how OntoModel builds on Eclipse and some of its available plug-ins: it builds on the Graphical Modeling Framework (GMF[2]), which in turn builds on the Graphical Editing Framework (GEF[3]) and the Eclipse Modeling Framework (EMF[4]).

EMF is a code generation facility for building applications based on a structured model. It helps to turn models into efficient, correct, and easily customizable Java code. Out of our Ecore metamodel, we created a corresponding set of Java classes using the EMF generator. The generated classes can be edited and the code is unaffected by model changes and regeneration. Only when the edited code depends on something that changed in the model, that code has to be adapted to reflect these changes.

[1] http://www.eclipse.org/.
[2] http://www.eclipse.org/gmf/.
[3] http://www.eclipse.org/gef/.
[4] http://www.eclipse.org/modeling/emf/.

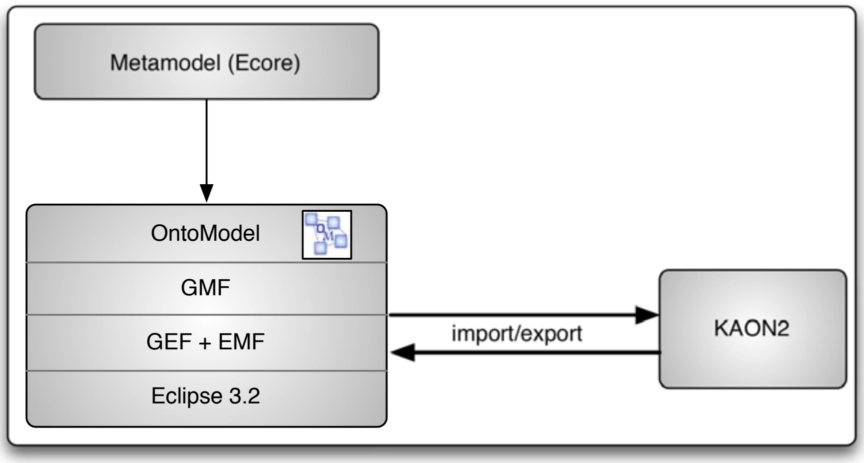


Figure 9.1.: Architecture of the prototype implementation

EMF consists of two fundamental frameworks: the core framework and EMF.Edit. The core framework provides basic generation and runtime support to create Java classes for a model, whereas EMF.Edit extends and builds on the core framework, adding support for generating a basic working model editor as well as adapter classes that enable viewing and editing of a model. EMF started out as an implementation of the MOF specification. It can be thought of as a highly efficient Java implementation of MOF, and its MOF-like metamodel is called Ecore.

The EMF adapter listens for model changes. When an object changes its state, GEF becomes aware of the change, and performs the appropriate action, such as redrawing a figure due to a move request. GEF provides the basic graphical functionality for GMF.

GMF is the layer connecting OntoModel with GEF and EMF. It defines and implements many functionalities of GEF to be used directly in an application and complements the standard EMF generated editor. Figure 9.2 illustrates the main components and models used during the GMF-based development of OntoModel.

The graphical definition model is the core concept of GMF. It contains the information related to the graphical elements that are used in our ontology models. However, they do not have any direct connection to our metamodel. The graphical definition model is generated semi-automatically from our Ecore metamodel.[5] Similarly, the tooling definition model is generated semi-automatically from the metamodel to design the palette and other periphery (menus, toolbars, etc).

[5]Note that, although our editor is UML-based, GEF also provides other graphical elements.

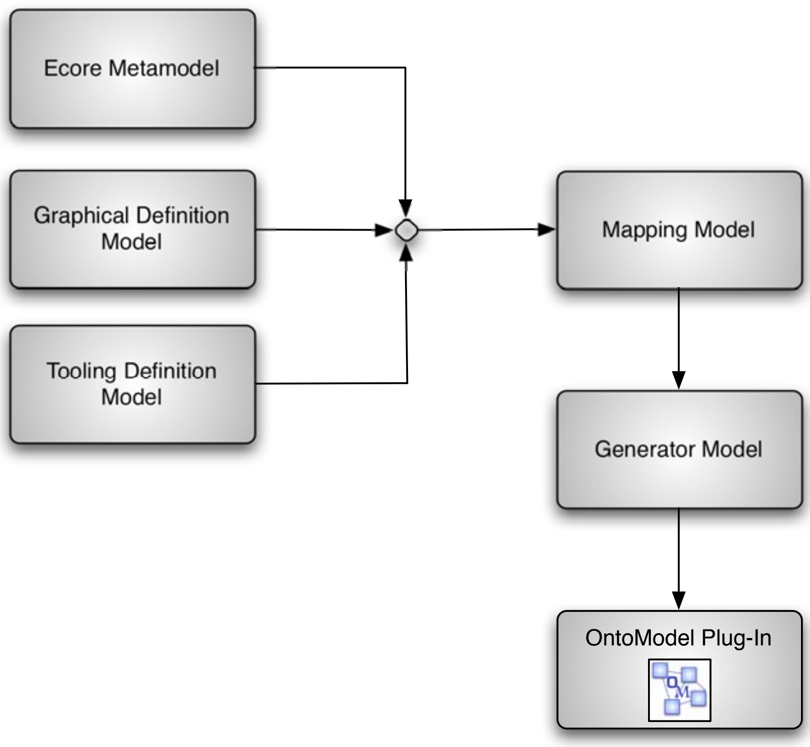


Figure 9.2.: The main components and models used during the GMF-based development of OntoModel

A goal of GMF is to allow the graphical definition to be reused for several domains. This is achieved by using a separate mapping model to link the graphical and tooling definitions to the selected domain metamodel. This semi-automatically generated model is a key model to GMF development and is used as input to a transformation step producing the generator model.

The generator model generates the OntoModel plug-in which is then refined with dialogs and other application-specific user interface aspects. The plug-in bridges the graphical notation and the domain metamodel when a user is working with a diagram, and also provides for the persistence and synchronization of both. An important aspect of GMF-based application development is that a service-based approach to EMF and GEF functionality is provided which can be leveraged by non-generated applications.

Finally, Figure 9.1 illustrates that for import and export of OWL ontologies and DL-Safe

Mappings, we rely on KAON2[6], an infrastructure providing among others an API for programmatic management of ontologies in these languages.

Figure 9.3 shows a screenshot of the prototype. The navigator pane on the left lists the various models in use together with their diagrams, whereas the palette on the right provides a menu for the different ontology elements that can be modeled. The main window displays a simple ontology.

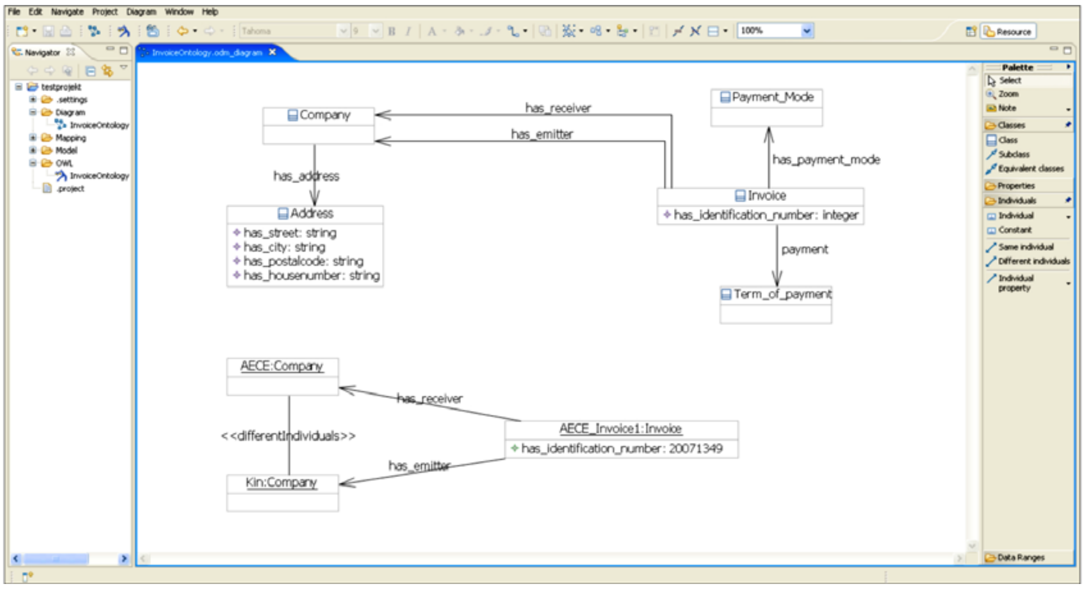


Figure 9.3.: A screenshot of the implemented prototype

9.2. Case Study Setting

The pharmacy sector is an important economic market in Europe. All European countries have a similar scenario with the same actors and similar legislation. The main actors in this sector are governmental entities, pharmacists and laboratories. Pharmainnova is a cooperation of several pharmaceutical laboratories in Spain with the objective of improving the commercial relations between laboratories and their partners, by introducing electronic interchanges. One axis of technology improvement of Pharmainnova, addressed as a case study in the European project NeOn[7], aims at supporting the process of electronic invoicing between pharmaceutical

[6]http://kaon2.semanticweb.org/.

[7]NeOn is a project involving fourteen European partners and co-funded by the European Commission's Sixth Framework Programme, started in March 2006 for a duration of four years. For further information see http://www.neon-project.org/.

labatories, their customers and their suppliers. The main steps to introduce the electronic exchange of invoice, are:

- Sending digital invoices over internet using electronic signature to guarantee security;
- Reception, storage and search for the electronic invoices;
- Connecting the electronic invoice platform to the companies' external systems to integrate the data flow;
- The management of the complete invoice cycle, including issues as confirmations, disputes or corrections of sent or received invoices.

This project involves several thousands of laboratories and their customers and suppliers.

The first advance in introducing electronic invoicing was the regulation of SIMFT (Interchange Invoicing System by Telematics Resources), a closed system based on EDI (Electronic Data Interchange) communications and standards. However, only some of the participants in Pharmainnova have an internal system based on EDI. Hence, the SIMFT solution was an advance but was not frequently used because of the closed system, and also because of high implementation costs.

The main problem of invoice exchange is the heterogeneity of the invoice types. Existing technology and methods do not allow to build generic solutions which allow to automatically process any type of invoice. Currently, two main possible measures are applied. Some associations agree to define a common invoicing infrastructure in terms of shared platforms, invoice formats, invoice structures and processes. In other cases, a common invoicing infrastructure is identified and for each partner a specific plug-in is implemented to automate invoice exchange.

The partners of the Pharmainnova association decided to mix these two options by defining a common invoice model but keeping their internal infrastructure. Although this solution brings the benefit that all partners can keep their infrastructure, this process can be costly. For example, when a new member enters the Pharmainnova cooperation, both the common model and all partners' models must be revised. Additionally, possible changes in the law or in the business dynamics enforce updates.

The case study in the NeOn project aims at abstracting away the difficulties of heterogeneous invoice models by applying ontologies. The use of ontologies will guarantee a conceptual framework where the semantics of invoicing will be easily defined and maintained.

Instead of enforcing one single ontology for all partners, from each partner's invoice model a separate ontology is built and consequently mapped to one central reference ontology provided by Pharmainnova. Hence, although most or all partners have different invoice models, the partners do not need to change the invoice model nor the system they are using.

With our work, we address the following uses cases of this case study:

- Build, adapt and visualize the common Pharmainnova ontology as well as the partners' ontologies.
- Define mappings between the common and each partner's ontology.

As the typical end user in this scenario is not experienced in modeling ontologies, we provide the user with the Pharmainnova ontology. The user can adapt this ontology until it reflects the partner's invoice structure, instead of modeling the ontology from scratch himself. Thus, the demanded input from the end users is expected to be considerably lower. When the user finished adapting the ontology, he specifies which elements of both ontologies (the Pharmainnova ontology and the own ontology) can be mapped to each other, for example using an equivalence relation.

In the following sections we discuss the evaluation of our approach. User testing with real end users (i.e., people which will most likely be deploying the tool in the future) is the most fundamental usability testing method. The evaluation of our approach consists of two experiments with real users. One experiment compares our approach quantitatively with the primary other available approach, which is the tree form approach. The classes and properties of an ontology are presented in a tree and additional relationships or attributes can be seen in a separate window per element. The second experiment is qualitative and aims at evaluating whether our tool OntoModel provides the necessary functionalities and options for end users. [Nie93] calls these experiments a *summative* respectively a *formative* evaluation. Sections 9.3 and 9.4 address both experiments in detail.

9.3. Summative Evaluation

The following sections address the methodology, the test users and the tasks as well as the outcomes of our first evaluation.

9.3.1. Methodology

The goal of our first evaluation is to assess the overall quality of our approach. For this purpose, we compared it with the primary other approach available. This approach which is used in (among others) Protégé[8], the most popular currently available tool for modeling ontologies, is the tree from approach. As other existing approaches do not support ontology mappings, the experiment focused on ontologies themselves to evaluate the approach.

A typical method for such general quality evaluation, called a summative evaluation, is a measurement test in which certain measures are collected while a group of test users perform a predefined set of test tasks. Clearly, the time it takes a participant to complete a certain task

[8]http://protege.stanford.edu/.

should be measured in a comparison of different approaches. However, the test would not be valid if we would only measure the time, without looking at other important measures. Not only the time, but also the correctness of the models as well as the level of satisfaction of the users should be measured. The measures time, correctness and satisfaction together form a measure for intuitiveness.

To measure *time*, all participants got a sheet to write down the time on the moment that they finished reading a task description and started to accomplish the task, as well as on the moment that they finished the task. *Correctness* was measured by two ontology experts who checked the test results on soundness and completeness afterwards. For each elementary task, they counted the number of participants per approach that did not complete the task correctly. In doing so, nonrelevant issues such as slightly differently named elements were ignored. At the end of the experiment, all participants filled in a questionnaire about their experience in the test. Using directed questions the participant's *satisfaction* was measured. Appendix A.11 gives the questions of the questionnaire.

Comparing the usability of several systems can be done by between-subject testing, where each participant evaluates only one system, or within-subject testing, where all users test all systems [Nie93]. As within-subject testing has the major disadvantage that the test users can not be considered as representative end users anymore after they learned how to use the first approach, we chose for between-subject testing. The participants were divided randomly into two test groups, each using one of the two approaches in completing the tasks of the experiment.

In the beginning of the experiment, the participants got a short introduction about ontologies and the different approaches, including a small test case to let them become acquainted with the tools and the approaches. During the experiment, a facilitator was available for each group to answer specific questions or provide minimal help.

9.3.2. Test Users

[Nie93] argues that the group of participants in a summative evaluation should consist of at least 20 to 25 persons to achieve a confidence level of 95 %. With this number of test users, variations in various user skills concerning the experiment are smoothed over when randomly dividing the participants over the different groups. Moreover, the group should be homogeneous and the participants should have the same level of experience in the topic. They should represent the end user as much as possible.

For our experiment, we had a group of 24 students of economics attending a lecture 'Knowledge Management' at Universität Karlsruhe(TH). The students do not have much computer science experience but have little experience in (visual) modeling through various lectures they attend. One of the lectures they attended in the beginning of their studies, covered UML, hence they all had some experience in this specific modeling language.

Additionally, all participants knew the invoicing domain but had no particular experience. Moreover, as we conducted the experiment in the beginning of the semester, the lecture 'Knowledge Management' only started some weeks before and ontologies were not addressed yet. The participants did not have any experience with ontologies. Hence generally, our test users represented end users as they had no knowledge about ontologies, little experience in UML, and some knowledge about the invoicing domain.

9.3.3. Tasks

The participants were given an initial ontology containing five classes 'Address', 'Company', 'PaymentMode', 'Invoice' and 'TermOfPayment'. Moreover, the ontology contained three object properties between these classes: a property 'hasAddress' between 'Company' and 'Address', a property 'hasPaymentMode' between 'Invoice' and 'PaymentMode', and a property 'payment' between 'Invoice' and 'TermOfPayment'.

All participants received a sheet with three tasks, all consisting of several small tasks to expand the given ontology. The task descriptions guided the users by giving hints on how to accomplish the task, so that all test users modeled the same ontology in the end and correctness could be checked easily. The task descriptons as they were given to the participants, are presented below.

First task The original ontology is very simplified. We will now add some more details about a company. Currently, a property 'hasAddress' is defined between the class 'Company' and the class 'Address', denoting that a company has an address. We want to specify the street and city of an address. For this we need properties. As the values are just strings, we use data properties. Define two data properties for the class 'Address', called 'hasStreet' and 'hasCity', that both have strings as values.
Additionally, we want to have some specific companies in our ontology. Such instances are called individuals. Define two individuals called 'Kin' and 'Aece', both being instances of the class 'Company'.

Second task We want to explicitly specify that the two companies, 'Kin' and 'Aece', are really different companies. Define the relation that both individuals are different.
To specify the address for the company 'Kin', we define an individual 'KinAddress' as an instance of 'Address'. Define a property instantiation for the property 'hasAddress' from the individual 'Kin' to its address.

Third task Finally, a company wants to distinguish between emitted invoices and received invoices. Define two subclasses of the class 'Invoice', called 'EmittedInvoice' and 'ReceivedInvoice'.
We want to define that an emitted invoice is received by some company, whereas a received

invoice is sent to a company. Define a property 'hasReceiver' from the class 'EmittedInvoice' to the class 'Company', and a property 'hasSender' from the class 'ReceivedInvoice' to the class 'Company'.
Additionally, change the name of the class 'TermOfPayment' to 'PaymentDetails'.

9.3.4. Outcomes

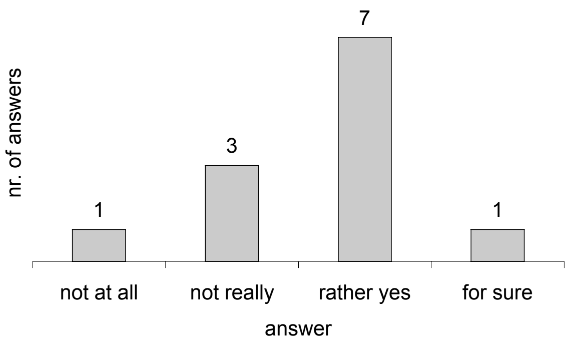
A first measure, correctness, was measured by the experts after the test users completed the tasks. The results showed that all except one elementary task were completed properly by about all participants in both groups. The task to specify two individuals as different from each other, which was part of the second task in the experiment, could only be finished by few of the participants evaluating the tree form approach (Protégé) but was finished correctly by all participants evaluating our UML approach. Possibly, this can be attributed to the fact that the tree form approach does not give a clear overview of relationships between elements whereas with the UML approach the user sees all elements, and can define that the two individuals are different by dragging the appropriate dependency arrow between them.

The time measure was calculated using the times the participants wrote down before and after each task. There were no real differences in time needed by the test users to complete the tasks. The very small existing variations rather seemed coincidental and appeared within one group as much as between the groups. Hence we can draw the conclusion that there is no big difference in time needed to accomplish certain tasks in both approaches. However, this conclusion does not count for the elementary task of specifying that two individuals are different from each other as this task was not finished correctly by most of the participants evaluating the tree form approach. Possibly, more time would have been needed to correctly accomplish this task using the tree form, if the test users would have known that what they did was incorrect.

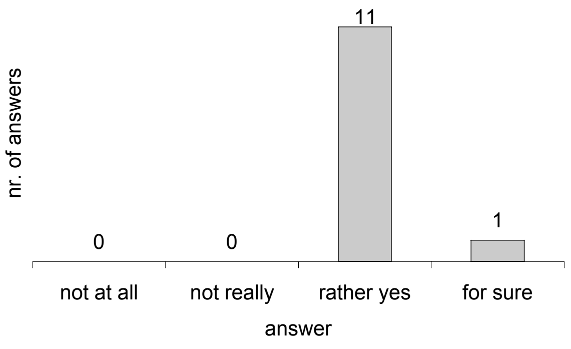
	low	**average**	**high**
Task 1			
Tree form approach	7	5	0
UML approach	8	4	0
Task 2			
Tree form approach	5	4	3
UML approach	3	7	2
Task 3			
Tree form approach	2	6	4
UML approach	2	10	0

Table 9.1.: Difficulties needed to overcome in order to complete the different tasks

The third measure we took into account, is the satisfaction of the participants in the experiment. For this, we rely on the results of the questionnaire. The questionnaire contained general questions about the tasks themselves, questions about the usability of the approach, and questions about its effectiveness and efficiency. Below we discuss the answers of the participants of both groups.



(a) Tree form approach



(b) UML approach

Figure 9.4.: Intuitiveness

Three questions were asked about the difficulties to complete the different tasks. Table 9.1 shows the answers of the participants. At first view, it looks like the difficulties needed to overcome the first two tasks were received about similar by both approaches. However, the correctness measure showed that the second task was not done properly by the participants using the tree form approach. Hence, although they had the feeling that it was not difficult, they did not achieve the right result. The third task was clearly perceived easier by the participants using the UML-approach.

	not at all	not really	rather yes	for sure
Tree form approach	1	6	5	0
UML approach	0	2	8	2

Table 9.2.: Satisfaction with notation for ontology elements

Figure 9.4 illustrates the general intuitiveness of both approaches as experienced by the test users. The UML approach was perceived as intuitive by all, whereas the tree form approach was evaluated as 'not intuitive at all' by one user and 'not really intuitive' by three others. This means that 33 % of the the participants evaluating the tree form approach, did not find it intuitive. This measure can probably be attributed to the fact that the participants already used UML before, but also to the fact that the tree form approach gives the user only a limited overview of the relationships between different elements and specific windows have to be opened to maintain relationships.

