WP1-D1.1

Report on the Combined Metamodel

Project title:	Marrying Ontology and Software Technology
Project acronym:	MOST
Project number:	ICT-2008-216691
Project instrument:	EU FP7 STREP
Document type:	D (deliverable)
Nature of document:	R (report)
Dissemination level:	PU (public)
Document number:	ICT216691/UoKL/WP1-D1.1/D/PU/a1
Responsible editors:	Fernando Silva Parreiras, Tobias Walter
Reviewers:	BOC, CMR
Contributing participants:	UoKL, BOC, CMR, SAP
Contributing workpackages:	WP1, WP2, WP5, WP6
Contractual date of deliverable:	31 July 2008
Actual submission date:	31 July 2008

Abstract
UML-based models and OWL ontologies constitute modeling approaches with different strengths and weaknesses that make them appropriate for use of specifying different aspects of software systems. Though MOF based metamodels and UML profiles for OWL have been proposed in the past, an integrated approach of both modeling approaches in a coherent framework has been lacking so far. This deliverable presents a framework involving the integration of existing metamodels and profiles for UML and OWL modeling, including relevant (sub)standards such as OCL and considering newer developments such as SWRL, a weaving metamodel and an UML profile for developing integrated models. We start covering the usage scenarios and use cases for the application of ontologies and software technologies. The usage scenarios are scrutinized to identify uses cases. The use cases are detailed and mapped onto the usage scenarios. State of art OWL metamodels and UML-based metamodels are analyzed and reference models are proposed. Subsequently, an integration between both reference models is described. We exemplify a concrete integration with the TwoUse approach.

Keyword List
software modeling, ontologies, metamodeling, integration

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Seventh Framework Programme.

Report on the Combined Metamodel

Fernando Silva Parreiras[1] **and Tobias Walter**[1]

[1] ISWeb — Information Systems and Semantic Web
Institute for Computer Science
University of Koblenz-Landau
Universitaetsstrasse 1
56070 Koblenz, Germany
Email: {`parreiras, walter`}`@uni-koblenz.de`

31 July 2008

Abstract
UML-based models and OWL ontologies constitute modeling approaches with different strengths and weaknesses that make them appropriate for use of specifying different aspects of software systems. Though MOF based metamodels and UML profiles for OWL have been proposed in the past, an integrated approach of both modeling approaches in a coherent framework has been lacking so far. This deliverable presents a framework involving the integration of existing metamodels and profiles for UML and OWL modeling, including relevant (sub)standards such as OCL and considering newer developments such as SWRL, a weaving metamodel and an UML profile for developing integrated models. We start covering the usage scenarios and use cases for the application of ontologies and software technologies. The usage scenarios are scrutinized to identify uses cases. The use cases are detailed and mapped onto the usage scenarios. State of art OWL metamodels and UML-based metamodels are analyzed and reference models are proposed. Subsequently, an integration between both reference models is described. We exemplify a concrete integration with the TwoUse approach.

Keyword List
software modeling, ontologies, metamodeling, integration

Contents

Change Log

Version	Date	Author(s)	Changes
1.3	31.07.08	Fernando Silva Parreiras, Tobias Walter	Chapter 1 to 13 added

1 Introduction

Software modeling languages (e.g. UML2) and ontology modeling languages (e.g. the Web Ontology Language OWL) have different strengths and weaknesses that make them appropriate for specifying different aspects of software systems. UML is a general-purpose modeling language that is capable of capturing information about different views of systems, e.g. static structure and dynamic behavior. Ontologies model domain knowledge and provide shared conceptualizations by a well-defined vocabulary using precise definitions via logical axioms. OWL provides a class definition language for ontologies. In particular, OWL allows for the definition of classes by required and implied logical constraints on the properties of their members.

UML and OWL comprise some constituents which are similar in many respects such as classes, associations, properties, packages, types, generalization and instances. However, both approaches have their advantages and disadvantages. For example, UML provides means to express dynamic behavior, whereas OWL does not. On the other hand, OWL is capable of inferring generalization and specialization between classes as well as class membership of objects based on the constraints imposed on the properties of class definitions. UML class diagrams, on the other hand, do not allow for dynamic specialization/generalization of classes and class memberships.