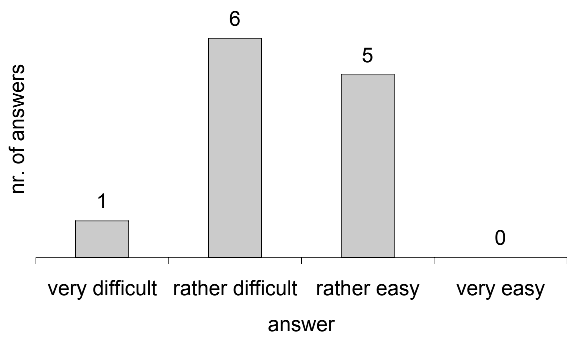
	not really	rather yes	for sure
Tree form approach	3	6	3
UML approach	0	9	3

Table 9.3.: Clear and simple sequences of steps to accomplish an action

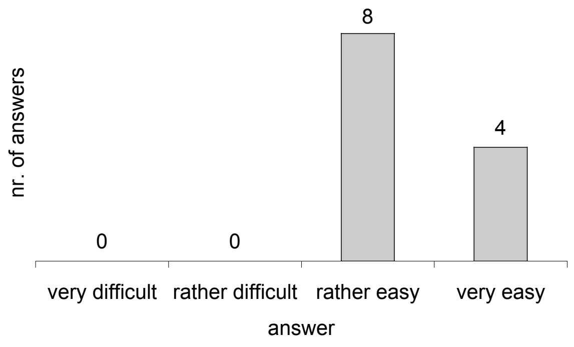
Table 9.2 shows the answers on the question whether the participants were satisfied with the choice of the specific notations for the different ontology elements. None of the two test groups received only positive answers on this question. However, in the group evaluating the UML approach only two were not happy with the notations whereas in the group evaluating the tree form approach, the negative answers counted up to more than 50 %. Two people in the UML group were even very satisfied. The satisfaction level in this group could be related to the fact that with our UML notation we tried to reuse the existing UML constructs as much as possible to provide the user with less new notations or new intentions for known notations.

Our experiment also showed that users find it easier to get acquainted with an ontology using the UML approach than with the tree form approach. Figure 9.5 illustrates the division of the different answers on the question how difficult they found to get acquainted with the given ontology. With the UML approach, all participants found it rather or very easy, whereas with the tree form approach it was perceived difficult by more than half of the participants. The main reason for this is that the UML approach gives one overview for all elements, their attributes as well as their relationships to other elements, whereas the tree form approach only gives a high-level overview of some elements without immediately showing all relationships.

Another question in the questionnaire asked the participants whether they find that the



(a) Tree form approach

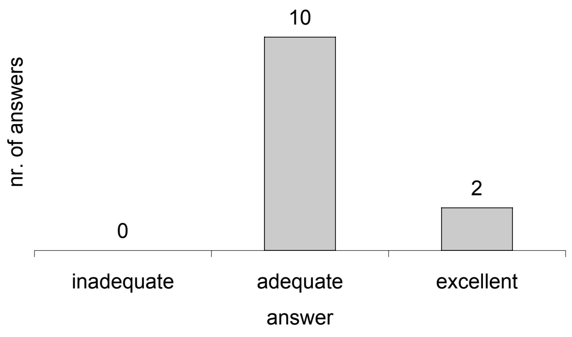


(b) UML approach

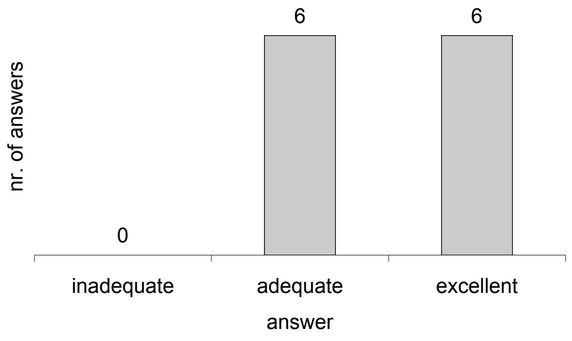
Figure 9.5.: Acquaintance

editor allows to set a clear and simple sequence of steps to accomplish each necessary action, e.g. create a new instance of a concept. In the group evaluating the UML approach, all users found the step sequences rather or very clear and simple, whereas in the other group, three test users answered negatively. Otherwise, the participants in both groups gave the same answers. Table 9.3 gives an overview of the answers in both groups. The difference in this measure can be attributed to the fact that in the tree form approach, often a new window has to be opened to see or maintain constructs or relationships, even in case of simple elements because element details can not be represented in the tree.

Finally, Figure 9.6 illustrates how the participants of both groups perceived the effectiveness of the approach. Whereas two of the users evaluating the tree form approach perceived it as excellent and the other ten as adequate, in the group evaluating the UML approach, half of the participants found the effectiveness excellent and the others adequate.



(a) Tree form approach



(b) UML approach

Figure 9.6.: Overall effectiveness

The main reason for the generally more positive answers on all questions in the group evaluating the UML approach, is probably the available experience with UML as well as the fact that the UML approach provides one overview for all elements and their relationships, while the tree form approach has very limited possibilities for showing and maintaining relationships between elements.

9.4. Formative Evaluation

This sections addresses our second evaluation. We discuss the methodology, the test users, the tasks as well as the outcomes of the experiment.

9.4.1. Methodology

With the second experiment, we evaluate whether our approach suffices for the needs of end users in the Pharmainnova case study, again using the implemented prototype. The goal of this formative experiment is to learn which detailed aspects of our approach are good and bad, and how the overall approach can be improved.

A typical method to use for formative evaluation is a thinking-aloud test. It involves having one user at a time use the tool for a given set of tasks while being asked to think out loud, accompanied by an expert who observes the experiment. By verbalizing their thoughts, the test users allow the observer to determine what they are doing and why they are doing it. In doing so, the observer could derive which actions from the participant were actually incorrect for what he wanted to achieve, and then guide the participant. Additionally, the observer could make notes of remarks and feedback of the test users during the experiment.

To obtain the necessary measures, the participants were asked to fill in a questionnaire about their experience with the approach after the experiment. Using directed questions on the usability, effectiveness and efficiency of the approach, the participant's *satisfaction* was measured. Appendix A.12 gives the questions of the questionnaire. As the participants were guided by the observer in completing the tasks correctly, we did not measure the correctness of the results afterwards.

In the beginning of the experiment, the participants got a short introduction on ontologies, UML and the tool, including a small test case to let them become acquainted with the tool and the approach. While completing the tasks, the observer was available for answering specific questions.

9.4.2. Test Users

To test whether our approach fulfills the needs of the target group, it is not necessary to have a big group of test users. The more severe usability problems are typically detected by the first few participants, and 80 % of all usability problems is detected with 4 or 5 participants [Vir92]. The group of test users should be representative and cover the target population.

We conducted the experiment with five participants of which two were real end users, namely employees of one of the Pharmainnova members. Three other participants were people leading the e-invoicing project of Pharmainnova. Although they are not real end users themselves, they are very well aware of the experience and the needs of the end users.

None of the participants had any experience with ontologies. It was totally new for them to build or even just see an ontology, and they did not have any idea of the existing ontology constructs and their intention. However, they were all experienced in the invoicing domain, although one of them only little. Two of the participants were actually very experienced in invoicing. Hence all five participants were understanding the task domain. As regards UML, one of the test users heard about UML but had no knowledge about the notation or about how to model diagrams. The other four test users had some or much experience with UML. Thus, generally our group of test users was representative and covered the target population.

9.4.3. Tasks

In the experiment, the participants were provided the tool OntoModel with the header-part of the Pharmainnova ontology. This part of the ontology contains the different elements in an invoice header, and their relationships. Next to the data about the invoice emitter and the invoice receiver such as various address fields, contact data, internal code and tax number, an invoice header contains some data about the invoice itself such as the currency, the invoice number and the type of invoice.

Next to this part of the Pharmainnova ontology, the participants received a document describing a Pharmainnova partner's invoice model in the form of a table. The table gave an overview of the company's invoice structure by describing all invoice fields with their name, their format and in some cases an informal description of the field. All participants received the same document[9]. The differences between the partner's invoice structure and the model of the Pharmainnova ontology consisted of a different language (the Pharmainnova ontology is written in English whereas the partner's invoice structure was written in Spanish), elements that were available in one but not in the other model, elements that were structured or related differently such as street and street number together in one field or in two separate fields, or elements that were a special type of a specific element in the other model.

The experiment consisted of two tasks, which are described below.

First task Based on the table of the partner's invoice structure, the participants adapted the header-part of the Pharmainnova ontology until it reflected the partner's invoice structure. They changed the names or types of elements, deleted elements, and added new elements.

Second task After completing the first task, the header-parts of both the Pharmainnova ontology and the partner's ontology were depicted in the mapping mode of the tool, and the participants were asked to model the semantic relationships between the two ontologies as ontology mappings.

9.4.4. Outcomes

In this evaluation we measured the participants' satisfaction by asking questions on the usability, effectiveness and efficiency of the approach in the questionnaire. Additionally, we

[9]Note that not all Pharmainnova partners' invoice models have the same complexity. We explicitly asked Pharmainnova to give us a document with a representative complexity.

became feedback and remarks during the experiment by letting the test users think out loud. All outcomes that are discussed below are based on the answers given by the users during or after the experiment.

In general, the questions of the questionnaire were answered positively. The questions on usability mostly got a negative answer from one participant. This can be attributed to the fact that there was one participant that did not have any experience with UML. However, on the question on the overall effectiveness of the approach, all participants so also the one without UML experience, answered positively. We discuss the different outcomes below.

Table 9.4 gives the answers on the questions asking for the difficulty of completing the different tasks. Four participants specified the difficulty of both tasks as 'low', whereas the

Task 1	**low**	**average**	**high**
	4	1	0
Task 2	**low**	**average**	**high**
	4	1	0

Table 9.4.: Difficulties needed to overcome in order to complete the different tasks

fifth participant answered twice 'average'. During the experiment it became clear that the participant without UML experience found it more difficult to accomplish the tasks. However, he did not find the difficulty high but average. As all participants were unexperienced with ontologies, we believe that this could be a first positive result.

Figure 9.7 presents the participants' answers on the question about the intuitiveness of our visual approach to model ontologies. Although most found it rather intuitive, again one partic-

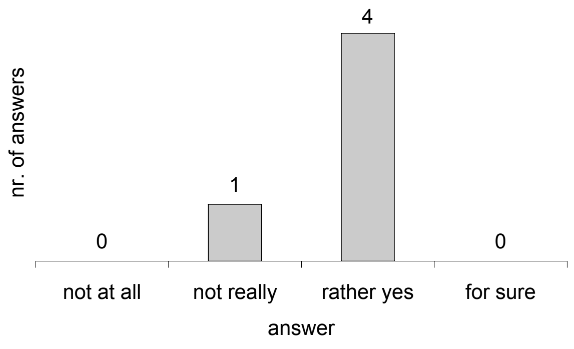


Figure 9.7.: Intuitiveness of ontology modeling

ipant found it not really intuitive. A similar result we received on the question on how difficult it was to get acquainted with the given ontology. Figure 9.8 shows that two participants found this very easy and two found it rather easy, but one found it rather difficult. A possible reason

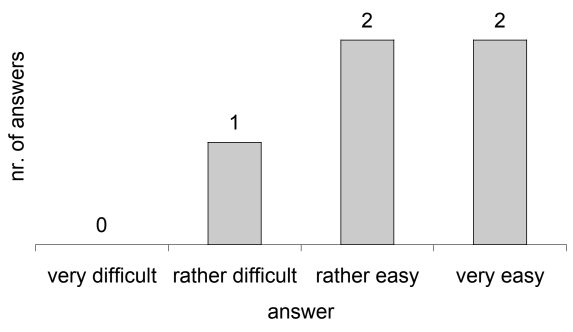


Figure 9.8.: Getting acquainted with the given ontology

for this is that it can be difficult to apply the UML concepts to the model of invoices, as was indicated by one of the participants during the experiment. Another remark from a participant was that it was difficult to understand that there are different ways to express the information to appear in an invoice.

We could improve these issues by providing the user more guidance in modeling the different kinds of information.

As for our visual approach for modeling ontology mappings, all participants perceived it as intuitive. One answered to find it very intuitive. Figure 9.9 presents the results on this question. The difference with the perceived intuitiveness for modeling ontologies themselves can probably be explained by the fact that the additional visual constructs for mappings are limited and mainly only consist of arrows with an icon.

Although the participants found the notation for mappings intuitive, they got confused by all the arrows when several to many mappings were defined. They talked about several possible improvements for the visualizations of the mappings. Two participants had the idea of giving mapped element certain colors and leaving out the arrows. Another participant thought of sticking to the way of dragging arrows, but he proposed to not display the arrows but displaying the mapping in a list on the side of the window immediately after defining it. A last recommendation from a user was to leave the concept of arrows as it is, but give the user the possibility to hide an arrow, comparably with the possibility of hiding the attributes of a class as provided in many existing UML tools.

Figure 9.10 gives the participants' answers on the question whether they find the UML-notation suitable for modeling ontologies. Three users found it 'rather suitable' whereas two found it 'for sure suitable'. So also the user without experience with UML found the notation

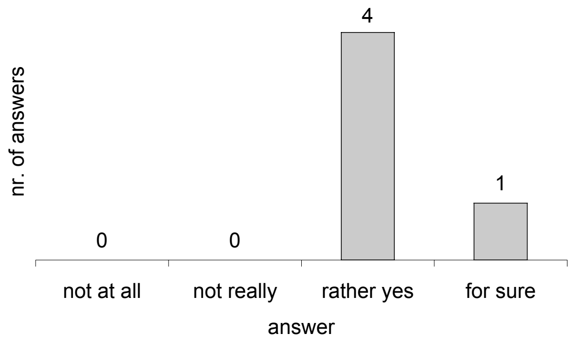


Figure 9.9.: Intuitiveness of ontology mapping modeling

suitable.

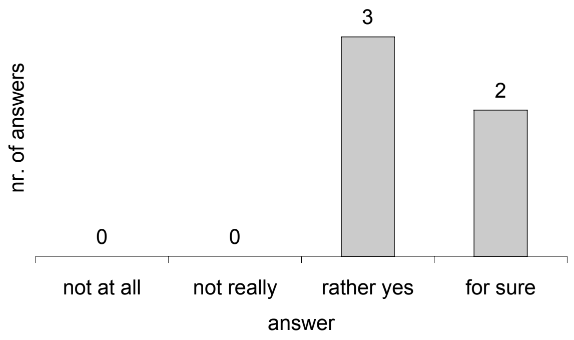


Figure 9.10.: Suitability of the UML-notation for modeling ontologies

Finally, Figure 9.11 presents the answers on the question how the participants perceived the overall effectiveness of the approach. The answers, three times 'excellent' and two times 'adequate', show that even the participant who had more difficulties to complete the tasks as he did not have any UML experience found the approach in general effective.

During the experiment, two participants recommended to provide a tool that hides all options that are rarely used, so the user can focus on the main and basic concepts. Although our prototype implementation already hid many of these rarely used constructs in the main menu, we learned that we should hide even more constructs when applying our approach. More experienced users should be able to extend the possibilities as they want.
We also noticed during the experiment that the users often wanted to rely on the known right-click menu. However, our prototype only provided few options in this menu. When extending

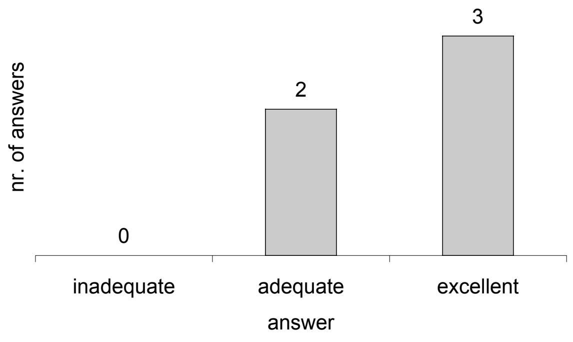


Figure 9.11.: Overall effectiveness of the approach

the prototype implementation, these right-click menus should be more complete to allow the users to use functionalities they are used to.

The answers on the last question let us conclude our evaluation with a generally positive result. The evaluation outcomes clearly show the advantage that persons with UML experience have, and that persons without UML experience still are happy with the approach, although it takes them some more time and effort to model ontologies. The participants gave us several interesting ideas for improving our approach in the future.

9.5. Conclusion

We implemented our approach as a prototype called OntoModel, built on Eclipse. OntoModel implements our UML profile for ontologies and ontology mappings based on their MOF metamodels. OWL ontologies can be imported, and ontologies as well as ontology mappings can be modeled and exported in a textual syntax.

Using the prototype, we conducted two evaluations. First, a summative evaluation was conducted to assess the overall quality of our approach by comparing it with the most important other approach currently available for modeling ontologies, the tree form approach. The outcomes of the evaluation show that generally, the UML approach was perceived as more intuitive by the test users. We believe that the available UML experience of the test users takes part in this result. However, we also believe that the few overview possibilities with the tree form approach, and the many separate windows that have to be utilized to see or maintain the various details and relationships of ontology elements, which is independent of the experience with the approach, can give the user an unsatisfactory feeling.

The second evaluation we conducted was a formative evaluation, to assess whether our approach suffices for the needs of end users in the Pharmainnova case study. On some questions, one participant gave a slightly negative answer whereas all others gave a positive answer. This can be explained by the fact that one participant, in contrast to all others, did not have any UML experience. However, on questions like the one asking about the overall effectiveness of the approach, all participants answered positively. The outcomes of this evaluation proof the benefit that users have when they have UML experience, and that our approach is also effective for users without UML experience, although they have more difficulties.

The two evaluations give a clear indication that our approach is a suitable solution to allow employees of companies to model and maintain ontologies and hence benefit from semantic technologies, especially when the users have some experience with UML. Additionally, the second evaluation provided us with some possible improvements of the notation.

10. Related Work

In recent years, a considerable amount of effort is spent in approaching the visual modeling paradigm for ontologies as well as in providing MOF support for ontologies. This chapter introduces such approaches related to our work. Section 10.1 addresses existing ontology support using MOF and Section 10.2 highlights existing work towards general support for visual modeling of ontologies. Section 10.3 concludes the chapter.

10.1. Specific Ontology Support based on MOF Metamodels

In early 2003, the OMG requested a MOF-based metamodel and a UML profile for the purpose of defining ontologies, called Ontology Definition Metamodel (ODM) [Obj03]. In answer to this request, four proposals were submitted. [IBM03] defines stereotypes for representing the new aspects introduced by ontologies in UML. In our opinion, the proposal has several weaknesses. For example, OWL properties are visually modelled as UML classes instead of mapping them to UML associations. Similarly, the different kinds of class constructors found in OWL are reflected by associations, while we believe that an appropriate specialization of UML classes is more appropriate. [Gen03] follows the RDF serialization of OWL documents and represents the ontology in a graph-like notation that is close to the actual (ambigous) RDF representation. Also various kinds of OWL properties are modeled as a hierarchy of UML classes. Complex OWL class definitions, i.e. using union and other operators, are modeled as comments associated to a class. [IKSL03] departs from the OMG request and introduces a metamodel for the Open Knowledge Base Connectivity (OKBC) standard. [DST03] suggests the OWL Full language, but neither provides a visual syntax nor introduces a metamodel.

The various proposals have been merged into one submission [Gro06b, CRH$^+$06] that covers metamodels for RDF, OWL, Common Logic [Del06], and Topic Maps [GM05], as well as transformations between some of them and a UML profile for RDF, OWL and Topic Maps. At the moment of writing this document, the recommendation is undergoing the finalization process for standardization. Two implementations accompany the proposal and support the visual modeling of ontologies with import and export functionality for OWL: VOM (Visual Ontology Modeler)[1], commercially developed and maintained by Sandpiper inc., and IODT (Integrated Ontology Development Toolkit)[2], provided partially open-source by IBM.

[1]http://www.sandsoft.com/products.html.
[2]http://www.alphaworks.ibm.com/tech/semanticstk.

While obviously the scope of the proposal is quite extensive, the metamodel for OWL concentrates on its RDF serialization. Moreover, the OWL metamodel builds on the metamodel for RDF, hence it can only be used in a combined manner. Our approach departs from the OMG approach in two aspects. Firstly, we do not only support ontologies but also other aspects of ontologies, namely rules and ontology mappings. Secondly, we focus on the new version of OWL and provide an autonomous metamodel for it, which can be adapted or extended according to general changes of the language specifications in the future.

[GDD07] describes another effort inspired by the OMG request for an Ontology Definition Metamodel, although not officially submitted. The contribution consists of a metamodel for OWL 1.0 with an accompanying UML profile, both designed following the MOF approach but with different design considerations than our work and the ODM of OMG. While a prototype implementation is given, the different parts of the approach are not integrated and an external component is used for the OWL import and export.

As part of the research project REWERSE[3] funded by the EU Commission and Switzerland within the Sixth Framework Programme, the rule language R2ML [WAL05] is being developed. R2ML integrates OCL, SWRL and the Rule Markup Language (RuleML, [WATB04]), but although it is meant for enriching ontologies with rules, it is not like SWRL developed for use on top of OWL. For model transformation between R2ML and SWRL, a MOF metamodel is defined[4]. Additionally, [LW06b] demonstrates a UML-like syntax for R2ML called URML, which however does not consist of a UML profile but an extension of the UML metamodel. Hence, existing UML tools can not be reused as in the case of UML profiles, and it is questionable whether the notation is still intuitive for UML-users, as several new visual constructs are introduced. [LW06a] presents Strelka, an editor for the proposed visual language providing serialization into the R2ML XML format. The implementation does not incorporate the MOF-based metamodel for R2ML.

To the best of our knowledge, our work presents the only currently available MOF-based metamodel and accompanying UML profile combining ontologies, rules and ontology mappings. Moreover, we believe that our approach to abstract from RDF in the OWL metamodel is necessary to provide a more adequate metamodel for OWL.

10.2. General Visual Syntaxes for Ontology Languages

The utility of a visual syntax for modeling languages has been shown in practice and visual modeling paradigms such as the Entity Relationship (ER) model or UML are used frequently for the purpose of conceptual modeling. Consequently, the necessity of a visual syntax for KR

[3]http://rewerse.net/.

[4]http://oxygen.informatik.tu-cottbus.de/R2ML/0.4/metamodel/R2MLv0.4.htm.

languages has been argued for frequently in the past [Gai91, Kre98]. Particular representation formalisms such as conceptual graphs [Sow92] or Topic Maps are based on well-defined graphical notations.

Description Logic-based ontology languages such as OWL, however, are usually defined in terms of an abstract (text-based) syntax, and most care is spent on the formal semantics. The earlier absence of a visual syntax[5] has lead to several proposals. [Gai91] proposes a particular visual notation for the CLASSIC description logic. Newer developments have abandoned the idea of a proprietary syntax and propose to rely on UML class diagrams. [CP99] suggests to directly use UML as an ontology language, whereas [BKK$^+$01] proposes to predefine several stereotypes such that a more detailed mapping from UML to the primitives offered by the DAML+OIL description logic can be achieved. [BKK$^+$01] further argue that the UML metamodel should be extended with elements such as property and restriction such that UML is more compatible with knowledge representation languages such as OWL.

The tool that gained the most visibility in the current ontology community in the past years is Protégé. It supports visualization and editing of ontologies in RDF and OWL as well as rules in SWRL in a tree form approach. Protégé is free and open source and enables users to load and save ontologies and rules, edit and visualize classes, properties and rules, define logical class characteristics, edit individuals and execute reasoners. Protégé is supported by a strong community of developers and academic, government and corporate users. However, many of its users dislike the fact that it still requires a significant knowledge of ontology and rules modeling [DMB$^+$06], and the imposed tree form still considerably restricts the visualization options.

A UML back-end plug-in provides an import and export mechanism between UML class diagrams and the Protégé knowledge model. However, as the UML notation is the standard class diagram notation without any adaptation, only those ontology elements that can be expressed in UML are supported.

An ontology editor applying the tree form approach of Protégé and allowing the user to build ontologies using DAML+OIL is provided by OilEd [BHGS01], initially only intended to demonstrate the use of the OIL language. It does not provide a full ontology development environment in the sense that it supports migration and integration of ontologies or many other activities that are involved in ontology construction, but is rather a simple ontology editor. Although OilEd can still be downloaded and used, it is no longer maintained.

KAON [BEH$^+$02], an open-source ontology management infrastructure, includes a graphical ontology browser for creating, editing and maintaining ontologies called OI-Modeller. The user interface appears to be a mix between simple 2D graph-tree visualisation based on

[5]Which can been seen as a direct result of the criticisms about the semantics of early diagrammatic semantic networks [Woo75, Bra79].

the TouchGraph library[6] and the approach of both OilEd and Protégé. The difference with our approach is not only the visual notation, but also the ontology export. The OI-Modeller stores ontologies in a specific KAON formalism.

The ontology engineering environment OntoStudio[7] combines the tree form approach with some arrow-based approach. Ontologies can be imported or can be modeled in tree form. With the import-function, existing data structures can be transferred into an ontology. These data structures can be existing ontologies, or other kinds of data structures such as database schemas. Using the export function, ontologies can be stored in the ontology formats OWL, RDF(S) as well as F-Logic.

OntoStudio allows the visualization of ontologies by showing the concepts and their subconcepts in a graph. Additionally, the Graphical Rule Editor provides a means to formulate rule constructs and model them graphically. Rule diagrams are converted into simple F-Logic rules. Also F-Logic queries can be defined. The plugin OntoMap extends OntoStudio and allows to construct dependencies between heterogenous constructed or imported ontologies graphically using an arrow-based approach.

In the EU project NeOn[8], OntoStudio is further developed as an open source toolkit, of which the reference architecture is fully compatible with our metamodel. The toolkit offers an Eclipse-based plug-in infrastructure, of which OntoModel will be one of the plug-ins.

[FW05] presents a visual approach for modeling OWL ontologies based on the Microsoft Visio[9] notation. The graphical notation is implemented as a commercial, simplified OWL editor called SemTalk for using semantic web knowledge for process models. Due to the specific notation, the approach lacks the flexibility in the sense of both reusing existing tools and present user experience.

Finally, TopBraid Composer[10] is an ontology development environment supporting RDF Schema, OWL and SWRL. It creates customizable forms based on class definitions as its primary editing mechanism for ontologies. Syntax highlighting and auto-completion is provided when editing complex class definitions and rules. Rules are executed with an internal rule engine to infer additional relationships among resources. The class definitions of an ontology can be created by importing UML class diagrams which are converted on a class-by-class basis.

Topbraid Composer has two ways of visualization. Relationships among ontology resources can be visualized as a graph, whereas class definitions can be illustrated in diagrams similar to UML class diagrams. As for mappings between different ontology models, they can be visu-

[6]http://touchgraph.sourceforge.net/.
[7]http://www.ontoprise.de/.
[8]http://www.neon-project.org/.
[9]http://office.microsoft.com/visio/.
[10]http://www.topbraidcomposer.com/.

alized using the class diagram notation. A class box in between mappable elements contains the textual syntax of the mapping.

Topbraid Composer does not support all ontology elements in the UML-like notation, and users can still not rely fully on their UML-experience as several new notations are used.

10.3. Conclusion

This chapter discussed the primary existing approaches related to our work. We surveyed several other MOF-based metamodels for ontology languages that were developed during the last years. However, none provides an independent metamodel supporting the latest version of OWL. Moreover, a combination of OWL with rules or ontology mappings is missing.

As for visual paradigms for ontologies, a very popular one is the tree form approach used in several applications, although users ask for even more visualization. Some first UML-like approaches, although not fully UML-compatible or not supporting the full ontology language yet, appear to be prefered by the users. Our UML profile is the first and currently only available UML-based notation fully supporting OWL ontologies and moreover combining them with SWRL rules and OWL ontology mappings.

11. Conclusion and Outlook

11.1. Summary

Being aware of the identified support needs for companies in utilizing ontologies and of the successful methodologies and technologies in software engineering, the central question we posed in the beginning of this work was: *How can ontology engineering be supported using existing software engineering methodologies and technologies?* In order to be able to answer this question, we started with an in-depth analysis of the MDA as a successful initiative in the field of software engineering to support model-driven engineering of systems. In particular, we analyzed two of its main components, UML and MOF. Further we identified and introduced the main existing languages for representing ontologies, rules and ontology mappings.
To respond on our central question, we applied MDA for ontologies, to support both model driven ontology development and automatic model transformations between different ontology languages. In the following we summarize the outcomes of our work along the two main questions into which we divided our central question, as well as the lessons we learned from our work.

How can existing approaches in software engineering be applied to support the use of different ontology languages by different partners? A broad range of different ontology languages is currently available and thus we started with asking the partial question: *Which are the ontology languages to be supported?* As a response on this question, we decided to take the standardized and most well-known ontology language OWL as the main language in our work. To support rule-extensions for OWL ontologies, we selected SWRL, a rule language built to use on top of OWL. As a second language we chose the rule-based ontology language F-Logic, not standardized but very known in many companies. We did not want to limit our approach to these languages for rule-extended ontologies, but chose to provide specific support for them with a general solution.
After we had answered this first partial question, we addressed the second partial question: *How can models defined in one language be automatically transformed to another language?* While answering this question, we also considered the third partial question already: *How can an extensible solution be provided, supporting the current semantic web area in which it is not in every case clear yet which language will be the (de facto) standard in the future?* As an answer to this question, we developed a solution based on the metamodeling framework

MOF. In the software engineering field, MOF is successfully used to enable the development and interoperability of model and metadata driven systems. To benefit from this success story in the field of ontology engineering, we presented MOF-based metamodels for OWL, SWRL and F-Logic. Such a metamodel refers to a language, whereas the instances of the metamodel are referred to as models. Here, models thus refer to concrete ontologies and rules. Next to a full specification of the metamodels including OCL constraints and accompanied with UML diagrams, we provided the language mappings from the metamodels to the respective languages by translating the terms of the metamodels to these of the formalism and so providing semantics for the metamodels.
Both metamodels are defined in MOF, hence they can be applied in the MDA framework to allow automatic transformations of F-Logic models to OWL and SWRL, and vice versa[1]. In doing so, trading partners using different ontology languages are able to cooperate without having to adapt their internal system or language in use. We provided some first model transformations from the core part of an ontology from F-Logic to OWL.

Our application of MDA to ontologies and their formalisms directly supports the use of different ontology languages by different partners. The modular approach of MOF provides us with an extensible solution, where new modules for additional ontology languages can be added in a straightforward manner when needed. As such, we answered both the second and the third partial question.

How can existing approaches in software engineering be applied to support the modeling of ontologies, rules and ontology mappings? This second main question immediately lead us to a first partial question: *Which requirements do users have that are not familiar with textual syntaxes of ontology languages?* As an answer to this question, we found that employees need an approach that is abstracted away from specific ontology issues as much as possible, and that moreover relies on an intuitive paradigm, preferably one which they are already familiar with.
Next, we posed the second partial question: *How can manual work be avoided, resulting in fewer errors and hence improving overall quality?* For easier modeling of ontologies, rules and ontology mappings, we rely on a visual modeling paradigm. Visual syntaxes have shown to bring benefits in many other areas. Visual modeling of ontologies in particular decreases syntactic and semantic errors and increases readability.
After we answered the second partial question and decided to provide a solution based on the visual modeling paradigm, the third partial question came up: *How can we rely on modeling paradigms with which users are generally familiar?* During the last decade, the visual language UML became very known and since, it plays an important role for conceptual mod-

[1]The open world semantic of OWL does not fit together with the F-Logic semantics, and unrestricted transformations between both languages will not be possible. However, possible transformations in a certain extent are desirable.

eling in many companies. Although first, several proposals came up for new particular visual notations for ontology modeling, these ideas have been abandoned in favor of proposing to rely on UML. The UML extension mechanism allows us to adapt the language for ontology modeling. Utilizing UML methodology, tools and technology seems to be a feasible answer to this partial question.
Further, we asked the partial question: *How can shortcomings from existing approaches for ontology engineering be considered?* To respond to this question, we looked at a recent comparison between the main existing approaches. One important result of this evaluation was that users prefer visual over nonvisual notations. This issue we addressed by answering the former question. But moreover, users did not like having to cope with specific ontology aspects. This issue we addressed in our last question:
How can be abstracted away from specific ontology language aspects when modeling ontologies? Partially, this question was already answered after having decided to rely on the visual modeling paradigm. Additionally, the UML profiling mechanism provides a very flexible way of defining notations and allows to choose an abstraction level as necessary for the application domain.
After having answered the five partial questions, we developed a UML profile for OWL ontologies as well as for SWRL rules and OWL ontology mappings as a solution for the second main question. This profile is based on the previously defined metamodel for OWL and SWRL, and a newly defined metamodel extension for ontology mappings.

Lessons Learned To show how our approach can be applied in practice, we implemented a prototype called OntoModel. OntoModel currently supports model driven development of ontologies by allowing the user to model ontologies and ontology mappings using the UML syntax. With two evaluations that we conducted using this prototype, we have shown the practicability and usability of our approach as received by (end) users.
Through our work, we proved that existing methodologies and technologies in software engineering can successfully be used to support the use of ontologies. By providing MOF based metamodels for the primary ontology languages as well as some first transformations between these metamodels, we showed how business partners using different ontology languages can bring together their knowledge based on existing MDA applications. Furthermore, we showed how employees can be supported in representing their company's knowledge in the form of ontologies, rules and ontology mappings by providing a visual notation based on UML.
We expect that our approach can contribute very well to the ability of companies to benefit from semantic technologies. We can now use a large array of industrial strength tools that is available for UML and other related OMG standards for the purpose of ontology development. Besides graphical editors and model transformation tools, other kinds of utilities offer further benefit. For example, we can utilize the Eclipse Modeling Framework to derive a Java API for OWL directly from the metamodel. We hope that the interoperability with existing software tools and applications will ease ontology development and thus contribute to the adoption of

semantic technologies and their success in real-life applications.
Our work is a first step to bring the W3C vision of a Semantic Web technology and the OMG vision of a Model Driven Architecture together. Both distinguish essentially the same levels of data and metadata. While the primary purpose of this work is to enable ontology development and maintenance with MDA technologies and tools, the bridge could also be exploited in the other direction as much potential is created by the flow of capabilities of the semantic web into the software development environment.

11.2. Open Questions and Future Directions

With the work we presented, we started to build a bridge between the semantic web area and the software engineering community. A variety of issues are still open and several interesting aspects can be explored in the future. In the following, we briefly discuss possible extensions of our work.

Representation of context Ontologies are produced by individuals or groups in certain settings for specific purposes. Therefore, they can almost never be considered as something absolute in their semantics and are often inconsistent with ontologies created by other parties. Each ontology can be viewed as valid in a certain context. In order to fully utilize ontology networks, a solution exists in representing context information, including different real-word interpretations and the inherent conceptual differences between them. Such context information can then be filtered and used directly in the reasoning process. Contextualization is currently heavily being researched [BGvH$^+$03, GMF04]. Our presented metamodel could be refined to deal with the explicit representation of such contexts.

Extended model transformations from F-Logic to OWL and SWRL, and vice versa Our work provides model transformations from the taxonomy-part of an F-Logic ontology to OWL. The full correspondence between the languages is not fully defined yet but is currently being researched [MHRS06, MR07]. When these investigations come in a further stage and allow to define semantically compatible mappings between the two languages, the transformations between the MOF-based metamodels could be extended to cover larger parts of the two languages.

Ontology modularization support Similar to modules in software engineering, self-contained and reusable ontology modules can be defined to improve clarity, reusability and exensibility [dSM06]. Modules are significant components of ontologies that are defined and managed independently from the ontology they are included in. In contrast to big and non-modularized ontologies, modules are easy to understand. Moreover, they ease collaborative ontology design and maintenance. Many formalisms for handling ontology modules have been proposed [BS03, SK03, SP04]. However, they have their own limitations in some scenarios [SR06, BCH06].

A future extension of our metamodel could support the definition and management of ontology modules. The modules themselves could be covered by a metamodel extension directly built on the metamodel for ontologies and the metamodel for ontology mappings could be extended to define the connections between different modules. A first version of such metamodel extension for modular ontology design is presented in [HRW$^+$06].

Support for other rule languages While we consider SWRL as the most relevant rule extension for OWL, other rule languages may become relevant as well. 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 by our metamodel. Using additional constraints it can be checked whether a rule is DL-safe. It should be noted that SWRL is not the only rule language which has been proposed for ontologies. Other prominent alternatives for rule languages are mentioned in the W3C Rule Interchange Format Working Group charter [W3C05b], namely the Web Rule Language WRL [ABdB$^+$05], the rules fragment of the Semantic Web Service Language SWSL [GKM05], the Rule Markup Language RuleML [WATB04] and the language R2ML [WAL05]. 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 define different metamodels, each of which is tailored to a specific rules language.

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 the SWRL metamodel defined here by augmenting it with some features to allow for the modeling of language primitives which are not present in SWRL. As a result, one would end up with a common metamodel for different rule languages, similar to the common metamodel we provided for OWL ontology mappings. An advantage of the latter approach would be a gain in flexibility.

Extended support for F-Logic In the scope of our work, the support for F-Logic is limited to the definition of a MOF-based metamodel together with its language mappings, and initial transformations to the metamodel for OWL and SWRL. We decided to focus on OWL in our work and did neither provide a metamodel extension for F-Logic ontology mappings, nor support F-Logic in the UML profile. Both are interesting possible extensions to allow F-Logic users to benefit even more from the MOF-based approach for ontologies. How F-Logic ontology mappings would be represented would first have to be researched as there is no established approach for this yet.

Extended support for concrete OWL ontology mappings We considered an abstract metamodel that was designed to cover a range of existing formalisms for specifying mappings, however, the modular approach allows to extend the metamodel for additional constructs or characteristics, or for additional formalisms in a straight-forward manner. In a first step, we linked the abstract metamodel to two concrete mapping formalisms [BGvH$^+$03, HM05]. This was done by creating specializations of the generic metamodel that correspond to individual mapping formalism, by adding restrictions to the metamodel in terms of OCL constraints that formalize the specific properties of the respective formalism. In order to be able to provide support not only for the acquisition of mappings but also for their implementation in one of the existing formalisms, two additional steps are to be taken. In a first step, a method can be developed for checking the compatibility of a given graphical model with a particular specialization of the metamodel. This is necessary for being able to determine whether a given model can be implemented with a particular formalism. As specializations are entirely described using OCL constraints, this can be done using an OCL model checker. In the second step, we can develop methods for translating a given graphical model into an appropriate mapping formalism. This task can be seen as a special case of code generation where instead of executable code, we generate a formal mapping model that can be operationalized using a suitable inference engine. In contrast to many existing proposals, this approach takes a knowledge-level perspective on mapping modeling and supports an iterative development process where the mapping model is refined in a stepwise manner and the decision for a specific implementation formalism is only taken later in the process.

Our contributions provide a promising first step for the issues we just shortly addressed. In the future, our work can be continued successfully in these directions.

A. Appendix

A.1. Detailed Overview of the OWL Metamodel

We describe the different metaclasses in the OWL metamodel in alphabetical order, on top of the explanations of Chapter 4 starting on page 37. As in the UML specifications, we present each metaclass according to the following specification conventions:

Description
An informal description of the metaclass.

Attributes, Associations and Generalizations
All attributes of the metaclass, ends of associations that start from the considered class, as well as generalizations of the class are listed together with a short comment.

Constraints
Constraints are expressed in OCL. They define invariants for the metaclass that must be fulfilled by all instances of that metaclass. The context of a constraint is always the considered metaclass.

1 - Annotation

Description
An annotation to provide certain remarks to ontologies and their elements. Next to a more general annotation, two specific types of annotations exist: label and comment.

Attributes, Associations and Generalizations

- URI: String[0..1] defines the URI of the annotation in case of a self-defined annotation.
- annotationValue: Constant[1] defines the mandatory value of the annotation.

Constraints

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

2 - AntisymmetricObjectProperty
Description
An axiom that states that for a certain object property, it holds that when the pair (x,y) belongs to its extension, then the pair (y,x) does not belong to its extension[1].

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property that is defined to be antisymmetric.
- Specializes class ObjectPropertyAxiom.

Constraints
No constraints.

3 - ClassAssertion
Description
A fact asserting an individual to a class.

Attributes, Associations and Generalizations

- individual: Individual[1] specifies the individual that is asserted to a class.
- class: Description[1] specifies the class to which the individual is asserted.
- Specializes class Fact.

Constraints
No constraints.

4 - ClassAxiom
Description
An abstract supertype for all types of class axioms.

Attributes, Associations and Generalizations

- Specializes class OWLAxiom.

Constraints
No constraints.

5 - Comment

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

Description
A specific type of annotation.

Attributes, Associations and Generalizations

- Specializes class Annotation.

Constraints

1. A comment does not have a URI:
   self.URI = 0

6 - Constant

Description
A constant which can be typed or untyped. When typed, a constant consists of a string and a datatype URI; when untyped, a constant has a string and a language tag.

Attributes, Associations and Generalizations

- languageTag: String[0..1] defines the language tag of an untyped constant.
- value: String[1] defines the string of a constant.
- URI: String[0..1] defines the datatype URI in case of a typed constant.

Constraints

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

7 - DataAllValuesFrom

Description
A class description defining a class of the individuals which have only values of the specified data range as values for the specified data property.

Attributes, Associations and Generalizations

- property: DataPropertyExpression[1..*] specifies the data property on which the restriction is defined. OWL allows more data properties to be specified in the restriction.
- range: DataRange[1] specifies the restricting data range.

- Specializes class Description.

Constraints
No constraints.

8 - DataComplementOf
Description
A data range defined as the complement of another data range.

Attributes, Associations and Generalizations

- dataRange: DataRange[1] links the data range to its complement.
- Specializes class DataRange.

Constraints
No constraints.

9 - DataExactCardinality
Description
A class description defining a class of the individuals which have exactly the given number of data values of the specified data range as values for the specified data property. Obviously the cardinality may not be negative.

Attributes, Associations and Generalizations

- property: DataPropertyExpression[1] specifies the data property on which the restriction is defined.
- range: DataRange[0..1] specifies the restricting data range.
- cardinality: Integer[1] specifies the cardinality.
- Specializes class Description.

Constraints

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

10 - DataHasValue
Description
A class description defining a class based on the existence of particular values for a data property.

Attributes, Associations and Generalizations

- property: DataPropertyExpression[1] specifies the data property on which the restriction is defined.
- value: Constant[1] specifies the restricting constant.
- Specializes class Description.

Constraints
No constraints.

11 - DataMaxCardinality

Description
A class description defining a class of the individuals which have at most the given number of data values of the specified data range as values for the specified data property. Obviously the cardinality may not be negative.

Attributes, Associations and Generalizations

- property: DataPropertyExpression[1] specifies the data property on which the restriction is defined.
- range: DataRange[0..1] specifies the restricting data range.
- cardinality: Integer[1] specifies the cardinality.
- Specializes class Description.

Constraints

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

12 - DataMinCardinality

Description
A class description defining a class of the individuals which have at least the given number of data values of the specified data range as values for the specified data property. Obviously the cardinality may not be negative.

Attributes, Associations and Generalizations

- property: DataPropertyExpression[1] specifies the data property on which the restriction is defined.
- range: DataRange[0..1] specifies the restricting data range.

- cardinality: Integer[1] specifies the cardinality.
- Specializes class Description.

Constraints

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

13 - DataOneOf

Description

A data range which is defined through an enumeration of data values. Different types of data values are allowed.

Attributes, Associations and Generalizations

- constants: Constant[1..*] defines the data values contained in the data range.
- Specializes class DataRange.

Constraints

No constraints.

14 - DataProperty

Description

A data property, which is a property holding between an individual and a data value.

Attributes, Associations and Generalizations

- Specializes class OWLEntity.
- Specializes class DataPropertyExpression.

Constraints

No constraints.

15 - DataPropertyAssertion

Description

A fact stating that a certain constant is the value of a specified individual through a specified data property.

Attributes, Associations and Generalizations

- source: Individual[1] specifies the subject individual.

- target: Constant[1] specifies the value constant.
- property: DataPropertyExpression[1] specifies the data property that is instantiated.
- Specializes class Fact.

Constraints
No constraints.

16 - DataPropertyAxiom
Description
An abstact supertype of all types of data property axioms.

Attributes, Associations and Generalizations

- Specializes class OWLAxiom.

Constraints
No constraints.