Future software development should make use of the benefits of both approaches to overcome their restrictions. Advantages of OWL ontologies can support other languages by reducing its model, enabling validation or automated consistency checking. Furthermore ontology languages provide a better support for reasoning than MOF-based languages. These advantages require either to bridge software models from both approaches, or to integrate them into a composed hybrid model.

1.1 Problem

Though UML-Based metamodels and UML profiles for OWL have been proposed in the past, an integrated use of both modeling approaches in a coherent framework has been lacking. To allow developers to take advantage of the benefits of both UML models and ontologies a common metamodel must be designed. This will allow developers to construct software models consisting both of UML classes and ontology concepts, that is, develop hybrid models.

Languages for software models (like UML) and ontologies (like OWL) have different elements and properties and, hence, different metamodels. Since MOST contributes to the smooth integration between the two worlds, this deliverable aims to answer the following questions: *How can we integrate UML-based metamodels and OWL metamodels at the language layer? What are the concrete syntaxes for the integrated models?*

So far, OWL metamodels have only been used in ‘”stand-alone”’ mode of ontology engineering, but a combined metamodel bridges over the two aspects and provides for dual use of classes (e.g. as Java classes and as OWL concepts).

1.2 Objectives

The objective of this deliverable is to extend existing UML-based metamodels as well as OWL metamodels and their respective UML profiles, including relevant (sub)standards such as OCL and considering newer developments such as SWRL (semantic web rule language).

Thus, this deliverable tackles the following issues:

1. What are the syntaxes of the above mentioned languages (UML with its different diagrams, OCL, OWL, SWRL, RIF, etc.)?
2. How can they be represented by metamodeling approaches such as MOF? Which difficulties arise when they are to be integrated into a joint metamodel?
3. What will be a visual concrete syntax (e.g. a UML profile) for such a joint metamodel?

1.3 How to Read This Document

Chapter 2 will give an overview terms and definitions.

Chapter 3 describes the usage scenarios of an integrated approach. Chapter 4 is built on these scenarios and describes the use cases. An analysis of existing OWL metamodels is presented in Chapter 5 as well as a Reference Layer for OWL (Section 5.2). The Reference Layer is a set of abstract classes that are common over different packages and languages and defines the core elements of a given domain.

Different metamodeling approaches and a Reference Layer for different metamodeling approaches like UML, MOF and Ecore are described in Chapter 6.

The MOST Metamodeling Architecture is described in Chapter 7. There, the connections with the Reference Layers and a guideline for extending the metamodel are described. Moreover, the concrete syntax for developing hybrid models including a UML profile and an Weaving Metamodel is explained. A foundation library and guidelines for mappingis are presented.

Finally, in Chapter 8 the metamodels are analyzed and validated according to the requirements and use cases presented in Chapter 3. Chapter 9 finishes the document summarizing the contributions.

2 Terms and Definitions

Abstract Syntax. The abstract syntax delineates the body of concepts and how they may be combined to create models. It comprises definitions of the concepts, the relationships between concepts and well-formedness rules.

Concrete Syntax. The concrete syntax provides the notation to present the constructs defined in the abstract syntax. It can be categorized in textual syntax and visual syntax.

Class-based modeling. Class-based modeling is an approach consisting basically of the essential constructs used to model a UML class diagram that are common on others metamodeling approaches like MOF and Ecore. Examples of these constructs are Class, Property, Operation, Classifier, Attribute.

UML-based Models. UML-based models are models described by metamodels using different architectures derived or based on UML. Examples of UML-based models are MOF models, Ecore models, SysML models or BPMN models.

Metamodel. Metamodel is a model defined on the M2 level.

Metaclass. Metaclass is the class construct on the M2 level according to the the OMG's Four layered metamodel architecture. In fact when describing metamodels, metaclasess are simply referred to as classes.

Model Transformation. Model transformation is a function that receives a source model, a source metamodel, a target metamodel and a transformation script as input and produces a target model conforming with a target metamodel.

Reference Layer. Reference Layer is a set of abstract classes that are common over different packages. It defines the core elements of a given domain.

Implementation Layer. Implementation Layer is the set of classes that extend abstract classes in the Reference Layer by redefining or specifying their properties and operations.

Metamodeling Architecture. Metamodeling Architecture comprises the set of metamodels and packages declared on M2 level, one or more concrete syntaxes to design models conforming with the set of metamodels and mapping rules to accomplish the translation from the concrete syntax to the abstract syntax (metamodels).