17 - DataPropertyDomain
Description
An axiom defining the domain of a data property.

Attributes, Associations and Generalizations

- property: DataPropertyExpression[1] specifies the data property of which the domain is defined.
- domain: Description[1] specifies the description that is defined to be the domain of the specified data property.
- Specializes class DataPropertyAxiom.

Constraints
No constraints.

18 - DataPropertyExpression
Description
An abstract supertype of a data property.

Attributes, Associations and Generalizations
No attributes, associations or generalizations.

Constraints
No constraints.

19 - DataPropertyRange
Description
An axiom defining the range of a data property.

Attributes, Associations and Generalizations

- property: DataPropertyExpression[1] specifies the data property of which the range is defined.
- range: DataRange[1] specifies the data range that is defined to be the range of the specified data property.
- Specializes class DataPropertyAxiom.

Constraints
No constraints.

20 - DataRange
Description
An abstract supertype of the different types of data ranges. This concerns datatype, data range complement, data range restriction and datavalue enumeration.

Attributes, Associations and Generalizations
No attributes, associations or generalizations.

Constraints
No constraints.

21 - DataSomeValuesFrom
Description
A class description defining a class of the individuals which have at least some values of the specified data range as values for the specified data property.

Attributes, Associations and Generalizations

- properties: DataPropertyExpression[1..*] specifies the data property on which the restriction is defined. OWL allows more data properties to be specified in the restriction.
- range: DataRange[1] specifies the restricting data range.
- Specializes class Description.

Constraints
No constraints.

22 - Datatype

Description
A simple datatype. A set of primitive datatypes is predefined, of wich *rdfs:Literal* is in an exceptional position. Similar to *owl:Thing* which contains all individuals, *rdfs:Literal* contains all data values.

Attributes, Associations and Generalizations

- Specializes class OWLEntity.
- Specializes class DataRange.

Constraints
No constraints.

23 - DatatypeRestriction

Description
A data range which is specified by defining a facet with a certain value on an existing data range.

Attributes, Associations and Generalizations

- datatypeFacet: String[1] defines the facet of the restriction.
- restrictionValue: Constant[1] defines the facet's value.
- dataRange: DataRange[1] specifies the data range on which the facet is applied.
- Specializes class DataRange.

Constraints

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

24 - Declaration

Description

A declaration of an entity.

Attributes, Associations and Generalizations

- entity: OWLEntity[1] specifies the entity which is declared.
- Specializes class OWLAxiom.

Constraints

No constraints.

25 - Description

Description

An abstract supertype of all simple and complex class constructs. A complex class description specifies the extension of an anonymous class.

Attributes, Associations and Generalizations

No attributes, associations or generalizations.

Constraints

No constraints.

26 - DifferentIndividuals

Description

A fact stating that several individuals are different.

Attributes, Associations and Generalizations

- differentIndividuals: Individual[2..*] specifies the individuals that are defined to be different. At least two individuals must be specified.
- Specializes class Fact.

Constraints

No constraints.

27 - DisjointClasses

Description
A class axiom defining that several classes are pair-wise disjoint.

Attributes, Associations and Generalizations

- disjointClasses: Description[2..*] specifies the class descriptions that are defined to be disjoint.
- Specializes class ClassAxiom.

Constraints
No constraints.

28 - DisjointDataProperties

Description
An axiom stating that several data properties are disjoint.

Attributes, Associations and Generalizations

- disjointProperties: DataPropertyExpression[2..*] specifies the data properties that are defined to be disjoint. At least two data properties must be specified.
- Specializes class DataPropertyAxiom.

Constraints
No constraints.

29 - DisjointObjectProperties

Description
An axiom stating that the extensions of several object properties are disjoint.

Attributes, Associations and Generalizations

- disjointProperties: ObjectPropertyExpression[2..*] specifies the object properties that are defined to be disjoint. At least two object properties must be specified.
- Specializes class ObjectPropertyAxiom.

Constraints
No constraints.

30 - DisjointUnion

Description
A class axiom defining that one class is the union of a set of other classes, which are pair-wise

disjoint.

Attributes, Associations and Generalizations

- disjointClasses: Description[2..*] specifies the disjoint class descriptions.
- unionClass: OWLClass[1] specifies the union class.
- Specializes class ClassAxiom.

Constraints
No constraints.

31 - EntityAnnotation
Description
A type of axiom defining annotations on entities.

Attributes, Associations and Generalizations

- entity: OWLEntity[1] specifies the entity on which the annotations are defined.
- entityAnnotation: Annotation[*] specifies the annotations on the entity.
- Specializes class OWLAxiom.

Constraints
No constraints.

32 - EquivalentClasses
Description
A class axiom defining that several classes are equivalent.

Attributes, Associations and Generalizations

- equivalentClasses: Description[2..*] specifies the class descriptions that are defined to be equivalent.
- Specializes class ClassAxiom.

Constraints
No constraints.

33 - EquivalentDataProperties
Description
An axiom stating that several data properties are equivalent.

Attributes, Associations and Generalizations

- equivalentProperties: DataPropertyExpression[2..*] specifies the data properties that are defined to be equivalent. At least two data properties must be specified.
- Specializes class DataPropertyAxiom.

Constraints
No constraints.

34 - EquivalentObjectProperties
Description
An axiom stating that the extensions of several object properties are equal.

Attributes, Associations and Generalizations

- equivalentProperties: ObjectPropertyExpression[2..*] specifies the object properties that are defined to be equivalent. At least two object properties must be specified.
- Specializes class ObjectPropertyAxiom.

Constraints
No constraints.

35 - Fact
Description
An abstract supertype for all types of facts.

Attributes, Associations and Generalizations

- Specializes class OWLAxiom.

Constraints
No constraints.

36 - FunctionalDataProperty
Description
An axiom that states that a certain data property is functional.

Attributes, Associations and Generalizations

- property: DataPropertyExpression[1] specifies the data property that is defined to be functional.
- Specializes class DataPropertyAxiom.

Constraints
No constraints.

37 - FunctionalObjectProperty

Description
An axiom that states that a certain object property is functional.

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property that is defined to be functional.
- Specializes class ObjectPropertyAxiom.

Constraints
No constraints.

38 - Individual

Description
An instance of a class.

Attributes, Associations and Generalizations

- Specializes class OWLEntity.

Constraints
No constraints.

39 - InverseFunctionalObjectProperty

Description
An axiom that states that a certain object property is inverse functional.

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property that is defined to be inverse functional.
- Specializes class ObjectPropertyAxiom.

Constraints
No constraints.

40 - InverseObjectProperties

Description

An axiom stating that the extensions of two object properties are the inverse of each other.

Attributes, Associations and Generalizations

- inverseProperties: ObjectPropertyExpression[2] specifies the two object properties that are defined to be inverse.
- Specializes class ObjectPropertyAxiom.

Constraints
No constraints.

41 - InverseObjectProperty
Description
An object property which is defined as the inverse of another object property.

Attributes, Associations and Generalizations

- inverseProperty: ObjectPropertyExpression[1] specifies the inverse object property of the newly defined object property.
- Specializes class ObjectPropertyExpression.

Constraints
No constraints.

42 - IrreflexiveObjectProperty
Description
An axiom that states that a certain object property is irreflexive.

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property that is defined to be irreflexive.
- Specializes class ObjectPropertyAxiom.

Constraints
No constraints.

43 - Label
Description
A specific type of annotation.

Attributes, Associations and Generalizations

- Specializes class Annotation.

Constraints

1. A label does not have a URI:
   self.URI = 0

44 - NegativeDataPropertyAssertion

Description

A fact stating that a certain constant is not the value of a specified individual through a specified data property.

Attributes, Associations and Generalizations

- source: Individual[1] specifies the subject individual.
- target: Constant[1] specifies the value constant.
- property: DataPropertyExpression[1] specifies the data property that is instantiated.
- Specializes class Fact.

Constraints

No constraints.

45 - NegativeObjectPropertyAssertion

Description

A fact stating that a certain individual is not the value of another specified individual through a specified object property.

Attributes, Associations and Generalizations

- source: Individual[1] specifies the subject individual.
- target: Individual[1] specifies the value individual.
- property: ObjectPropertyExpression[1] specifies the object property that is instantiated.
- Specializes class Fact.

Constraints

No constraints.

46 - ObjectAllValuesFrom

Description

A class description defining a class of the individuals which have only instances of the specified class description as values for the specified object property.

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property on which the restriction is defined.
- class: Description[1] specifies the restricting class.
- Specializes class Description.

Constraints

No constraints.

47 - ObjectComplementOf

Description

A class description defining a class as the complement of another class.

Attributes, Associations and Generalizations

- class: Description[1] specifies the complementary class description.
- Specializes class Description.

Constraints

No constraints.

48 - ObjectExactCardinality

Description

A class description defining a class of the individuals which have exactly the given number of individuals of the specified class as values for the specified object property. Obviously the cardinality may not be negative.

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property on which the restriction is defined.
- class: Description[0..1] specifies the restricting class. When no class description is specified, the property restriction is unqualified and the restricting class is *owl:Thing*.
- cardinality: Integer[1] specifies the cardinality.

- Specializes class Description.

Constraints

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

49 - ObjectExistsSelf

Description

A class description defining a class of the individuals that have themselves as values for the specified object property.

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property on which the restriction is defined.
- Specializes class Description.

Constraints

No constraints.

50 - ObjectHasValue

Description

A class description defining a class based on the existence of particular values for an object property.

Attributes, Associations and Generalizations

- value: Individual[1] specifies the particular value that instance of the class can have for the specified property.
- property: ObjectPropertyExpression[1] specifies the object property on which the restriction is defined.
- Specializes class Description.

Constraints

No constraints.

51 - ObjectIntersectionOf

Description

A class description defining a class as the intersection of some other classes.

Attributes, Associations and Generalizations

- classes: Description[2..*] specifies the class descriptions in the intersection.
- Specializes class Description.

Constraints
No constraints.

52 - ObjectMaxCardinality

Description
A class description defining a class of the individuals which have at most the given number of individuals of the specified class as values for the specified object property. Obviously the cardinality may not be negative.

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property on which the restriction is defined.
- class: Description[0..1] specifies the restricting class. When no class description is specified, the property restriction is unqualified and the restricting class is *owl:Thing*.
- cardinality: Integer[1] specifies the cardinality.
- Specializes class Description.

Constraints

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

53 - ObjectMinCardinality

Description
A class description defining a class of the individuals which have at least the given number of individuals of the specified class as values for the specified object property. Obviously the cardinality may not be negative.

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property on which the restriction is defined.
- class: Description[0..1] optionally specifies the restricting class. When no class description is specified, the property restriction is unqualified and the restricting class is *owl:Thing*.

- cardinality: Integer[1] specifies the cardinality.
- Specializes class Description.

Constraints

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

54 - ObjectOneOf

Description
A class description defining a class by an enumerating its associated individuals.

Attributes, Associations and Generalizations

- individuals: Individual[1..*] specifies the individuals in the enumeration.
- Specializes class Description.

Constraints
No constraints.

55 - ObjectProperty

Description
An object property, which is a property holding between individuals.

Attributes, Associations and Generalizations

- Specializes class OWLEntity.
- Specializes class ObjectPropertyExpression.

Constraints
No constraints.

56 - ObjectPropertyAssertion

Description
A fact stating that a certain individual is the value of another specified individual through a specified object property.

Attributes, Associations and Generalizations

- source: Individual[1] specifies the subject individual.

- target: Individual[1] specifies the value individual.
- property: ObjectPropertyExpression[1] specifies the object property that is instantiated.
- Specializes class Fact.

Constraints
No constraints.

57 - ObjectPropertyAxiom
Description
An abstract supertype of all types of object property axioms.

Attributes, Associations and Generalizations

- Specializes class OWLAxiom.

Constraints
No constraints.

58 - ObjectPropertyDomain
Description
An axiom defining the domain of an object property.

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property of which the domain is specified.
- domain: Description[1] specifies the description that is specified as the domain of the object property.
- Specializes class ObjectPropertyAxiom.

Constraints
No constraints.

59 - ObjectPropertyExpression
Description
An abstract supertype of the two object property types, normal object properties and inverse object properties.

Attributes, Associations and Generalizations

No attributes, associations or generalizations.

Constraints
No constraints.

60 - ObjectPropertyRange

Description
An axiom defining the range of an object property.

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property of which the range is specified.
- range: Description[1] specifies the description that is specified as the range of the object property.
- Specializes class ObjectPropertyAxiom.

Constraints
No constraints.

61 - ObjectSomeValuesFrom

Description
A class description defining a class of the individuals which have at least some instances of the specified class description as values for the specified object property.

Attributes, Associations and Generalizations

- class: Description[1] specifies the restricting class.
- property: ObjectPropertyExpression[1] specifies the object property on which the restriction is defined.
- Specializes class Description.

Constraints
No constraints.

62 - ObjectUnionOf

Description
A class description defining a class as the union of some other classes.

Attributes, Associations and Generalizations

- classes: Description[2..*] specifies the class descriptions in the union.
- Specializes class Description.

Constraints
No constraints.

63 - Ontology
Description
An ontology consisting of axioms, eventually annotated and importing other ontologies.

Attributes, Associations and Generalizations

- URI: String[1] defines the URI of an ontology. A URI is mandatory for every ontology.
- importedOntology: Ontology[*] links the ontology to other ontologies which it imports. Importing another ontology is optional, and the possible number of imported ontologies is unlimited.
- ontologyAnnotation: Annotation[*] links the ontology to the annotations that are defined on it. Annotations on ontologies are optional.
- ontologyAxiom: OWLAxiom[*] links the ontology to its axioms.

Constraints
No constraints.

64 - OWLAxiom
Description
An abstract supertype of all types of ontology axioms. This concerns class axioms, object property axioms, data property axioms, declarations, facts and entity annotations.

Attributes, Associations and Generalizations

- axiomAnnotation: Annotation[*] links an axiom to its annotations. Annotations are not mandatory for axioms.

Constraints
No constraints.

65 - OWLClass
Description
An atomic class. Two atomic classes are predefined: the universal class *owl:Thing* containing all individuals, and the empty class *owl:Nothing* containing nothing.

Attributes, Associations and Generalizations

- Specializes class OWLEntity.
- Specializes class Description.

Constraints
No constraints.

66 - OWLEntity
Description
An abstract supertype of all types of OWL entities. This concerns data type, OWL class, object property, data property and indivdual.

Attributes, Associations and Generalizations

- URI: String[1] defines the URI of the entity.

Constraints
No constraints.

67 - ReflexiveObjectProperty
Description
An axiom that states that a certain object property is reflexive.

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property that is defined to be reflexive.
- Specializes class ObjectPropertyAxiom.

Constraints
No constraints.

68 - SameIndividual
Description
A fact stating that several individuals are the same.

Attributes, Associations and Generalizations

- sameIndividuals: Individual[2..*] specifies the individuals that are defined to be the same. At least two individuals must be specified.

- Specializes class Fact.

Constraints
No constraints.

69 - SubClassOf
Description
A class axiom defining that one class is a subclass of the other.

Attributes, Associations and Generalizations

- subClass: Description[1] specifies the subclass.
- superClass: Description[1] specifies the superclass.
- Specializes class ClassAxiom.

Constraints
No constraints.

70 - SubDataPropertyOf
Description
An axiom stating that the extension of one data property is a subset of the extension of another data property.

Attributes, Associations and Generalizations

- subProperty: DataPropertyExpression[1] specifies the subproperty.
- superProperty: DataPropertyExpression[1] specifies the superproperty.
- Specializes class DataPropertyAxiom.

Constraints
No constraints.

71 - SubObjectPropertyOf
Description
An axiom stating that the extension of one or several object properties are subsets of the extension of a certain other object property.

Attributes, Associations and Generalizations

- subProperties: ObjectPropertyExpression[1..*] specifying the subproperties. When more subproperties are defined, they are ordered.

- superProperty: ObjectPropertyExpression[1] specifying the superproperty.
- Specializes class ObjectPropertyAxiom.

Constraints
No constraints.

72 - SymmetricObjectProperty

Description
An axiom that states that a certain object property is symmetric.

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property that is defined to be symmetric.
- Specializes class ObjectPropertyAxiom.

Constraints
No constraints.

73 - TransitiveObjectProperty

Description
An axiom that states that a certain object property is transitive.

Attributes, Associations and Generalizations

- property: ObjectPropertyExpression[1] specifies the object property that is defined to be transitive.
- Specializes class ObjectPropertyAxiom.

Constraints
No constraints.

A.2. Mappings between OWL and the OWL Metamodel

The table below defines the mapping function ρ from OWL to the MOF-based OWL metamodel which we defined in Chapter 4 starting on page 37. For each OWL element or partial element presented in the functional-style syntax, the corresponding metamodel element is indicated and its attributes and associations specified. In doing so, we partially fall back on the OCL syntax. Attributes are indented, below the element to which they belong. The inverse mapping from the metamodel to OWL can be derived directly from ρ. The following notations are used in the definition of the mapping function:

- a represents an annotation URI;
- d represents a datatype URI;
- c, c_1, ..., c_n represent OWL class URIs;
- o, o_1, ..., o_n represent object property URIs;
- r, r_1, ..., r_n represent data property URIs;
- i, i_1, ..., i_n represent individual URIs;
- t_1, t_2, ..., t_n represent ontology URIs;
- s, s_1, ..., s_n represent strings;
- x_1, ..., x_n represent axioms (standing for one of the specific types of axioms in OWL);
- n represents a nonnegative integer;
- l represents a language tag;
- f represents a datatype facet;
- g, g_1, ..., g_n represent constants.

When a partial construct can be of different types and the mapping function values of these types are defined elsewhere in the table, we take the simplest of them for the sake of clarity in introducing the main construct. Additionally, we do not introduce annotations within the different constructs but specify them separately[2]. Thus whenever an annotation is used in a construct, the function value for the annotation can be added to the function value of the construct.

[2]Note that annotations are always optional.

OWL	**Metamodel = ρ(OWL)**
Ontologies	
Ontology(t_1 [Import(t_2) ... Import(t_n)] [x_1 ... x_n])	Ontology [ontologyAxiom = Set{$\rho(x_1)$, ..., $\rho(x_n)$}] [importedOntology = Set{$\rho(t_2)$, ..., $\rho(t_n)$}] URI = t_1
Annotations	

Annotation(a s)	Annotation
	annotationValue = $\rho(s)$
	URI = a

Label(s)	Label annotationValue = $\rho(s)$
Comment(s)	Comment annotationValue = $\rho(s)$
Annotation(a s) (on an axiom)	(axiom) axiomAnnotation = ρ(Annotation(a s))
Annotation(a s) (on an ontology)	(ontology) ontologyAnnotation = ρ(Annotation(a s))

EntityAnnotation(c Annotation(a s))	EntityAnnotation
	entity = $\rho(c)$
	entityAnnotation = ρ(Annotation(a s))

Entities (in entity declarations)	
Datatype(d)	Datatype URI = d
OWLClass(c)	OWLClass URI = c
ObjectProperty(o)	ObjectProperty URI = o
DataProperty(r)	DataProperty URI = r
Individual(i)	Individual URI = i
InverseObjectProperty(o)	InverseObjectProperty inverseProperty = ρ(ObjectProperty(o))

OWL	**Metamodel = ρ(OWL)**
Entity declarations	
Declaration(Datatype(d))	Declaration entity = $\rho(d)$
Declaration(OWLClass(c))	Declaration entity = $\rho(c)$
Declaration(ObjectProperty(o))	Declaration entity = $\rho(o)$
Declaration(DataProperty(r))	Declaration entity = $\rho(r)$
Declaration(Individual(i))	Declaration entity = $\rho(i)$
Constants	

s [@ l]	Constant
	languageTag = l
	value = s

s^^d	Constant
	value = s
	URI = d

URIs	
d	Datatype URI = d
c	OWLClass URI = c
o	ObjectProperty URI = o
r	DataProperty URI = r
i	Individual URI = i
Data ranges	
DataComplementOf(d)	DataComplementOf dataRange = $\rho(d)$
DataOneOf(g_1 ... g_n)	DataOneOf constants = Set{$\rho(g_1)$, ..., $\rho(g_n)$}

DatatypeRestriction(d f g)	DatatypeRestriction
	datatypeFacet = f
	restrictionValue = $\rho(g)$
	dataRange = $\rho(d)$

OWL	**Metamodel = ρ(OWL)**
Class descriptions	
ObjectUnionOf(c_1 c_2 ... c_n)	ObjectUnionOf classes = Set{$\rho(c_1)$, $\rho(c_2)$, ..., $\rho(c_n)$}
ObjectIntersectionOf(c_1 c_2 ... c_n)	ObjectIntersectionOf classes = Set{$\rho(c_1)$, $\rho(c_2)$, ..., $\rho(c_n)$}
ObjectComplementOf(c_1)	ObjectComplementOf class = $\rho(c_1)$
ObjectOneOf(i_1 ... i_n)	ObjectOneOf individuals = Set{$\rho(i_1)$, ..., $\rho(i_n)$}

ObjectAllValuesFrom(o c)	ObjectAllValuesFrom
	class =$\rho(c)$
	property = $\rho(o)$

ObjectSomeValuesFrom(o c)	ObjectSomeValuesFrom
	class=$\rho(c)$
	property = $\rho(o)$

ObjectExistsSelf(o)	ObjectExistsSelf property = $\rho(o)$

ObjectHasValue(o i)	ObjectHasValue
	value=$\rho(i)$
	property = $\rho(o)$

ObjectMinCardinality(n o [c])	ObjectMinCardinality
	cardinality = n
	property = $\rho(o)$
	[class = $\rho(c)$]

ObjectMaxCardinality(n o [c])	ObjectMaxCardinality
	cardinality = n
	property = $\rho(o)$
	[class = $\rho(c)$]

ObjectExactCardinality(n o [c])	ObjectExactCardinality
	cardinality = n
	property = $\rho(o)$
	[class = $\rho(c)$]

DataAllValuesFrom(r_1 ... r_n d)	DataAllValuesFrom
	properties = OrderedSet{$\rho(r_1)$, ..., $\rho(r_n)$}
	range = $\rho(d)$

DataSomeValuesFrom(r_1 ... r_n d)	DataSomeValuesFrom
	properties = OrderedSet{$\rho(r_1)$, ..., $\rho(r_n)$}
	range = $\rho(d)$

OWL	**Metamodel = ρ(OWL)**

DataHasValue(r g)	DataHasValue
	property = $\rho(r)$
	value = $\rho(g)$

DataMinCardinality(n r [d])	DataMinCardinality
	cardinality = n
	property = $\rho(r)$
	[range = $\rho(d)$]

DataMaxCardinality(n r [d])	DataMaxCardinality
	cardinality = n
	property = $\rho(r)$
	[range = $\rho(d)$]

DataExactCardinality(n r [d])	DataExactCardinality
	cardinality = n
	property = $\rho(r)$
	[range = $\rho(d)$]

Class axioms	

SubClassOf(c_1 c_2)	SubClassOf
	subClass = $\rho(c_1)$
	superClass = $\rho(c_2)$

EquivalentClasses(c_1 c_2 ... c_n)	EquivalentClasses equivalentClasses = Set{$\rho(c_1)$, $\rho(c_2)$, ..., $\rho(c_n)$}
DisjointClasses(c_1 c_2 ... c_n)	DisjointClasses disjointClasses = Set{$\rho(c_1)$, $\rho(c_2)$, ..., $\rho(c_n)$}

DisjointUnion(c_1 c_2 c_3 ... c_n)	DisjointUnion
	unionClass = $\rho(c_1)$
	disjointClasses = Set{$\rho(c_2)$, $\rho(c_3)$, ..., $\rho(c_n)$}

Object property axioms	

SubObjectPropertyOf(o_1 o_2)	SubObjectPropertyOf
	subProperties = $\rho(o_1)$
	superProperty = $\rho(o_2)$

SubObjectPropertyOf(
SubObjectPropertyChain(
o_1 o_2 ... o_n) o_{n+1})
	SubObjectPropertyOf
	subProperties = OrderedSet{$\rho(o_1)$, $\rho(o_2)$, ..., $\rho(o_n)$}
	superProperty = $\rho(o_{n+1})$

EquivalentObjectProperties(o_1 o_2 ... o_n)	EquivalentObjectProperties equivalentProperties = Set{$\rho(o_1)$, $\rho(o_2)$, ..., $\rho(o_n)$}

OWL	**Metamodel = ρ(OWL)**
DisjointObjectProperties(o_1 o_2 ... o_n)	DisjointObjectProperties disjointProperties = Set{$\rho(o_1)$, $\rho(o_2)$, ..., $\rho(o_n)$}
InverseObjectProperties(o_1 o_2)	InverseObjectProperties inverseProperties = Set{$\rho(o_1)$, $\rho(o_2)$}

ObjectPropertyDomain(o c)	ObjectPropertyDomain
	property = $\rho(o)$
	domain = $\rho(c)$

ObjectPropertyRange(o c)	ObjectPropertyRange
	property = $\rho(o)$
	range = $\rho(c)$

FunctionalObjectProperty(o)	FunctionalObjectProperty property = $\rho(o)$
InverseFunctionalObjectProperty(o)	InverseFunctionalObjectProperty property = $\rho(o)$
ReflexiveObjectProperty(o)	ReflexiveObjectProperty property = $\rho(o)$
IrreflexiveObjectProperty(o)	IrreflexiveObjectProperty property = $\rho(o)$
SymmetricObjectProperty(o)	SymmetricObjectProperty property = $\rho(o)$
AntisymmetricObjectProperty(o)	AntisymmetricObjectProperty property = $\rho(o)$
TransitiveObjectProperty(o)	TransitiveObjectProperty property = $\rho(o)$
Data property axioms	

SubDataPropertyOf(r_1 r_2)	SubDataPropertyOf
	subProperty = $\rho(r_1)$
	superProperty = $\rho(r_2)$

EquivalentDataProperties(r_1 r_2 ... r_n)	EquivalentDataProperties equivalentProperties = Set{ $\rho(r_1)$, $\rho(r_2)$, ..., $\rho(r_n)$}
DisjointDataProperties(r_1 r_2 ... r_n)	DisjointDataProperties disjointProperties = Set{$\rho(r_1)$, $\rho(r_2)$, ..., $\rho(r_n)$}

DataPropertyDomain(r c)	DataPropertyDomain
	property = $\rho(r)$
	domain = $\rho(c)$

OWL	**Metamodel = ρ(OWL)**

DataPropertyRange(r d)	DataPropertyRange
	property = $\rho(r)$
	range = $\rho(d)$

FunctionalDataProperty(r)	FunctionalDataProperty property = $\rho(r)$
Facts	
SameIndividual(i_1 i_2 ... i_n)	SameIndividual sameIndividuals = Set{ $\rho(i_1)$, $\rho(i_2)$, ..., $\rho(i_n)$ }
DifferentIndividuals(i_1 i_2 ... i_n)	DifferentIndividuals differentIndividuals = Set{ $\rho(i_1)$, $\rho(i_2)$, ..., $\rho(i_n)$ }

ClassAssertion(i c)	ClassAssertion
	individual = $\rho(i)$
	class = $\rho(c)$

ObjectPropertyAssertion(o i_1 i_2)	ObjectPropertyAssertion
	property = $\rho(o)$
	source = $\rho(i_1)$
	target = $\rho(i_2)$

NegativeObjectPropertyAssertion(o i_1 i_2)	NegativeObjectPropertyAssertion
	property = $\rho(o)$
	source = $\rho(i_1)$
	target = $\rho(i_2)$

DataPropertyAssertion(r i g)	DataPropertyAssertion
	property = $\rho(r)$
	source = $\rho(i)$
	target = $\rho(g)$

NegativeDataPropertyAssertion(r i g)	NegativeDataPropertyAssertion
	property = $\rho(r)$
	source = $\rho(i)$
	target = $\rho(g)$

A.3. Detailed Overview of the SWRL Metamodel Extension

On top of the introduction of the MOF metamodel for SWRL in Chapter 5 starting on page 59, this appendix presents the descriptions of the different metaclasses of the metamodel arranged in alphabetical order. As in the former appendix, we present them as in the UML specifications, according to the following specification conventions:

Description
An informal description of the metaclass.

Attributes, Associations and Generalizations
All attributes of the metaclass, ends of associations that start from the considered class, as well as generalizations of the class are listed together with a short comment.

Constraints
Constraints are expressed in OCL. They define invariants for the metaclass that must be fulfilled by all instances of that metaclass. The context of a constraint is always the considered metaclass.

1 - Antecedent

Description
The body of a rule. As a SWRL antecedent is interpreted as a conjunction of its atoms, a non-empty antecedent holds if and only if all its atoms hold.

Attributes, Associations and Generalizations

- bodyAtoms: Atom[*] links the antecedent to its atoms. An antecedent can be empty.

Constraints
No constraints.

2 - Atom

Description
An atom in the body or head of a rule. An atom has a predicate symbol and some terms.

Attributes, Associations and Generalizations

- atomName: PredicateSymbol[1] specifies the predicate symbol.
- atomArguments: Term[*] specifies the terms in a specific order.

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:
   self.atomName.oclIsTypeOf(Description) implies
   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:
   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:
   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:
   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:
   self.atomName.oclIsTypeOf(BuiltIn) implies
   self.atomArguments→forAll(oclIsTypeOf(Constant) or oclIsTypeOf(DataVariable))

3 - BuiltIn

Description

A SWRL built-in predicate. Built-ins are classified into seven modules, like built-ins for lists or built-ins for comparisons.

Attributes, Associations and Generalizations

- URI: String[1] identifies a built-in.
- Specializes class PredicateSymbol.

Constraints
No constraints.

4 - Consequent

Description
The head of a rule. As a SWRL consequent is interpreted as the conjunction of its atoms, a non-empty consequent holds if and only if all its atoms hold.

Attributes, Associations and Generalizations

- headAtoms: Atom[*] links the consequent to its atoms. A consequent can be empty.

Constraints
No constraints.

5 - Constant (augmented definition of the metaclass in the OWL metamodel)

Attributes, Associations and Generalizations

- Specializes class Term.

6 - DataProperty (augmented definition of the metaclass in the OWL metamodel)

Attributes, Associations and Generalizations

- Specializes class PredicateSymbol.

7 - DataRange (augmented definition of the metaclass in the OWL metamodel)

Attributes, Associations and Generalizations

- Specializes class PredicateSymbol.

8 - DataVariable

Description
A data variable, which is a variable for a data value.

Attributes, Associations and Generalizations

- Specializes class Variable.

Constraints
No constraints.

9 - Description (augmented definition of the metaclass in the OWL metamodel)

Attributes, Associations and Generalizations

- Specializes class PredicateSymbol.

10 - Individual (augmented definition of the metaclass in the OWL metamodel)

Attributes, Associations and Generalizations

- Specializes class Term.

11 - IndividualVariable

Description
An individual variable, which is a variable for an individual.

Attributes, Associations and Generalizations

- Specializes class Variable.

Constraints
No constraints.

12 - ObjectProperty (augmented definition of the metaclass in the OWL metamodel)

Attributes, Associations and Generalizations

- Specializes class PredicateSymbol.

13 - PredicateSymbol

Description
A predicate symbol of an atom in a rule.

Attributes, Associations and Generalizations

No attributes, associations or generalizations.

Constraints
No constraints.

14 - Rule
Description
A rule. Informally, the meaning of a rule can be described as: *when* the antecedent holds, *then* so does the consequent. An empty antecedent is treated as trivially true, whereas an empty consequent is treated as trivially false.

Attributes, Associations and Generalizations

- URI: String[0..1] specifies the URI to identify the rule. A URI is not mandatory for a SWRL rule.
- ruleBody: Antecedent[1] links a rule to its antecedent.
- ruleHead: Consequent[1] links a rule to its consequent.
- Specializes class OWLAxiom.

Constraints

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

15 - Term
Description
An abstract supertype for all types of terms in a rule atom. This concerns data variables, individual variables, constants and individuals. As the OWL specification uses constants to represent data values, so does the metamodel for the sake of uniformity.

Attributes, Associations and Generalizations

No attributes, associations or generalizations.

Constraints
No constraints.

16 - Variable
Description

An abstract supertype for variable terms. This concerns data variables and individual variables. Variables in SWRL rules are treated as universally quantified, with their scope limited to a given rule.

Attributes, Associations and Generalizations

- name: String[1] specifies the name of the variable.
- Specializes class Term.

Constraints
No constraints.

A.4. Mappings between SWRL and the SWRL Metamodel

As a consistent extension of Appendix A.2 starting on page 189, the table below defines the values of the mapping function ρ for SWRL elements. The function ρ maps the SWRL elements to elements of the metamodel. Each SWRL element is presented in the abstract syntax and is followed by its corresponding metamodel element, its attributes and associations. In doing so, we partially fall back on the OCL syntax. Attributes are indented, below the element to which they belong. From the function ρ, the inverse mapping from the metamodel to SWRL can be derived directly. The following notations are used in the definition of the mapping function:

- d represents a datatype URI;
- u represents a random URI;
- b represents a built-in URI;
- c represents an OWL class URI;
- o represents an object property URI;
- r represents a data property URI;
- e, e_1, ..., e_n represent terms (standing for one of the specific types of terms in SWRL);
- m_1, ..., m_t represent rule atoms (standing for one of the specific types of atoms in SWRL).

When a partial construct can be of different types and the mapping function values of these types are defined elsewhere by ρ, we take the simplest of them for the sake of clarity in introducing the main construct. Moreover, we do no present an annotation in the rule construct for the sake of clarity as the function value for the annotation can be taken from Appendix A.2 and inserted in the function value of a rule.

SWRL	**Metamodel = ρ(SWRL)**
Rules	

Implies([u]
Antecedent(m_1 ... m_n)
Consequent())
	Rule
	[URI = u]
	ruleBody = Antecedent
	bodyAtoms = Set{$\rho(m_1)$, ..., $\rho(m_n)$}
	ruleHead = Consequent

Implies([u]
Antecedent()
Consequent(m_1 ... m_n))
	Rule
	[URI = u]
	ruleBody = Antecedent
	ruleHead = Consequent
	headAtoms = Set{$\rho(m_1)$, ..., $\rho(m_n)$}

Implies([u]
Antecedent(m_1 ... m_n)
Consequent(m_{n+1} ... m_t))
	Rule
	[URI = u]
	ruleBody = Antecedent
	bodyAtoms = Set{$\rho(m_1)$, ..., $\rho(m_n)$}
	ruleHead = Consequent
	headAtoms = Set{$\rho(m_{n+1})$, ..., $\rho(m_t)$}

Atoms	

$c(e)$	Atom
	atomName = $\rho(c)$
	atomArguments = $\rho(e)$

$d(e)$	Atom
	atomName = $\rho(d)$
	atomArguments = $\rho(e)$

$o(e_1\ e_2)$	Atom
	atomName = $\rho(o)$
	atomArguments = OrderedSet{$\rho(e_1)$, $\rho(e_2)$}

$r(e_1\ e_2)$	Atom
	atomName = $\rho(r)$
	atomArguments = OrderedSet{$\rho(e_1)$, $\rho(e_2)$}

sameAs($e_1\ e_2$)	Atom
	atomName = BuiltIn
	URI = 'sameAs'
	atomArguments = OrderedSet{$\rho(e_1)$, $\rho(e_2)$}

differentFrom($e_1\ e_2$)	Atom
	atomName = BuiltIn
	URI = 'differentFrom'
	atomArguments = OrderedSet{$\rho(e_1)$, $\rho(e_2)$}

SWRL	**Metamodel = ρ(SWRL)**

builtin(b)	Atom
	  atomName = BuiltIn
	    URI = b

builtin(b e_1 ... e_n)	Atom
	  atomName = BuiltIn
	    URI = b
	  atomArguments = OrderedSet{$\rho(e_1)$, ..., $\rho(e_n)$}

Variable-terms	
I-variable(u)	IndividualVariable name = u
D-variable(u)	DataVariable name = u

A.5. Detailed Overview of the F-Logic Metamodel

On top of the explanation of the MOF based metamodel for F-Logic in Chapter 6 starting on page 65, we present the descriptions of the different metaclasses in this appendix, in alphabetical order. In doing so, we follow the specification conventions as also used in the UML specifications:

Description
An informal description of the metaclass.

Attributes, Associations and Generalizations
All attributes of the metaclass, ends of associations that start from the considered class, as well as generalizations of the class are listed together with a short comment.

Constraints
Constraints are expressed in OCL. They define invariants for the metaclass that must be fulfilled by all instances of that metaclass. The context of a constraint is always the considered metaclass.

1 - BuiltIn

Description
A built-in feature, including several comparison predicates, the basic arithmetic operators, and so forth.

Attributes, Associations and Generalizations

- Specializes class PredicateSymbol.

Constraints
No constraints.

2 - Conjunction

Description
A conjunction of formulas.

Attributes, Associations and Generalizations

- Specializes class Formula.
- connectedFormulas: Formula[2..*] specifies two or more formulas of the conjunction.

Constraints

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

3 - Constant

Description

A term to name objects. A constant can be seen as a functional term with zero arguments.

Attributes, Associations and Generalizations

- Specializes class FunctionalTerm.

Constraints

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

4 - Disjunction

Description

A disjunction of formulas.

Attributes, Associations and Generalizations

- Specializes class Formula.
- connectedFormulas: Formula[2..*] specifies two or more formulas of the disjunction.

Constraints

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

5 - Equivalence

Description

An equivalence relation between two formulas.

Attributes, Associations and Generalizations

- Specializes class Formula.
- connectedFormulas: Formula[2] specifies exactly two formulas for the equivalence connective.

Constraints

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

6 - Exists

Description

An existential quantifier, which is modeled as a partial formula.

Attributes, Associations and Generalizations

- Specializes class Formula.
- containedFormula: Formula[1] specifies the formula in which the variables are declared existentially.
- boundVariables: Variable[1..*] links the formula to the variables it declares. At least one variable must be specified.

Constraints

1. 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:
   self.boundVariables→forAll(v : Variable | v.isBoundToExists=self)

7 - FAtom

Description

An F-atom which is a simplified F-molecule containing either a simple term as host and exactly one method, or a combined host object with zero methods.

Attributes, Associations and Generalizations

- Specializes class FMolecule.

Constraints

1. An F-atom is an F-molecule with a simple term as host object and exactly one method, or an InstanceOf- or SubClassOf-atom as host object with zero methods:
   (self.host.oclIsTypeOf(Term) and self.methods→size()=1) or
   (not self.host.oclIsTypeOf(Term) and self.methods→size()=0)

8 - FLogicOntology

Description

An F-Logic ontology containing facts, rules and queries.

Attributes, Associations and Generalizations

- ontologyFormulas: Formula[*] defines the facts, rules and queries which are in the ontology.

Constraints

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:
   self.ontologyFormulas→forAll(oclIsTypeOf(FMolecule) or oclIsTypeOf(Rule) or oclIsTypeOf(Query) or oclIsTypeOf(PAtom))

9 - FMolecule

Description

An F-molecule which combines several F-atoms. An F-molecule consists of a host object and one or more method applications or signatures. The metamodel defines an F-atom as a special type of an F-molecule.

Attributes, Associations and Generalizations

- Specializes class Formula.
- host: HostObject[1] specifies the first part of the F-molecule.
- methods: Method[*] specifies the set of methods of the F-molecule. It is possible to define zero methods.
- Specializes class MethodValue.

Constraints

No constraints.

10 - ForAll

Description

A universal quantifier, which is modeled as a partial formula.

Attributes, Associations and Generalizations

- Specializes class Formula.
- containedFormula: Formula[1] specifies the formula in which the variables are declared universally.
- boundVariables: Variable[1..*] links the formula to the variables it declares. At least one variable must be specified.

Constraints

1. 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:
   self.boundVariables→forAll(v : Variable | v.isBoundToForAll=self)

11 - Formula

Description

An abstract supertype for all types of formulas in F-Logic. Not all subtypes are directly contained into an ontology. Some of the supertypes are logical connectives and quantifiers, so partial formulas that are used in rules and queries.

Attributes, Associations and Generalizations

No attributes, associations or generalizations.

Constraints

No constraints.

12 - FunctionalTerm

Description

A functional term, which is a term that has other terms as arguments. A constant in F-Logic is specified as a functional term without arguments, so is represented as a subclass.

Attributes, Associations and Generalizations

- arguments: Term[*] specifies the arguments of the functional term in a specific order. A normal functional term must have at least one argument but as the subtypes do not need at least 1 argument, the multiplicity is set to '*'.
- Specializes class Term.

Constraints

1. A functional term that is not a constant, must have at least one argument:
   self→forAll(not oclIsTypeOf(Constant) implies self.argument→size()>0)

13 - HostObject

Description

An abstract type of all types of host objects of F-molecules and F-atoms.

Attributes, Associations and Generalizations

No attributes, associations or generalizations.

Constraints

No constraints.

14 - Implication

Description

An implication between two formulas.

Attributes, Associations and Generalizations

- Specializes class Formula.
- antecedent: Formula[1] specifies the first argument of the implication.
- consequent: Formula[1] specifies the second argument of the implication.

Constraints

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

15 - InstanceOf

Description

An assertion defining that one term representing an instance, belongs to the other representing a class. This statement is used as host object in F-molecules, or as F-atom.

Attributes, Associations and Generalizations

- instance: Term[1] specifies the term representing the instance in the statement.
- class: Term[1] specifies the term representing the class in the statement.
- Specializes class HostObject.

Constraints

No constraints.

16 - Method

Description

An abstract type for all types of methods in F-molecules and F-atoms. This concerns single- or multi valued method applications, and single- or multi valued method signatures.

Attributes, Associations and Generalizations

- name: Term[1] specifies the name of the method.
- parameters: Term[*] specifies optional method parameters.

Constraints
No constraints.

17 - MethodValue

Description
An abstract supertype for the different types of method values. This concerns terms and F-molecules.

Attributes, Associations and Generalizations
No attributes, associations or generalizations.

Constraints
No constraints.

18 - MultiValuedApplication

Description
A method application with multiple values.

Attributes, Associations and Generalizations

- value: MethodValue[1..*] specifies the values of the method application.
- Specializes class Method.

Constraints
No constraints.

19 - MultiValuedSignature

Description
A method definition with multiple values.

Attributes, Associations and Generalizations

- value: MethodValue[1] specifies the value of the method signature which is a class.
- Specializes class Method.

Constraints
No constraints.

20 - Negation

Description

A negation of another formula.

Attributes, Associations and Generalizations

- Specializes class Formula.
- connectedFormula: Formula[1] specifies the formula of the negation.

Constraints

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

21 - PAtom

Description

A predicate symbol followed by one or more terms.

Attributes, Associations and Generalizations

- Specializes class Formula.
- predicate: PredicateSymbol[1] specifies the predicate symbol of the P-atom.
- parameters: Term[*] specifies the ordered list of terms. A predicate symbol without term is possible. When the predicate symbol is a built-in, at least one term must be specified.

Constraints

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

22 - PredicateSymbol

Description

A predicate symbol of a P-atom.

Attributes, Associations and Generalizations

- name: String[1] specifies the name of the predicate.

Constraints
No constraints.

23 - Query

Description
A query, asked on an F-Logic program of rules and facts.

Attributes, Associations and Generalizations

- name: String[0..1] defines the name of a query. A name for a query is not mandatory in F-Logic.
- Specializes class Formula.
- containedFormula: Formula[1] defines the formula of the query.

Constraints

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

24 - Rule

Description

A rule consisting of a head and a body.

Attributes, Associations and Generalizations

- name: String[0..1] defines the name of a rule. A name for a rule is not mandatory in F-Logic.
- Specializes class Formula.
- containedFormula: Formula[1] defines the formula of the rule.

Constraints

1. A rule is an implication formula and possibly has a universal quantifier in front:
   (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). The following constraint defines this for rules without a universal quantifier:
   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)
3. A similar constraint is defined for rules with universal quantifier:
   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:
   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:
   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:
   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:
   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)

25 - SingleValuedApplication

Description

A method application with one value.