UML-based Metamodeling. UML-based metamodeling consists of different metamodels that use constructs as class, property and operation as essential constructs. We use the term UML-based metamodeling to collectively refer to the metamodels UML, MOF and Ecore.

OMG's Four layered metamodel architecture. It is an architecture defined by OMG with four different levels: the metametamodel level (M3), the metamodel level (M2), the model level (M1) and the objects level (M0) (or real world).

Mapping Rules. Mapping rules are relationships among constructs in two distinct metamodels.

Ontology. In this document, the terms ontology and OWL ontology are used interchangeably. For using UML profiles we focus on OWL as language for ontologies.

Scenario. Scenarios are outlines of set of use cases where the integrated metamodel happens to be used. They describe the users' work and are used to extract use cases from it.

2.1 List of Abbreviations

AS	Abstract Syntax
CS	Concrete Syntax
CWA	Closed World Assumption
DL	Description Logics
DSL	Domain Specific Language
M0	Metamodel Level 0
M1	Metamodel Level 1
M2	Metamodel Level 2
M3	Metamodel Level 3
MBOTL	Model-based Ontology Translation Language
MDA	Model Driven Architecture
MM	Metamodel
MOF	Meta-Object Facility 2.0
MOST	Marrying Ontologies and Software Technologies
MTL	Model Transformation Language
NA	Not Available
OCL	UML 2.0 Object Constraint Language
ODM	Ontology Definition Metamodel
OMG	Object Management Group
OWA	Open World Assumption
OWL	Web Ontology Language
OWL 2	Web Ontology Language 2
OWL DL	The Description Logics Dialect of OWL
OWL Full	The Most Expressive Dialect of OWL
QVT	Query / View / Transformation
RDF	Resource Description Framework
RDFS	RDF Schema
RL	Reference Layer
TU	TwoUse
UML	Unified Modeling Language 2.0
WFR	Well Formedness Rules

3 Scenarios

In this section, firstly, we present some scenarios that make profit from the integration of OWL and UML; and describe them mostly with examples. These scenarios lead us to use cases, which we are described in Section 4. These scenarios are merely suggestions of use and are not intended to cover all possible usages of an integrated approach. Most of them are based on preliminary requirements elicitation together with MOST partners.

3.1 Scenario 1: Supporting Model Transformation with Ontologies

Scenario 1 covers the usage of an extended MTL that uses reasoning technologies to achieve more flexible transformation scripts. For example, a sofware developer is writing a model transformation from UML 1.4 Activity Graph into BPEL (UML14toBPEL). As his company adopts UML 2.0, he does not want to regenerate or rewrite the model transformation specifications, when it is possible. He wants to allow metamodel evolution without changing the model transformation.

Example This example uses ontologies for infering axioms that are not aways obvious and make these axioms assertions on models from Metamodeling Technical Space [Silva Parreiras et al., 2007].

Figure 1 illustrates a simplified example: (Step 0: Model) It depicts two models capturing bibliographical references. On the left side, the model `Ma` comprises the class `Publication`, which generalizes `Article`, `MScThesis` and `PhDThesis`. On the right side, the model `Mb` includes the classes `Entry` and `Thesis`. In the middle, there is a mapping model, `Mab`, with a link `MScThesis2Thesis` mapping `MScThesis` onto `Thesis` and a link `PhDThesis2Thesis` mapping a `PhDThesis` onto a `Thesis`.

(Step 1: Forward Transformation) The three models may be transformed into the TRIPLE language [Silva Parreiras et al., 2007]. The resulting TRIPLE model includes all instances and classes of `Ma` and `Mb` as RDF and furthermore it contains the two logical rules depicted in the lower part of Fig.1.