Attributes, Associations and Generalizations

- value: MethodValue[1] specifies the value of the method instance.
- Specializes class Method.

Constraints

No constraints.

26 - SingleValuedSignature

Description

A method definition with one value.

Attributes, Associations and Generalizations

- value: MethodValue[1] specifies the value of the method signature.

- Specializes class Method.

Constraints
No constraints.

27 - SubClassOf

Description
An assertion defining that one class is a subclass of another one. The subclass-statement is used as host object in F-molecules, or as F-atom.

Attributes, Associations and Generalizations

- subclass: Term[1] specifies the term representing the subclass in the statement.
- superclass: Term[1] specifies the term representing the superclass in the statement.
- Specializes class HostObject.

Constraints
No constraints.

28 - Term

Description
An abstract supertype for the different types of terms in F-Logic. This concerns variables, functional terms and constants.

Attributes, Associations and Generalizations

- name: String[1] specifies the name of the term.
- Specializes class HostObject.
- Specializes class MethodValue.

Constraints
No constraints.

29 - Variable

Description
A variable term. To distinguish between constants and variables, every variable has to be bound to a logical quantifier.

Attributes, Associations and Generalizations

- isBoundToExists: Exists[0..1] specifies the existential quantifier that declares the variable.
- isBoundToForAll: ForAll[0..1] specifies the universal quantifier that declares the variable.
- Specializes class Term.

Constraints

1. A variable must be bound to exactly one universal quantifier or existential quantifier:
   (self.isBoundToExists=1 or self.isBoundToForAll=1)
   and
   (self.isBoundToForAll=1 implies self.isBoundToForExists=0)
2. When a variable is bound to a quantifier in one direction, then this quantifier must have the variable defined as one of its bound variables in the other direction:
   (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))

A.6. Mappings between F-Logic and the F-Logic Metamodel

We define the mapping function ϕ from F-Logic to its MOF-metamodel in the following table. The table shows how ϕ maps each F-Logic element or partial element to an element of the metamodel with specific values for attributes and associations. We partly rely on OCL in presenting the mapping function, and we indent attributes, below the element to which they belong. From ϕ, the inverse mapping from the metamodel to F-Logic can be derived directly. We use the following notations:

- c represents a constant;
- v, v_1, ..., v_n represent variables;
- f represents a function symbol;
- p represents a predicate symbol;
- b represents a built-in predicate;
- s represents a string;
- t, t_1, ..., t_n represent terms in general (standing for one of the specific types of terms in F-Logic);
- d, d_1, ..., d_n represent methods (standing for one of the specific types of methods in F-Logic);
- a, a_1, ..., a_n represent method values (standing for one of the specific method values in F-Logic);
- h represents a host object for an F-molecule or F-atom (standing for one of the specific types of host objects);
- o_1, ..., o_n represent ontology formulas (standing for one of the specific formulas that can be directly in an F-Logic ontology: Query, Rule, P-atom, F-molecule, F-atom);
- m, m_1, ..., m_n represent formulas (standing for one of the specific types of formulas, including ontology formulas, connective formulas and quantifier formulas).

When a partial construct can be of different types and the mapping function values of these types are defined elsewhere in the table, we take the simplest of them for the sake of clarity in introducing the main construct.

F-Logic	**Metamodel = ϕ(F-Logic)**
F-Logic ontologies	
o_1. ... o_n	FLogicOntology
	ontologyFormulas = Set{$\phi(o_1)$, ..., $\phi(o_n)$ }
Rules and queries	
$m_1 \leftarrow m_2$	Rule
	containedFormula = Implication
	antecedent = $\phi(m_2)$
	consequent = $\phi(m_1)$
RULE s : $m_1 \leftarrow m_2$	Rule
	name = s
	containedFormula = Implication
	antecedent = $\phi(m_2)$
	consequent = $\phi(m_1)$
FORALL v_1 ... v_n $m_1 \leftarrow m_2$	Rule
	containedFormula = ForAll
	containedFormula = Implication
	antecedent = $\phi(m_2)$
	consequent = $\phi(m_1)$
	boundVariables = Set{
	Variable
	name = v_1
	isBoundToForAll = self.containedFormula
	...
	Variable
	name = v_n
	isBoundToForAll = self.containedFormula
	}
RULE s :	Rule
FORALL v_1 ... v_n $m_1 \leftarrow m_2$	name = s
	containedFormula = ForAll
	containedFormula = Implication
	antecedent = $\phi(m_2)$
	consequent = $\phi(m_1)$
	boundVariables = Set{
	Variable
	name = v_1
	isBoundToForAll = self.containedFormula
	...
	Variable
	name = v_n
	isBoundToForAll = self.containedFormula
	}

F-Logic	**Metamodel = ϕ(F-Logic)**
$\leftarrow m$	Query
	containedFormula = Implication
	antecedent = $\phi(m)$
	consequent = null
QUERY $s : \leftarrow m$	Query
	name = s
	containedFormula = Implication
	antecedent = $\phi(m)$
	consequent = null
FORALL v_1 ... $v_n \leftarrow m$	Query
	containedFormula = ForAll
	containedFormula = Implication
	antecedent = $\phi(m)$
	consequent = null
	boundVariables = Set{
	Variable
	name = v_1
	isBoundToForAll = self.containedFormula
	...
	Variable
	name = v_n
	isBoundToForAll = self.containedFormula
	}
QUERY s :	Query
FORALL v_1 ... $v_n \leftarrow m$	name = s
	containedFormula = ForAll
	containedFormula = Implication
	antecedent = $\phi(m)$
	consequent = null
	boundVariables = Set{
	Variable
	name = v_1
	isBoundToForAll = self.containedFormula
	...
	Variable
	name = v_n
	isBoundToForAll = self.containedFormula
	}

F-Logic	**Metamodel = ϕ(F-Logic)**
Logical connectives	
m_1 AND m_2 AND ... AND m_n	Conjunction connectedFormulas = Set{ $\phi(m_1)$, $\phi(m_2)$, ..., $\phi(m_n)$}
m_1 OR m_2 OR ... OR m_n	Disjunction connectedFormulas = Set{ $\phi(m_1)$, $\phi(m_2)$, ..., $\phi(m_n)$}

$m_1 \rightarrow m_2$	Implication
	antecedent = $\phi(m_1)$
	consequent = $\phi(m_2)$

$m_1 \leftarrow m_2$	Implication
	antecedent = $\phi(m_2)$
	consequent = $\phi(m_1)$

$m_1 \leftrightarrow m_2$	Equivalence connectedFormulas = Set{$\phi(m_1)$, $\phi(m_2)$}
NOT m	Negation connectedFormula = $\phi(m)$
Logical quantifiers	

EXISTS v_1 ... v_n m	Exists
	containedFormula = $\phi(m)$
	boundVariables = Set{
	Variable
	name = v_1
	isBoundToExists = self
	...
	Variable
	name = v_n
	isBoundToExists = self
	}

FORALL v_1 ... v_n m	ForAll
	containedFormula = $\phi(m)$
	boundVariables = Set{
	Variable
	name = v_1
	isBoundToForAll = self
	...
	Variable
	name = v_n
	isBoundToForAll = self
	}

F-Logic	**Metamodel = ϕ(F-Logic)**
F-atoms	

$t_1{:}t_2$	FAtom
	host = InstanceOf
	instance = $\phi(t_1)$
	class = $\phi(t_2)$

$t_1{::}t_2$	FAtom
	host = SubClassOf
	subclass = $\phi(t_1)$
	superclass = $\phi(t_2)$

$t_1[t_2 \Rightarrow t_3]$	FAtom
	host = $\phi(t_1)$
	methods = $\phi(t_2 \Rightarrow t_3)$

$t_1[t_2 \Rightarrow\!\!\!\!\!\!\Rightarrow t_3]$	FAtom
	host = $\phi(t_1)$
	methods = $\phi(t_2 \Rightarrow\!\!\!\!\!\!\Rightarrow t_3)$

$t_1[t_2 \rightarrow a_1]$	FAtom
	host = $\phi(t_1)$
	methods = $\phi(t_2 \rightarrow a_1)$

$t_1[t_2 \twoheadrightarrow \{t_3, t_4, ..., t_n\}]$	FAtom
	host = $\phi(t_1)$
	methods = $\phi(t_2 \twoheadrightarrow \{t_3, t_4, ..., t_n\})$

F-molecules	

$h[d_1; ...; d_n]$	FMolecule
	host = $\phi(h)$
	methods = Set $\{\phi(d_1), ..., \phi(d_n)\}$

Methods	

$t_1 \Rightarrow a_1$	SingleValuedMethodSignature
	name = $\phi(t_1)$
	value = $\phi(a_1)$

$t_1 \Rightarrow\!\!\!\!\!\!\Rightarrow a_1$	MultiValuedSignature
	name = $\phi(t_1)$
	value = $\phi(a_1)$

$t_1 \rightarrow a_1$	SingleValuedApplication
	name = $\phi(t_1)$
	value = $\phi(a_1)$

$t_1 \twoheadrightarrow \{a_1, a_2, ..., a_n\}$	MultiValuedApplication
	name = $\phi(t_1)$
	value = Set$\{\phi(a_1), \phi(a_2), ..., \phi(a_n)\}$

F-Logic	**Metamodel = ϕ(F-Logic)**

$t_1@(t_2, ..., t_n) \Rightarrow a_1$	SingleValuedMethodSignature
	name = $\phi(t_1)$
	parameters = Set{$\phi(t_2)$, ..., $\phi(t_n)$}
	value = $\phi(a_1)$

$t_1@(t_2, ..., t_n) \Rrightarrow a_1$	MultiValuedSignature
	name = $\phi(t_1)$
	parameters = Set{$\phi(t_2)$, ..., $\phi(t_n)$}
	value = $\phi(a_1)$

$t_1@(t_2, ..., t_n) \rightarrow a_1$	SingleValuedApplication
	name = $\phi(t_1)$
	parameters = Set{$\phi(t_2)$, ..., $\phi(t_n)$}
	value = $\phi(a_1)$

$t_1@(t_2, ..., t_n) \twoheadrightarrow \{a_1, a_2, ..., a_n\}$	MultiValuedApplication
	name = $\phi(t_1)$
	parameters = Set{$\phi(t_2)$, ..., $\phi(t_n)$}
	value = Set{$\phi(a_1)$, $\phi(a_2)$, ..., $\phi(a_n)$}

P-atoms	

p	PAtom
	predicate = PredicateSymbol
	name = p

$p(t_1, ..., t_n)$	PAtom
	predicate = PredicateSymbol
	name = p
	parameters = OrderedSet{$\phi(t_1)$, ..., $\phi(t_n)$}

Built-ins	

$b(t_1, ..., t_n)$	PAtom
	predicate = BuiltIn
	name = p
	parameters = OrderedSet{$\phi(t_1)$, ..., $\phi(t_n)$}

Terms	
c	Constant name = c

$f(t_1, ..., t_n)$	FunctionalTerm
	name = f
	arguments = OrderedSet{$\phi(t_1)$, ..., $\phi(t_n)$}

v	Variable name = v

A.7. Detailed Overview of the OWL Ontology Mappings Metamodel Extension

Next to introducing the OWL ontology mapping extension for the OWL metamodel in Chapter 7 starting on page 91, this appendix presents an alphabetically ordered overview of the different metaclasses of the metamodel extension according to the following specification conventions as also used in the UML specifications:

Description
An informal description of the metaclass.

Attributes, Associations and Generalizations
All attributes of the metaclass, ends of associations that start from the considered class, as well as generalizations of the class are listed together with a short comment.

Constraints
Constraints are expressed in OCL. They define invariants for the metaclass that must be fulfilled by all instances of that metaclass. The context of a constraint is always the considered metaclass.

1 - Containment

Description
A semantic relation to define that the source element is contained in the target element.

Attributes, Associations and Generalizations

- direction: String[1] specifies in which direction the containment is defined between the mappable elements. Possible values for this attribute are 'sound' and 'complete'.
- Specializes class SemanticRelation.

Constraints
No constraints.

2 - Equivalence

Description
A semantic relation to define that two mappable elements are equivalent.

Attributes, Associations and Generalizations

- Specializes class SemanticRelation.

Constraints
No constraints.

3 - MappableElement
Description
An abstract supertype for all types of mappable elements, which are elements that are mapped to each other in mapping assertions.

Attributes, Associations and Generalizations
No attributes, associations or generalizatons.

Constraints
No constraints.

4 - Mapping
Description
A set of mapping assertions between two ontologies.

Attributes, Associations and Generalizations

- domainAssumption: String[1] specifies the relationship between the connected domains. Possible values are 'overlap', 'soundContainment', 'completeContainment' and 'equivalence'.
- inconsistencyPropagation: Boolean[1] specifies whether inconsistencies are preserved across mapped ontologies.
- uniqueNameAssumption: Boolean[1] specifies whether names in the mappings are unique.
- URI: String[0..1] specifies an identification for the mapping. To assure a general metamodel, we do not define the URI to be mandatory.
- sourceOntology: Ontology[1] links the mapping to its source ontology.
- targetOntology: Ontology[1] links the mapping to its target ontology.
- elementMappings: MappingAssertion[*] links a mapping to its mapping assertions.

Constraints
No constraints.

5 - MappingAssertion
Description

A concrete mapping between two mappable elements.

Attributes, Associations and Generalizations

- sourceElement: MappableElement[1] links the mapping assertion to its source element.
- targetElement: MappableElement[1] links the mapping assertion to its target element.
- relationType: SemanticRelation[1] specifies the type of mapping relation between the two elements.

Constraints
No constraints.

6 - OntologyQuery
Description
A query is a more expressive mappable element.

Attributes, Associations and Generalizations

- Specializes class MappableElement,
- queryAtoms: Atom[1..*] links the query to its atoms. A query contains at least one atom.
- distinguishedVariables: Variable[*] specifies the variables that are returned by the query.

Constraints

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)))

7 - Overlap
Description
A semantic relation to define overlap between two mappable elements.

Attributes, Associations and Generalizations

- Specializes class SemanticRelation.

Constraints
No constraints.

8 - OWLEntity (augmented definition of the metaclass in the OWL metamodel)
Attributes, Associations and Generalizations

- Specializes class MappableElement.

9 - SemanticRelation
Description
An abstract supertype of all types of semantic relations in mappings.

Attributes, Associations and Generalizations

- negated: Boolean[1] specifies the negated version of the different semantic relations of mappings. When no value for this attribute is specified, the default value is 'false'.
- interpretation: String[1] specifies whether the mapping assertion is to be interpreted intentionally or extensionally. Possible values are 'intensional' and 'extensional'. By default, the attribute value is set to 'intensional'.

Constraints
No constraints.

A.8. Mappings between C-OWL and the Ontology Mapping Metamodel

As a consistent extension of Appendices A.2 stand A.4, the table below defines the values of the mapping function ρ for OWL ontology mapping elements in the language C-OWL. The function ρ maps the C-OWL elements to elements of the metamodel. Each C-OWL element is presented and is followed by its corresponding metamodel element, its attributes and associations. In doing so, we partially fall back on the OCL syntax. Attributes are indented, below the element to which they belong. From the function ρ, the inverse mapping from the metamodel to C-OWL can be derived directly. The following notations are used in the definition of the mapping function:

- o_1 and o_2 represent an ontology;
- m_1, ..., m_n represent a mapping assertion;
- e_1, ..., e_2 represent a mappable element.

C-OWL	**Metamodel = ρ(C-OWL)**
Mappings	

$(o_1, o_2, m_1 ... m_n)$	Mapping
	  domainAssumption = 'everlap'
	  inconsistencyPropagation = false
	  uniqueNameAssumption = false
	  sourceOntology = $\rho(o_1)$
	  targetOntology = $\rho(o_2)$
	  elementMappings = Set{$\rho(m_1)$, ..., $\rho(m_n)$}

Mapping assertions	

$e_1 \xrightarrow{\sqsubseteq} e_2$	MappingAssertion
	  relationType = Containment
	    direction = 'sound'
	    negated = false
	    interpretation = 'intensional'
	  sourceElement = $\rho(e_1)$
	  targetElement = $\rho(e_2)$

$e_1 \xrightarrow{\sqsupseteq} e_2$	MappingAssertion
	  relationType = Containment
	    direction = 'complete'
	    negated = false
	    interpretation = 'intensional'
	  sourceElement = $\rho(e_1)$
	  targetElement = $\rho(e_2)$

$e_1 \xrightarrow{\equiv} e_2$	MappingAssertion
	  relationType = Equivalence
	    negated = false
	    interpretation = 'intensional'
	  sourceElement = $\rho(e_1)$
	  targetElement = $\rho(e_2)$

$e_1 \xrightarrow{*} e_2$	MappingAssertion
	  relationType = Overlap
	    negated = false
	    interpretation = 'intensional'
	  sourceElement = $\rho(e_1)$
	  targetElement = $\rho(e_2)$

$e_1 \xrightarrow{\perp} e_2$	MappingAssertion
	  relationType = Overlap
	    negated = true
	    interpretation = 'intensional'
	  sourceElement = $\rho(e_1)$
	  targetElement = $\rho(e_2)$

A.9. Mappings between DL-Safe Mappings and the Ontology Mapping Metamodel

As a consistent extension of Appendices A.2 stand A.4, the table below defines the values of the mapping function ρ for OWL ontology mapping elements in the language DL-Safe Mappings. The function ρ maps the elements to elements of the metamodel. Each element of DL-Safe Mappings is presented and is followed by its corresponding metamodel element, its attributes and associations. In doing so, we partially fall back on the OCL syntax. Attributes are indented, below the element to which they belong. From the function ρ, the inverse mapping from the metamodel to DL-Safe Mappings can be derived directly. The following notations are used in the definition of the mapping function:

- o_1 and o_2 represent an ontology;
- $m_1, ..., m_n$ represent a mapping assertion;
- $e_1, ..., e_2$ represent a mappable element;
- $a_1, ..., a_n$ represent an atom;
- $s_1, ..., s_n, t_1, ..., t_n$ represent a string.

DL Safe Mappings	**Metamodel = ρ(DL-Safe Mappings)**
Mappings	

$(o_1, o_2, m_1 ... m_n)$	Mapping
	domainAssumption = 'equivalence'
	inconsistencyPropagation = true
	uniqueNameAssumption = false
	sourceOntology = $\rho(o_1)$
	targetOntology = $\rho(o_2)$
	elementMappings = Set{$\rho(m_1)$, ..., $\rho(m_n)$}

Mapping assertions	

$e_1 \sqsubseteq e_2$	MappingAssertion
	relationType = Containment
	direction = 'sound'
	negated = false
	interpretation = 'extensional'
	sourceElement = $\rho(e_1)$
	targetElement = $\rho(e_2)$

$e_1 \sqsupseteq e_2$	MappingAssertion
	relationType = Containment
	direction = 'complete'
	negated = false
	interpretation = 'extensional'
	sourceElement = $\rho(e_1)$
	targetElement = $\rho(e_2)$

$e_1 \equiv e_2$	MappingAssertion
	relationType = Equivalence
	negated = false
	interpretation = 'extensional'
	sourceElement = $\rho(e_1)$
	targetElement = $\rho(e_2)$

Queries	

Q({s_1, ..., s_n}, {t_1, ..., t_n}) = $a_1 \wedge ... \wedge a_n$	OntologyQuery
	queryAtoms = Set{$\rho(a_1)$, ..., $\rho(a_n)$}})
	distinguishedVariables = Set{
	Variable
	name = s_1
	...
	Variable
	name = s_n
	}

A.10. Mappings between the Metamodel and the UML Profile

In the table below we define the correspondences between the metamodel for OWL, SWRL and OWL mappings which we defined in Chapters 4 and 7 on the one hand, and the profile we defined in Chapter 8 on the other hand.

The first two columns represent the metamodel classes and attributes (including associations). In doing so, a meta attribute always belongs to the metaclass defined just before the attribute. The third and last column represents the value for the metamodel elements under the mapping function ψ, which maps them to elements of the UML profile. The mapping function value for an attribute is added to the value for the metaclass. Whereas Chapter 8 presented the profile with its visual notations, the last column in this table presents the profile using the UML metamodel elements and the additional stereotypes and tags we defined on it in the profile. Names of classes in the UML metamodel are written upper case, whereas attributes or associations in the UML metamodel are written lower case. Additionally, UML metamodel attributes are indented below the UML metamodel class to which they belong. When a stereotype is written with a UML element name, this means our profile applies this stereotype to that specific UML element.

A metaclass name written in italics denotes an abstract class of which we map an attribute which is applicable to all the subclasses. In this way, we do not need to define the function value of this attribute for each subclass. Naturally, no function value is defined for the abstract class itself.

Although the table only introduces ψ, the mapping in the other direction from the profile to the metamodel can directly be derived from ψ.

For the elements for which our profile provides different notations, the function denotes which one is the default notation. Consequently, this means that when we would map a UML diagram that uses the alternative notations to the metamodel and back, we would get a slightly different UML diagram. The difference would be that all constructs that were depicted using the alternative notation before, are depicted in the default notation after the round-trip mapping. Otherwise, round-trip mapping under the function ψ always results in the same diagram.

Our profile provides a compact notation for class descriptions. This alternative notation, as well as notations with icons, are not included in the mapping function ψ in this table. For the alternative notation for restrictions on data properties, namely as class attributes, instead of applying the mapping function to all different combinations, we only include it for the construct *DataHasValue*. A similar approach we took for the different predicate symbols in rule atoms with variables. Except for the built-ins which are new in the rule part, the mapping function values for all predicate symbols are given earlier in the table. As the value for variables is given explicitly, this can be combined with the values for the different predicate symbols to obtain all different combinations of predicate symbols with terms.

In the table, we follow the structure of the metamodel-chapters, and sometimes rely on the OCL syntax to define elements.