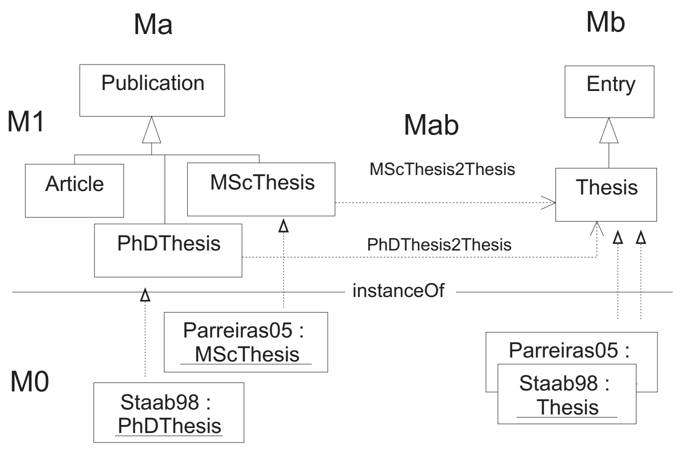
(Step 2: Logical Querying) The logical rules allow for querying of `Ma` instances through the view of `Mb` in TRIPLE [Decker et al., 2005]. The corresponding query is defined in TRIPLE by `FORALL X,Y,Z <- X[Y->Z]@Mb(Ma)`. These retrieved instance triples may be added as triples to the RDF space of the `Mb` part of TRIPLE.

(Step 3: Backward Transformation) Eventually, one may transform the latter results from TRIPLE technical space back to Metamodeling technical space.

While the given example is too simple to be of use in the software engineering process, real applications may exploit the TRIPLE inferencing and enrichment *(i)* to translate (database) objects between PSM/code models at runtime, or *(ii)* to perform more complex reflections (i.e. at the model level) that need the help of logic programming, e.g. recursive logical rules such as exploited in [Oberle et al., 2004].

3.2 Scenario 2: Using ontologies with variability management at runtime

Companies adopt software product line approach for systems engineering. Feature models are used for variability management. Since some variants are up to the user, the decision concerning some variation points is delayed until runtime. Thus the system must be able to validate and



```
// Mapping Mab
FORALL Ma @Mb(Ma) {
  // MScThesis2Thesis
  FORALL X MScThesis[typeOf->X]@Ma --> Thesis[typeOf->X]

  // PhDThesis2Thesis
  FORALL X PhDThesis[typeOf->X]@Ma --> Thesis[typeOf->X]
}
```

Figure 1: Mapping between two models Ma and Mb.

handle configurations at runtime. So this scenario covers the use of ontologies to describe the products and the use of reasoning over the ontologies to specify variability management. Depending of how complex variants are, OCL and OWL could be used to describe them. In OWL such a description takes the form of a class and can be easily reused whereas in OCL it takes the form of invariant stuck on an third class.

Example Today feature models are used to depict variability in software product lines. In this example, taken from a Comarch scenario, entities of a domain model should be mapped to features specified in a feature model (cf. figure 2)[com, 2008]. A domain model is used to specify business entities. These entities are augmented with ontologies, e.g. to state more precisely particular conditions between them. The feature model describes the product line itself. To realize the mapping between these models and later to validate the selection of entities with the feature model an weaving model is used. This weaving model defines the mapping between features of the feature model and business entities of the domain model. This approach led to different advantages. For example during design time the selection of features could be validated. Furthermore accounting for variation points is delayed until runtime.

3.3 Scenario 3: Description of database structures by ontologies

While specifying a part of a domain model, domain experts for example want to define an association between classes as a combination (e.g. union) of other associations or a class as a combination of other classes. Due to heavy performance requirements and large amounts of data that resides already in the database, domain experts are aware that the implementation

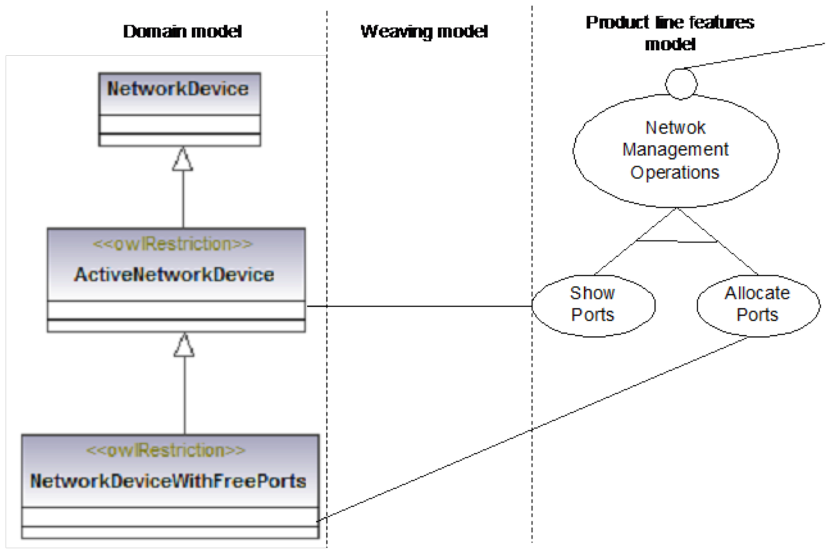


Figure 2: Mapping between feature model and domain model

must be based on database views. This scenario covers the expression of such combinations on conceptual level. Ontologies are used to construct such entities and to automatically generate appropriate database structures from these descriptions. Also ontologies could be helpful to describe a mapping between these entities and the object model. For detailed information we refer to deliverable *WP5-D1: Defnition of the case study requirements* and [com, 2008].