Meta class	**Meta attribute**	**Profile construct = ψ (Meta element)**
Ontologies		
Ontology	URI = String s	Package <<Ontology>> name = s
	importedOntology = o	Dependency <<OWLImport>> supplier $\psi(o)$
	ontologyAnnotation = a	$\psi(a)$
	ontologyAxiom = a	PackageableElement $\psi(a)$
Annotations		
Annotation	annotationValue = Constant c	Comment <<ExplicitAnnotation>> body = $\psi(c)$
	URI = String s	Tag type = s
Label		Comment <<Label>>
Comment		Comment <<Comment>>
Constant		
	languageTag = String s_1 value = String s_2	$s_2[@s_1]$
	URI = String s_1 value = String s_2	s_2 '^^' s_1
Entities		
OWLEntity		
	URI = String s	name = s
Datatype		(default) Class <<Datatype>> <<Primitive>> (alternative) DataType
OWLClass		Class <<OWLClass>>
ObjectProperty		Class <<ObjectProperty>>
DataProperty		Class <<DataProperty>>
Individual		InstanceSpecification
Declaration	entity = OWLEntity e	$\psi(e)$
EntityAnnotation	entity = OWLEntity e entityAnnotation = Annotation a	$\psi(e)$ $\psi(a)$

Meta class	**Meta attribute**	**Profile construct = ψ (Meta element)**
Data ranges		
DataComplementOf	dataRange = DataRange d	$\psi(d)$ <<DataComplementOf>>

DatatypeRestriction	dataRange = DataRange d
	datatypeFacet = String s
	restrictionValue = Constant c
		$\psi(d)$
		attribute 1 = Property
		name = 'datatypeFacet'
		defaultValue = s
		attribute 2 = Property
		name = 'restrictionValue'
		defaultValue = $\psi(c)$

DataOneOf		Class
	constants = Constant Set { c_1, ..., c_n}	Dependency supplier = Connector <<DataOneOf>> Dependency supplier = InstanceSpecification c_1 ... Dependency supplier = InstanceSpecification c_n
Class descriptions		
ObjectUnionOf	classes = Description Set {d_1, d_2, ..., d_n}	Class Dependency supplier = Connector <<ObjectUnionOf>> Dependency supplier = $\psi(d_1)$ Dependency supplier = $\psi(d_2)$ ... Dependency supplier = $\psi(d_n)$

Meta class	**Meta attribute**	**Profile construct =** ψ **(Meta element)**

ObjectIntersectionOf	classes = Description Set $\{d_1, d_2, ..., d_n\}$	Class
		  Dependency
		    supplier = Connector
		    <<ObjectIntersectionOf>>
		      Dependency
		        supplier = $\psi(d_1)$
		      Dependency
		        supplier = $\psi(d_2)$
		      ...
		      Dependency
		        supplier = $\psi(d_n)$

ObjectComplementOf	class = Description d	Class
		  Dependency
		  <<ObjectComplementOf>>
		    supplier = $\psi(d)$

ObjectOneOf	individuals = Individual Set $\{i_1, ..., i_n\}$	Class
		  Dependency
		    supplier = Connector
		    <<ObjectOneOf>>
		      Dependency
		        supplier = $\psi(i_1)$
		      ...
		      Dependency
		        Supplier = $\psi(i_n)$

ObjectAllValuesFrom	class = Description d
	property = ObjectProperty-Expression o
		Class
		  Association
		  <<ObjectAllValuesFrom>>
		    name = o.entityURI.name
		    memberEnd = $\psi(d)$

ObjectSomeValuesFrom	class = Description d
	property = ObjectProperty-Expression o
		Class
		  Association
		  <<ObjectSomeValuesFrom>>
		    name = o.entityURI.name
		    memberEnd = $\psi(d)$

Meta class	**Meta attribute**	**Profile construct =** **ψ (Meta element)**
ObjectHasValue	value = Individual *i* property = ObjectProperty- Expression *o*	Class Association <<ObjectSomeValuesFrom>> name = *o*.entityURI.name memberEnd = Class Dependency <<ObjectOneOf>> memberEnd = $\psi(i)$
ObjectExistsSelf	property = ObjectProperty- Expression *o*	Class Association <<ObjectExistsSelf>> name = *o*.entityURI.name memberEnd = self
ObjectExactCardinality	cardinality = Integer *n* class = Description *d* property = ObjectProperty- Expression *o*	Class Association <<ObjectCardinality>> name = *o*.entityURI.name lowerValue = n upperValue = n memberEnd = $\psi(d)$
ObjectMaxCardinality	cardinality = Integer *n* class = Description *d* property = ObjectProperty- Expression *o*	Class Association <<ObjectCardinality>> name = *o*.entityURI.name lowerValue = 0 upperValue = n memberEnd = $\psi(d)$
ObjectMinCardinality	cardinality = Integer *n* class = Description *d* property = ObjectProperty- Expression *o*	Class Association <<ObjectCardinality>> name = *o*.entityURI.name lowerValue = n upperValue = * memberEnd = $\psi(d)$

Meta class	**Meta attribute**	**Profile construct = ψ (Meta element)**

ObjectExactCardinality	cardinality = Integer *n*
	property = ObjectPropertyExpression *o*
		Class
		Association <<ObjectCardinality>>
		name = *o*.entityURI.name
		lowerValue = n
		upperValue = n
		memberEnd = Class <<OWLClass>>
		name = owl:Thing

ObjectMaxCardinality	cardinality = Integer *n*
	property = ObjectPropertyExpression *o*
		Class
		Association <<ObjectCardinality>>
		name = *o*.entityURI.name
		lowerValue = 0
		upperValue = n
		memberEnd = Class <<OWLClass>>
		name = owl:Thing

ObjectMinCardinality	cardinality = Integer *n*
	property = ObjectPropertyExpression *o*
		Class
		Association <<ObjectCardinality>>
		name = *o*.entityURI.name
		lowerValue = n
		upperValue = *
		memberEnd = Class <<OWLClass>>
		name = owl:Thing

DataAllValuesFrom	range = DataRange *d*
	properties = DataPropertyExpression *p*
		Class
		Association <<DataAllValuesFrom>>
		name = *p*.entityURI.name
		memberEnd = $\psi(d)$

DataSomeValuesFrom	range = DataRange *d*
	properties = DataPropertyExpression *p*
		Class
		Association <<DataSomeValuesFrom>>
		name = *p*.entityURI.name
		memberEnd = $\psi(d)$

Meta class	**Meta attribute**	**Profile construct =** **ψ (Meta element)**

DataHasValue	
	property =
	DataPropertyExpression *p*
	value = Constant *c*
		Class
		attribute = Property
		name = *p*.entityURI.name
		defaultValue = $\psi(c)$

DataExactCardinality	
	cardinality = Integer *n*
	range = DataRange *d*
	property =
	DataPropertyExpression *p*
		Class
		Association
		<<DataCardinality>>
		name =
		p.entityURI.name
		lowerValue = n
		upperValue = n
		memberEnd = $\psi(d)$

DataMaxCardinality	
	cardinality = Integer *n*
	range = DataRange *d*
	property =
	DataPropertyExpression *p*
		Class
		Association
		<<DataCardinality>>
		name =
		p.entityURI.name
		lowerValue = 0
		upperValue = n
		memberEnd = $\psi(d)$

DataMinCardinality	
	cardinality = Integer *n*
	range = DataRange *d*
	property =
	DataPropertyExpression *p*
		Class
		Association
		<<DataCardinality>>
		name =
		p.entityURI.name
		lowerValue = n
		upperValue = *
		memberEnd = $\psi(d)$

DataExactCardinality	 cardinality = Integer *n* property = DataPropertyExpression *p*	Class Association <<DataCardinality>> name = *p*.entityURI.name lowerValue = n upperValue = n memberEnd = Class <<Datatype>> name = rdfs:Literal

Meta class	**Meta attribute**	**Profile construct = ψ (Meta element)**

DataMaxCardinality	
	cardinality = Integer *n*
	property = DataPropertyExpression *p*
		Class
		Association
		<<DataCardinality>>
		name = *p*.entityURI.name
		lowerValue = 0
		upperValue = n
		memberEnd = Class
		<<Datatype>>
		name = rdfs:Literal

DataMinCardinality	
	cardinality = Integer *n*
	property = DataPropertyExpression *p*
		Class
		Association
		<<DataCardinality>>
		name = *p*.entityURI.name
		lowerValue = n
		upperValue = *
		memberEnd = Class
		<<Datatype>>
		name = rdfs:Literal

Axioms		
OWLAxiom		
	axiomAnnotation = a	$\psi(a)$
Class axioms		

SubClassOf	
	
	subClass = Description d_1
	superClass = Description d_2
		(default) Dependency
		<<SubClassOf>>
		client = $\psi(d_1)$
		supplier = $\psi(d_2)$
		(alternative) Generalization
		specific = $\psi(d_1)$
		general = $\psi(d_2)$

DisjointClasses	
	disjointClasses = Description Set $\{d_1, d_2\}$
		Two-way Dependency
		<<DisjointClasses>>
		client = $\psi(d_1)$
		supplier = $\psi(d_2)$

Meta class	**Meta attribute**	**Profile construct =** **ψ (Meta element)**
DisjointClasses	disjointClasses = Description Set $\{d_1, d_2, ..., d_n\}$	Connector <<DisjointClasses>> Dependency supplier = $\psi(d_1)$ Dependency supplier = $\psi(d_2)$ ... Dependency supplier = $\psi(d_n)$
DisjointUnion	unionClass = OWLClass c disjointClasses = Description Set $\{d_1, d_2, ..., d_n\}$	$\psi(c)$ Dependency supplier = Connector <<DisjointUnion>> Dependency supplier = $\psi(d_1)$ Dependency supplier = $\psi(d_2)$ ... Dependency supplier = $\psi(d_n)$

EquivalentClasses	equivalentClasses =
	Description Set $\{d_1, d_2\}$
		(default) Two-way Dependency
		<<EquivalentClasses>>
		client = $\psi(d_1)$
		supplier = $\psi(d_2)$
		(alternative) Two-way
		Generalization
		specific = $\psi(d_1)$
		general = $\psi(d_2)$

EquivalentClasses	equivalentClasses = Description Set $\{d_1, d_2, ..., d_n\}$	Connector <<EquivalentClasses>> Dependency supplier = $\psi(d_1)$ Dependency supplier = $\psi(d_2)$ ... Dependency supplier = $\psi(d_n)$

Meta class	**Meta attribute**	**Profile construct =** ψ **(Meta element)**
Object property axioms		

SubObjectPropertyOf	subProperties =
	ObjectPropertyExpression o_1
	superProperty =
	ObjectPropertyExpression o_2
		(default) Dependency
		<<SubObjectProperty>>
		client = $\psi(o_1)$
		supplier = $\psi(o_2)$
		(alternative) Generalization
		specific = $\psi(o_1)$
		general = $\psi(o_2)$

SubObjectPropertyOf	subProperties =
	ObjectProperty-
	Expression Set $\{o_1, ..., o_n\}$
	superProperty =
	ObjectProperty-
	Expression o_{n+1}
		(alternative) Generalization
		general = $\psi(o_{n+1})$
		specific = $\psi(o_1)$
		...
		specific = $\psi(o_n)$

EquivalentObject-
Properties
	equivalent-
	Properties =
	ObjectProperty-
	Expression Set
	$\{o_1, o_2\}$
		(default) Two-way Dependency
		<<EquivalentObjectProperties>>
		client = $\psi(o_1)$
		supplier = $\psi(o_2)$
		(alternative) Two-way
		Generalization
		specific = $\psi(o_1)$
		general = $\psi(o_2)$

EquivalentObject- Properties	equivalent- Properties = ObjectProperty- Expression Set $\{o_1, o_2, ..., o_n\}$	Connector <<EquivalentObjectProperties>> Dependency supplier = $\psi(o_1)$ Dependency supplier = $\psi(o_2)$ ... Dependency supplier = $\psi(o_n)$

Meta class	**Meta attribute**	**Profile construct = ψ (Meta element)**

DisjointObjectProperties	disjointProperties = ObjectProperty-Expression Set $\{o_1, o_2\}$	Two-way Dependency <<DisjointObjectProperties>>
		client = $\psi(o_1)$
		supplier = $\psi(o_2)$

DisjointObjectProperties	disjointProperties = ObjectProperty-Expression Set $\{o_1, o_2, ..., o_n\}$	Connector <<DisjointObjectProperties>> Dependency supplier = $\psi(o_1)$ Dependency supplier = $\psi(o_2)$ ... Dependency supplier = $\psi(o_n)$

InverseObjectProperties	inverseProperties = ObjectProperty-Expression Set $\{o_1, o_2\}$	Two-way Dependency <<InverseObjectProperties>>
		client = $\psi(o_1)$
		supplier = $\psi(o_2)$

ObjectPropertyDomain
 	
ObjectPropertyRange
	property = Object-PropertyExpression o
	domain = Description d_1
	
	property = Object-PropertyExpression o
	range = Description d_2
		(default) Class
		<<ObjectProperty>>
		name = o.entityURI.name
		Dependency
		<<ObjectPropertyDomain>>
		supplier = $\psi(d_1)$
		Dependency
		<<ObjectPropertyRange>>
		supplier = $\psi(d_2)$
		(alternative) $\psi(d_1)$
		Association
		name = o.entityURI.name
		memberEnd = $\psi(d_2)$

FunctionalObjectProperty	property = Object-PropertyExpression o	$\psi(o)$ <<Functional>>
InverseFunctional-ObjectProperty	property = Object-PropertyExpression o	$\psi(o)$ <<InverseFunctional>>

Meta class	**Meta attribute**	**Profile construct = ψ (Meta element)**
TransitiveObjectProperty	property = Object-PropertyExpression o	$\psi(o)$ <<Transitive>>
ReflexiveObjectProperty	property = ObjectProperty-Expression o	$\psi(o)$ <<Reflexive>>
IrreflexiveObjectProperty	property = ObjectProperty-Expression o	$\psi(o)$ <<Irreflexive>>
SymmetricObjectProperty	property = ObjectProperty-Expression o	$\psi(o)$ <<Symmetric>>
AntisymmetricObjectProperty	property = ObjectProperty-Expression o	$\psi(o)$ <<Antisymmetric>>
Data property axioms		

SubDataPropertyOf	subProperty = DataProperty-Expression p_1
	superProperty = DataProperty-Expression p_2
		(default) Dependency <<SubDataProperty>>
		client = $\psi(p_1)$
		supplier = $\psi(p_2)$
		(alternative) Generalization
		specific = $\psi(p_1)$
		general = $\psi(p_2)$

EquivalentDataProperties	equivalent-Properties = DataProperty-Expression Set $\{p_1, p_2\}$	(default) Two-way Dependency <<EquivalentDataProperties>>
		client = $\psi(p_1)$
		supplier = $\psi(p_2)$
		(alternative) Two-way Generalization
		specific = $\psi(p_1)$
		general = $\psi(p_2)$

Meta class	**Meta attribute**	**Profile construct =** ψ **(Meta element)**
EquivalentDataProperties	equivalentProperties = DataProperty-Expression Set $\{p_1, p_2, ..., p_n\}$	Connector <<EquivalentDataProperties>> Dependency supplier = $\psi(p_1)$ Dependency supplier = $\psi(p_2)$ ... Dependency supplier = $\psi(p_n)$

DisjointDataProperties	disjointProperties = DataProperty-Expression Set $\{p_1, p_2\}$	Two-way Dependency <<DisjointDataProperties>>
		client = $\psi(p_1)$
		supplier = $\psi(p_2)$

DisjointDataProperties	disjointProperties = DataProperty-Expression Set $\{p_1, p_2, ..., p_n\}$	Connector <<DisjointDataProperties>> Dependency supplier = $\psi(p_1)$ Dependency supplier = $\psi(p_2)$ ... Dependency supplier = $\psi(p_n)$
FunctionalDataProperty	property = Data-PropertyExpression p	$\psi(p)$ <<Functional>>

DataPropertyDomain
 	
DataPropertyRange
	property = Data-PropertyExpression p
	domain = Description d_1
	
	property = Data-PropertyExpression p
	range = DataRange d_2
		(default) Class <<DataProperty>>
		name = p.entityURI.name
		Dependency <<DataPropertyDomain>>
		supplier = $\psi(d_1)$
		Dependency <<DataPropertyRange>>
		supplier = $\psi(d_2)$
		(alternative) $\psi(d_1)$
		Association
		name = p.entityURI.name
		memberEnd = $\psi(d_2)$

Meta class	**Meta attribute**	**Profile construct =** ψ **(Meta element)**
Facts		

DifferentIndividuals	differentIndividuals = Individual Set { i_1, i_2}	Connector <<DifferentIndividual>>
		end = $\psi(i_1)$
		end = $\psi(i_2)$

DifferentIndividuals	differentIndividuals = Individual Set { i_1, i_2, ..., i_n}	Connector <<DifferentIndividual>> Dependency supplier = $\psi(i_1)$ Dependency supplier = $\psi(i_2)$ ... Dependency supplier = $\psi(i_n)$

SameIndividual	sameIndividuals = Individual Set { i_1, i_2}	Connector <<SameIndividual>>
		end = $\psi(i_1)$
		end = $\psi(i_2)$

SameIndividual	sameIndividuals = Individual Set { i_1, i_2, ..., i_n}	Connector <<SameIndividual>> Dependency supplier = $\psi(i_1)$ Dependency supplier = $\psi(i_2)$ ... Dependency supplier = $\psi(i_n)$

ClassAssertion	individual = Individual *i*
	class = OWLClass *c*
		(default) InstanceSpecification
		name = *i*.entityURI.name
		classifier = *c*.entityURI.name
		(alternative) InstanceSpecification
		name = *i*.entityURI.name
		Dependency <<ClassType>>
		supplier = $\psi(c)$

ClassAssertion	individual = Individual *i*
	class = Description *d*
		InstanceSpecification
		name = *i*.entityURI.name
		Dependency <<ClassType>>
		supplier = $\psi(d)$

Meta class	**Meta attribute**	**Profile construct = ψ (Meta element)**
ObjectProperty-Assertion	property = ObjectPropertyExpression o source = Individual i_1 target = Individual i_2	$\psi(i_1)$ Association name = o.entityURI.name memberEnd = $\psi(i_2)$
NegativeObject-PropertyAssertion	property = ObjectPropertyExpression o source = Individual i_1 target = Individual i_2	$\psi(i_1)$ Association <<Not>> name = o.entityURI.name memberEnd = $\psi(i_2)$
DataProperty-Assertion	property = DataPropertyExpression p source = Individual i target = Constant c	InstanceSpecification name = i.entityURI.name attribute = Property name = p.entityURI.name defaultValue = $\psi(c)$
NegativeData-PropertyAssertion	property = DataPropertyExpression p source = Individual i targetValue = Constant c	InstanceSpecification name = i.entityURI.name attribute = Property <<Not>> name = p.entityURI.name defaultValue = $\psi(c)$

Meta class	**Meta attribute**	**Profile construct =** **ψ (Meta element)**
Rules		
Rule		Package <<Rule>>
	URI = String u	Tag URI = u
	ruleBody = Antecedent a	PackageableElement = $\psi(a)$
	ruleHead = Consequent c	PackageableElement = $\psi(c)$
Antecedent	bodyAtoms = Atom Set = $\{a_1, ..., a_n\}$	$\psi(a_1)$ <<Precondition>> ... $\psi(a_n)$ <<Precondition>>
Consequent	headAtoms = Atom Set = $\{a_1, ..., a_n\}$	$\psi(a_1)$ <<Postcondition>> ... $\psi(a_n)$ <<Postcondition>>

Atom	atomArguments =
	OrderedSet $\{t_1, ..., t_n\}$
	atomName = BuiltIn p
	URI = String u
		InstanceSpecification <<Built-in>>
		name = u
		Association
		name = '1'
		supplier = $\psi(t_1)$
		...
		Association
		name = 'n'
		supplier = $\psi(t_n)$

Terms		
Variable	name = String s	InstanceSpecification <<Variable>> name = s

Meta class	**Meta attribute**	**Profile construct =** **ψ (Meta element)**
Mappings		
Mapping		Class <<MappingDefinition>>

	domainAssumption =
	String s
		attribute = Property
		name = 'domainAssumption'
		defaultValue = s

	inconsistencyPropagation =
	Boolean b_1
		attribute = Property
		name =
		'inconsistencyPropagation'
		defaultValue = b_1

	uniqueNameAssumption =
	Boolean b_2
		attribute = Property
		name =
		'uniqueNameAssumption'
		defaultValue = b_2

	URI = String u	name = u
	sourceOntology = Ontology o_1	Dependency <<SourceOntology>> supplier = $\psi(o_1)$
	targetOntology = Ontology o_2	Dependency <<TargetOntology>> supplier = $\psi(o_2)$
	elementMappings = MappingAssertion Set = $\{m_1, ..., m_n\}$	Dependency <<Mapping>> supplier = $\psi(m_1)$ ... Dependency <<Mapping>> supplier = $\psi(m_n)$

MappingAssertion	relationType =
	SemanticRelation r
	sourceElement =
	MappableElement e_1
	targetElement =
	MappableElement e_2
		r-Iconed Dependency
		client = $\psi(e_1)$
		supplier = $\psi(e_2)$

Meta class	**Meta attribute**	**Profile construct =** **ψ (Meta element)**
Queries		
OntologyQuery		(when belonging to source ontology) Package <<SourceQuery>> (when belonging to target ontology) Package <<TargetQuery>>
	queryAtoms = Atom Set $\{a_1, ..., a_n\}$	PackageableElement $\psi(a_1)$ ... PackageableElement $\psi(a_n)$
	distinguishedVariables = Variable Set $\{v_1, ..., v_n\}$	PackageableElement $\psi(v_1)$ <<Distinguished>> ... PackageableElement $\psi(v_n)$ <<Distinguished>>

A.11. Questionnaire for the Summative Evaluation

A.11.1. A - Task Observation

A-1. Which editor did you use during the experiment?

Protégé	OntoModel

A-2. How would you rate your previous experience with the editor used in the test?

beginner	moderate	expert	NA/DK

A-3. How would you rate your previous experience in ontology engineering?

beginner	moderate	expert	NA/DK

A-4. How would you rate your previous experience in modeling with UML?

beginner	moderate	expert	NA/DK

A-5. How would you rate your previous experience in modeling in general?

beginner	moderate	expert	NA/DK

A-6. If applicable, which modeling approaches did you use already before the experiment?