Example Figure 3 depicts a domain model of Comarchs network application OSS. In this model `Edge` has two subclasses - `SimpleEdge` and `ComplexEdge`. `SimpleEdge` can be regarded as point-to-point connection and therefore has two termination points represented in the model by two associations to `Node`: `MasterStartNode` and `MasterEndNode`. `ComplexEdge` in turn may have multiple termination points - each is represented by an `EdgeNodeRelation` which associates a single edge (`MasterEdge`) with a single node (`MasterNode`).

So seen very abstract, networks consist of nodes and edges. These nodes and edges are saved in databases on which queries are executed. Before a node in the network can be deleted the system checks if there are no edges assigned to this node. A query is needed to ask the database if a node fulfils these conditions. In this case a network operator might ask the system about all edges assigned to that node. Considering that there are two types of edges that are assigned to nodes in a different way there must be a way to specify which associations must be used to compute the list of all associated edges.

```
CREATE VIEW dbo.v_node_edge (edge_id, node_id, x_create_date, x_modify_date)
AS
  SELECT ID, start_node_id, x_create_date, x_modify_date
    FROM edge
  UNION
  SELECT ID, end_node_id, x_create_date, x_modify_date
```

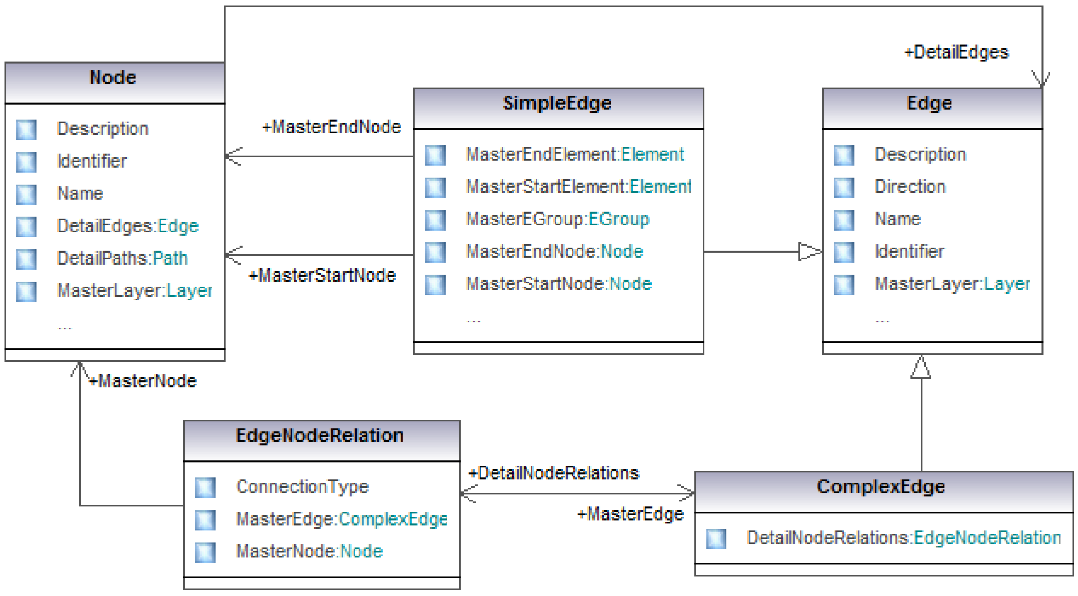


Figure 3: Domain model without usage of OWL

```
  FROM edge
UNION
SELECT edge_id, node_id, x_create_date, x_modify_date
  FROM edge_node_rel
```

For querying a view of an underlying database schema is defined. The role of this view is to specify a mapping of keys of related objects. The view in this example has following definition: In this case a mapping between identifiers of nodes and their assigned edges is specified. An attribute is defined that represents all edges assigned to a node. This attribute can be queried whenever there is a need of listing all edges assigned to a given node. Therefore in current approach the logic that contains specification of all assigned Edges relationship is hidden in PSM - in the specification of a database view.