Protégé: yes / no
OntoModel: yes / no
Topbraid: yes / no
Rational Software Architect: yes / no
Others (fill in):

A-7. How was your understanding of the tasks in the experiment?

very bad	bad	good	very good

A-8. How were the difficulties you needed to overcome during the experiment in order to complete task 1?

low	average	high

A-9. How were the difficulties you needed to overcome during the experiment in order to complete task 2?

low	average	high

A-10. How were the difficulties you needed to overcome during the experiment in order to complete task 3?

low	average	high

A-11. How did you find the support in the two tasks provided by the experimenter?

very inadequate	inadequate	good	very good

A.11.2. B - Usability

B-1. Did you find the approach for modeling ontologies intuitive?

not at all	not really	rather yes	for sure

B-2. Please indicate how difficult you found to get acquainted with the given ontology.

very difficult	rather difficult	rather easy	very easy

B-3. Please indicate how difficult you found to understand the different ontology constructs.

very difficult	rather difficult	rather easy	very easy

B-4. Are you satisfied with the choice of specific notations for the different ontology elements?

not at all	not really	rather yes	for sure

B-5. What was the main obstacle that you found during the tasks?

A.11.3. C - Effectiveness and Efficiency

C-1. Did you find the approach allows to set a clear and simple sequence of steps to accomplish each necessary action, e.g. create a new instance of a concept?

not really	rather yes	for sure

C-2. How was the overall effectiveness of the ontology modeling approach?

inadequate	adequate	excellent

A.12. Questionnaire for the Formative Evaluation

A.12.1. A - Task Observation

A-1. How was your understanding of the tasks in the experiment?

very bad	bad	good	very good

A-2.How did you find the support in the two tasks provided by the experimenter?

very inadequate	inadequate	good	very good

A-3. How would you rate your previous experience in ontology engineering?

beginner	moderate	expert	NA/DK

A-4. How would you rate your previous experience in modeling with UML?

beginner	moderate	expert	NA/DK

A-5. How would you rate your previous experience in modeling in general?

beginner	moderate	expert	NA/DK

A-6. How would you rate your previous experience in the invoicing domain?

beginner	moderate	expert	NA/DK

A-7. If applicable, which modeling approaches did you use already before the experiment?

Protégé: yes / no
OntoModel: yes / no

Topbraid: yes / no
Rational Software Architect: yes / no
Others (fill in):

A-8. How were the difficulties you needed to overcome during the experiment in order to complete task 1?

low	average	high

A-9. How were the difficulties you needed to overcome during the experiment in order to complete task 2?

low	average	high

A.12.2. B - Usability

B-1. Did you find the visual modeling of ontologies intuitive?

not at all	not really	rather yes	for sure

B-2. Did you find the visual modeling of ontology mappings intuitive?

not at all	not really	rather yes	for sure

B-3. Do you find the UML-notation suitable for modeling ontologies?

not at all	not really	rather yes	for sure

B-4. Please indicate how difficult you found to get acquainted with the Pharmainnova ontology using the visual model.

very difficult	rather difficult	rather easy	very easy

B-5. Was the visual model helpful to understand the different ontology constructs?

not at all	not really	rather yes	for sure

B-6. Are you satisfied with the choice of specific notations (box, arrow, ...) for the different ontology elements?

not at all	not really	rather yes	for sure

If not, which notations do you recommend?

B-7. What was the main obstacle that you found during the tasks?

B-8. Please, briefly list any suggestion to improve usability.

A.12.3. C - Effectiveness and Efficiency

C-1. Did you find the approach allows to set a clear and simple sequence of steps to accomplish each necessary action, e.g. create a new instance of a concept?

not really	rather yes	for sure

C-2. How was the overall effectiveness of the approach?

inadequate	adequate	excellent

C-3. Please write a list with approximately the five most repeated operations during the experiment.

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). `http://www.w3.org/Submission/WRL/`, September 2005. W3C Member Submission.

[BBN99] M. Biezunski, M. Bryan, and S. Newcomb. Topic Maps: Information Technology – Document Description and Markup Languages. ISO/IEC 13250, `http://www.y12.doe.gov/sgml/sc34/document/0129.pdf`, December 1999.

[BCH06] J. Bao, D. Caragea, and V. Honavar. Modular Ontologies - A Formal Investigation of Semantics and Expressivity. In R. Mizoguchi, Z. Shi, and F. Giunchiglia, editors, *Asian Semantic Web Conference 2006*, volume 4185 of *LNCS*, pages 616–631. Springer, 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 City, New York, 2003.

[Bec04] D. Beckett. RDF/XML Syntax Specification. `http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/`, February 2004.

[BEH$^+$02] E. Bozsak, M. Ehrig, S. Handschuh, A. Hotho, A. Maedche, B. Motik, D. Oberle, C. Schmitz, S. Staab, L. Stojanovic, N. Stojanovic, R. Studer, G. Stumme, Y. Sure, J. Tane, R. Volz, and V. Zacharias. KAON - Towards a large scale Semantic Web. In K. Bauknecht, A. M. Tjoa, and G. Quirchmayr, editors, *E-Commerce and Web Technologies*, volume 2455 of *LNCS*, pages 304–313, Aix-en-Provence, France, September 2002. Springer.

[BG04] D. Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. `http://www.w3.org/TR/rdf-schema/`, February 2004.

[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.

[BGSSH06] S. Brockmans, A. Geyer-Schulz, R. Studer, and P. Hitzler. Visual Ontology Modeling for Electronic Markets. In T. Dreier, R. Studer, and C. Weinhardt, editors, *Information Management and Market Engineering*, pages 85–99, Karlsruhe, Germany, September 2006. Universitätsverlag Karlsruhe.

[BGvH+03] P. Bouquet, F. Giunchiglia, F. van Harmelen, L. Serafini, and H. Stuckenschmidt. C-OWL: Contextualizing Ontologies. In D. Fensel, K. Sycara, and J. Mylopoulos, editors, *The Semantic Web - ISWC 2003*, volume 2870 of *LNCS*, pages 164–179. Springer, October 2003.

[BHGS01] S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. OilEd: a Reason-able Ontology Editor for the Semantic Web. In F. Baader, editor, *Proceedings of KI2001*, volume 2174 of *LNCS*, pages 396–408, Vienna, September 2001. Springer.

[BHHS06] S. Brockmans, P. Haase, P. Hitzler, and R. Studer. A Metamodel and UML Profile for Rule-extended OWL DL Ontologies. In Y. Sure and J. Domingue, editors, *The Semantic Web: Research and Applications*, volume 4011 of *LNCS*, pages 303–316, Budva, Montenegro, June 2006. Springer.

[BHS06a] S. Brockmans, P. Haase, and H. Stuckenschmidt. Formalism-Independent Specification of Ontology Mappings - A Metamodeling Approach. In R. Meersman and Z. Tari, editors, *OTM 2006 Conferences*, volume 4275 of *LNCS*, pages 901–908, Montpellier, France, October 2006. Springer.

[BHS06b] S. Brockmans, P. Haase, and R. Studer. A MOF-based Metamodel and UML Syntax for Networked Ontologies. In *International Semantic Web Conference 2006 Workshop Proceedings*, Athens, Georgia, November 2006.

[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 M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *LNCS*, pages 342–260, Toronto, Canada, October 2001. Springer.

[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.

[BMW+07] J.-S. Brunner, L. Ma, C. Wang, L. Zhang, D. C. Wolfson, Y. Pan, and K. Srinivas. Explorations in the Use of Semantic Web Technologies for Product Information Management. In P. Patel-Schneider and P. Shenoy, editors, *Proceedings of the Sixteenth International World Wide Web Conference (WWW 2007)*, Banff, Alberta, Canada, May 2007. Association for Computing Machinery (ACM).

[BPSM+06] T. Bray, J. Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0. `http://www.w3.org/TR/REC-xml/`, September 2006.

[Bra79] R.J. Brachman. On the Epistemological Status of Semantic Nets. In N.V. Findler, editor, *Associative Networks: The Representation and Use of Knowledge by Computers*, pages 3–50, New York, June 1979.

[BS03] A. Borgida and L. Serafini. Distributed Description Logics: Assimilating Information from Peer Sources. *Journal of Data Semantics*, 1:153–184, 2003.

[BVEL04] S. Brockmans, R. Volz, A. Eberhart, and P. Loeffler. Visual Modeling of OWL DL Ontologies using UML. In F. van Harmelen, S. A. McIlraith, and D. Plexousakis, editors, *The Semantic Web - ISWC 2004*, volume 3298 of *LNCS*, pages 198–213, Hiroshima, Japan, November 2004. Springer.

[CDL01] D. Calvanese, G. De Giacomo, and M. Lenzerini. A Framework for Ontology Integration. In I. Cruz, S. Decker, J. Euzenat, and D. McGuinness, editors, *Proceedings of the Semantic Web Working Symposium*, pages 303–316, Stanford, CA, July 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 *LNCS*, pages 41–60. Springer, July 2002.

[Che76] P. P. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

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

[CPL+05] D. K. W. Chiu, J. K. M. Poon, W. C. Lam, C. Y. Tse, W. H. T. Sui, and W. S. Poon. How Ontologies can Help in an E-Marketplace. In *Proceedings of the 13th European Conference on Information Systems, Information Systems in a Rapidly Changing Economy, ECIS 2005*, Regensburg, Germany, May 2005.

[CRH+06] R. Colomb, K. Raymond, L. Hart, P. Emery, C. Welty, G. T. Xie, and E. Kendall. The Object Management Group Ontology Definition Metamodel. In C. Calero, F. Ruiz, and M. Piattini, editors, *Ontologies for Software Engineering and Technology*, pages 217–248. Springer, 2006.

[Del06] H. Delugach. Information technology - Common Logic (CL) - A framework for a family of logic-based languages. ISO/IEC FDIS 24707, `Internet:http://cl.tamu.edu/docs/cl/24707-31-Dec-2006.pdf`, December 2006.

[DMB+06] M. Dzbor, E. Motta, C. Buil, J. Gomez, O. Görlitz, and H. Lewen. Developing ontologies in OWL: An observational study. In B. Cuenca Grau, P. Hitzler, C. Shankey, and E. Wallace, editors, *Proceedings of OWL: Experiences and Directions 2006*, volume 216 of *CEUR Workshop Proceedings*, Athens, Georgia, November 2006.

[dSM06] M. d'Aquin, M. Sabou, and E. Motta. Modularization: a Key for the Dynamic Selection of Relevant Knowledge Components. In *Workshop on Modular Ontologies*, Athens, Georgia, November 2006.

[DST03] DSTC. Ontology Definition MetaModel Initial Submission. `http://www.omg.org/docs/ad/03-08-01.pdf`, August 2003.

[FHK+97] J. Frohn, R. Himmeröder, P.-T. Kandzia, G. Lausen, and C. Schlepphorst. FLORID: A Prototype for F-Logic. In W. A. Gray and P. Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering*, page 583, Birmingham UK, April 1997. IEEE Computer Society.

[Fow03] M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, Massachusetts, 2003.

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

[FW05] C. Fillies and F. Weichhardt. Semantically correct Visio Drawings. In A. Gmez Prez, editor, *Proceedings of the 2nd European Semantic Web Conference*, volume 3532 of *LNCS*, Heraklion, Crete, May 2005. Springer.

[Gai91] B. R. Gaines. An Interactive Visual Language for Term Subsumption Languages. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 817–823, Sydney, Australia, August 1991. Morgan Kaufmann.

[GDD07] D. Gašević, D. Djurić, and V. Devedžić. MDA-based automatic OWL ontology development. *International Journal on Software Tools for Technology Transfer (TTT)*, 9(2):103–117, 2007.

[Gen03] Gentleware. Ontology Definition Meta-Model. `http://www.omg.org/docs/ad/03-08-09.pdf`, August 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*, Washington, DC, April 2005.

[GM05] L. M. Garshol and G. Moore. Topic Maps - Data Model. ISO/IEC 13250-2, `http://www.isotopicmaps.org/sam/sam-model/`, December 2005.

[GM06] B. Cuenca Grau and B. Motik. OWL 1.1 Web Ontology Language - Model-Theoretic Semantics. `http://owl1-1.cs.manchester.ac.uk/semantics.html`, November 2006.

[GM07] B. Cuenca Grau and B. Motik. OWL 1.1 Web Ontology Language - Mapping to RDF Graphs. `http://owl1-1.cs.manchester.ac.uk/rdf\_mapping.html`, February 2007.

[GMF04] R. Guha, R. McCool, and R. Fikes. Contexts for the Semantic Web. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *Proceedings of the International Semantic Web Conference*, volume 3298 of *LNCS*. Springer, November 2004.

[GMPS07] B. Cuenca Grau, B. Motik, and P. Patel-Schneider. OWL 1.1 Web Ontology Language - XML Syntax. `http://owl1-1.cs.manchester.ac.uk/xml\_syntax.html`, February 2007.

[Gro05a] Object Management Group. MOF QVT Final Adopted Specification. `http://www.omg.org/docs/ptc/05-11-01.pdf`, November 2005.

[Gro05b] Object Management Group. Unified Modeling Language: Superstructure. `http://www.omg.org/docs/formal/05-07-04.pdf`, July 2005.

[Gro06a] Object Management Group. Object Constraint Language. `http://www.omg.org/docs/formal/06-05-01.pdf`, May 2006.

[Gro06b] Object Management Group. Ontology Definition Metamodel. `http://www.omg.org/cgi-bin/doc?ad/2006-10-11`, October 2006.

[HEC$^+$04] L. Hart, P. Emery, B. Colomb, K. Raymond, S. Taraporewalla, D. Chang, Y. Ye, and M. Dutra E. Kendall. OWL Full and UML 2.0 Compared. `http://www.itee.uq.edu.au/$\sim$colomb/Papers/UML-OWLont04.03.01.pdf`, March 2004.

[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)*, pages 57–67, Lake District, UK, June 2006. AAAI Press.

[HM05] P. Haase and B. Motik. A Mapping System for the Integration of OWL-DL Ontologies. In *Proceedings of the ACM-Workshop: Interoperability of Heterogeneous Information Systems (IHIS05)*, Bremen, Germany, November 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. `http://www.w3.org/Submission/2004/SUBM-SWRL-20040521/`, May 2004.

[HRW$^+$06] P. Haase, S. Rudolph, Y. Wang, S. Brockmans, R. Palma, J. Euzenat, and M. d'Aquin. D1.1.1 Networked Ontology Model. Technical Report D1.1.1, Universität Karlsruhe (TH), November 2006.

[IBM03] IBM. Ontology Definition Metamodel (ODM) Proposal. `http://www.omg.org/docs/ad/03-07-02.pdf`, August 2003.

[IKSL03] Sandpiper Software Inc. and Stanford University Knowledge Systems Laboratory. UML for Knowledge Representation - A Layered, Component-Based Approach to Ontology Development. `http://www.omg.org/docs/ad/03-08-06.pdf`, March 2003.

[JK06] F. Jouault and I Kurtev. On the Architectural Alignment of ATL and QVT. In *Proceedings of SAC '06*, Dijon, France, April 2006.

[KC04] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. `http://www.w3.org/TR/rdf-concepts/`, February 2004.

[KLW95] M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the Association for Computing Machinery*, 42(4):741–843, July 1995.

[Kre98] R. Kremer. Visual Languages for Knowledge Representation. In *Proceedings of 11th Workshop on Knowledge Acquisition, Modeling and Management (KAW'98)*, Banff, Canada, April 1998. Morgan Kaufmann. `http://ksi.cpsc.ucalgary.ca/KAW/KAW98/kremer/`.

[LW06a] S. Lukichev and G. Wagner. UML-Based Rule Modeling with Fujaba. In B. Westfechtel and H. Giese, editors, *Proceedings of 4th International Fujaba Days 2006*, Bayreuth, Germany, September 2006.

[LW06b] S. Lukichev and G. Wagner. Visual Rules Modeling. In A. Voronkov and I. Virbitskaite, editors, *Proceedings of Sixth International Conference Perspectives of Systems Informatics*, volume 4378 of *LNCS*, pages 467–673, Novosibirsk, Russia, June 2006. Springer.

[MB02] S. J Mellor and M. J. Balcer. *Executable UML: A Foundation for Model Driven Architecture*. Addison Wesley Professional, 2002.

[MHRS06] B. Motik, I. Horrocks, R. Rosati, and U. Sattler. Can OWL and Logic Programming Live Together Happily Ever After? In I. F. Cruz, S. Decker, D. Allemang, C. Preist, D. Schwabe, P. Mika, M. Uschold, and L. Aroyo, editors, *International Semantic Web Conference*, volume 4273 of *LNCS*, pages 501–514, Athens, Georgia, November 2006. Springer.

[MKW04] S. J. Mellor, S. Kendall, and A. Uhl D. Weise. *MDA Distilled.* Addison Wesley Longman Publishing Co., Inc., Redwood City, California, 2004.

[MR07] B. Motik and R. Rosati. A Faithful Integration of Description Logics with Logic Programming. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI2007)*, pages 477–482, Hyderabad, India, January 2007. Morgen Kaufmann.

[MSS04] B. Motik, U. Sattler, and R. Studer. Query Answering for OWL-DL with Rules. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *The Semantic Web - ISWC 2004*, volume 3298 of *LNCS*, pages 549–563, Hiroshima, Japan, November 2004. Springer.

[Mv03] D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. `Internet:http://www.w3.org/TR/owl-features/`, August 2003.

[Nie93] J. Nielsen. *Usability Engineering.* Morgan Kaufmann, 1993.

[Obj03] Object Management Group. Ontology Definition Metamodel - Request For Proposal. `http://www.omg.org/docs/ontology/03-03-01.rtf`, March 2003.

[Obj06] Object Management Group. Meta Object Facility (MOF) Core Specification. `http://www.omg.org/docs/formal/06-01-01.pdf`, January 2006.

[Ont06] Ontoprise. F-Logic-Programs. `http://www.ontoprise.de/documents/tutorial_flogic.pdf`, January 2006.

[PSHM07] P. F. Patel-Schneider, I. Horrocks, and B. Motik. OWL 1.1 Web Ontology Language - Structural Specification and Functional-Style Syntax. `Ihttp://owl1-1.cs.manchester.ac.uk/syntax.html`, February 2007.

[Sch02] W. Schnotz. Wissenserwerb mit Texten, Bildern und Diagrammen. In L. J. Issing and P. Klimsa, editors, *Information und Lernen mit Multimedia und Internet*, pages 65–81. Belz, PVU, Weinheim, Germany, third, completely revised edition, 2002.

[SK03] H. Stuckenschmidt and M. C. A. Klein. Integrity and Change in Modular Ontologies. In G. Gottlob, editor, *18th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 900–908, Acapulco, Mexico, August 2003.

[Sow92] J. F. Sowa. Conceptual Graphs Summary. In P. Eklund, T. Nagle, J. Nagle, and L. Gerholz, editors, *Conceptual Structures: Current Research and Practice*, pages 3–52. Ellis Horwood, New York, 1992.

[SP04] Evren Sirin and Bijan Parsia. Pellet: An OWL DL Reasoner. In *Description Logics*, 2004.

[SR06] J. Seidenberg and A. Rector. Web Ontology Segmentation: Analysis, Classification and Use. In C. Goble and M. Dahlin, editors, *Proceedings of the World Wide Web Conference (WWW)*, Edinburgh, UK, June 2006. ACM Press.

[SS89] M. Schmidt-Schauss. Subsumption in KL-ONE is Undecidable. In R. J. Brachman, H. J. Levesque, and R. Reiter, editors, *Proceedings of the First International Conference on the Principles of Knowledge Representation and Reasoning (KR-89)*, pages 421–431, Toronto, Canada, May 1989. Morgan Kaufmann.

[SSW05] 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.

[SU05] H. Stuckenschmidt and M. Uschold. Representation of Semantic Mappings. In Y. Kalfoglou, M. Schorlemmer, A. Sheth, S. Staab, and M. 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. `http://www.w3.org/TR/xmlschema-1/`, October 2004.

[TF05] S. Tessaris and E. Franconi. Rules and Queries with Ontologies: a Unifying Logical Framework. In I. Horrocks, U. Sattler, and F. Wolter, editors, *Proceedings of the 2005 International Workshop on Description Logics (DL2005)*, volume 147 of *CEUR Workshop Proceedings*, Edinburgh, Scotland, UK, July 2005. CEUR-WS.org.

[TV56] A. Tarski and R. Vaught. Arithmetical Extensions of Relational Systems. *Compositio Mathematica*, 13:81–102, 1956.

[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. `http://www.daml.org/2001/03/reference.html`, March 2001.

[Vir92] R. A. Virzi. Refining the test phase of usability evaluation: How many subjects is enough? *Human Factors*, 34(4):457–468, 1992.

[Vol04] R. Volz. *Web Ontology Reasoning with Logic Databases.* Phd thesis, Universität Karlsruhe (TH), Karlsruhe, Germany, http://www.ubka.uni-karlsruhe.de/cgi-bin/psview?document=2004/wiwi/2, February 2004.

[W3C05a] *Accepted Papers of the W3C Workshop on Rule Languages for Interoperability*, Washington, DC, April 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.

[WAL05] G. Wagner, A.Giurca, and S. Lukichev. R2ML: A General Approach for Marking up Rules. In F. Bry, F. Fages, M. Marchiori, and H. Ohlbach, editors, *Principles and Practices of Semantic Web Reasoning*, volume 05371 of *Principles and Practices of Semantic Web Reasoning*, 2005.

[WATB04] G. Wagner, G. Antoniou, S. Tabet, and H. Boley. The Abstract Syntax of RuleML - Towards a General Web Rule Language Framework. In *Web Intelligence 2004*, pages 628–631, Beijing, China, September 2004.

[WK04] J. Warmer and A. Kleppe. *Object Constraint Language 2.0.* MITP Verlag, 2004.

[Woo75] W.A. Woods. What's in a link: Foundations for semantic networks. In D.G. Bobrow and A.M. Collins, editors, *Representation and Understanding: Studies in Cognitive Science*, pages 35–82, 1975.