The goal is that these kinds of logic are specified at the conceptual level for example by ontologies. At runtime these specifications are used and the ontology enriched PIM is transformed into database constructs.

3.4 Scenario 4: Improving design patterns with ontologies

Scenario 4 deals with problems in common design patterns [Gamma et al., 1995] that are used for different problems in software development. In addition to their advantages, [Gamma et al., 1995] also characterized consequences including side effects and disadvantages are caused by their use. For example, solutions based on patterns like Strategy embed the treatment of variants into the client's code at various locations, leading to an unnecessary tight coupling of classes. In this scenario a hybrid approach with ontologies and UML classes come into play to dynamically in-

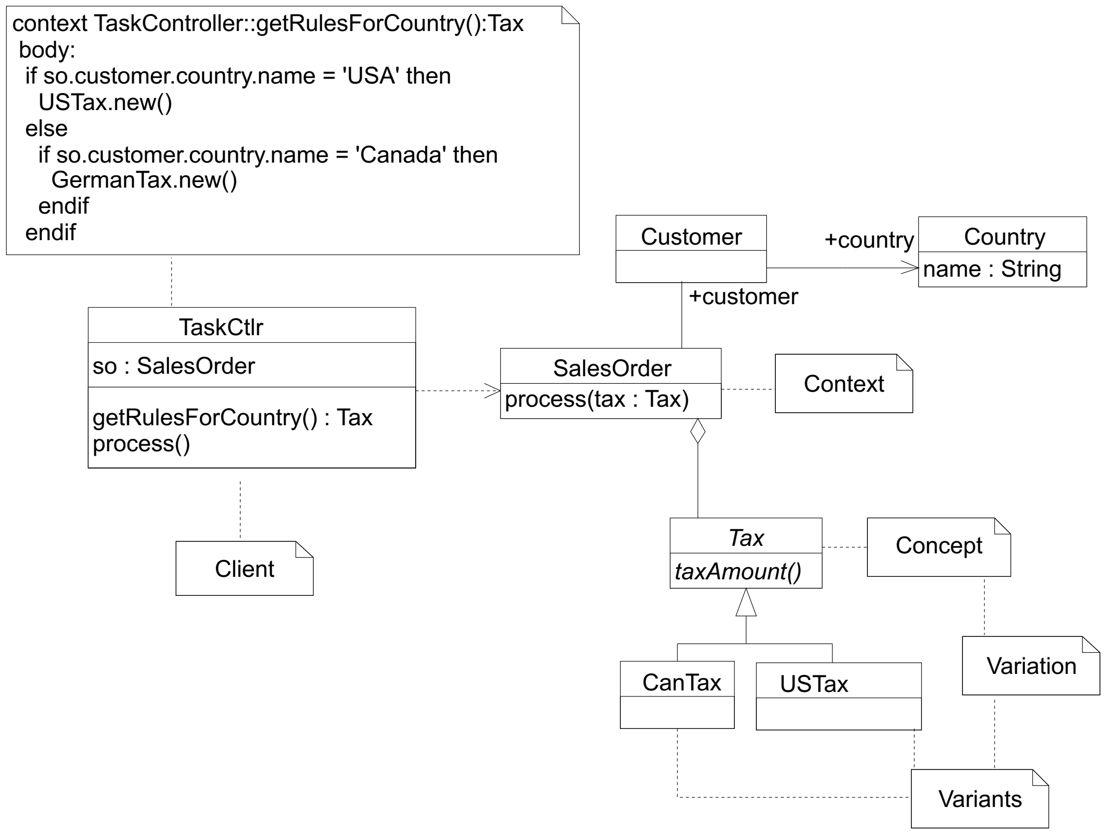


Figure 4: Application of the Strategy Pattern in the problem.

fer class subsumption and object classification. For detailed information we refer to deliverable *WP5-D1: Defnition of the case study requirements.*

Example To illustrate an application of such patterns, we take a well-known example of an order-processing system for an international e-commerce company in the United States [Shalloway and Trott, 2002]. This system must be able to process sales orders in many different countries, like the US and Germany, and handle different tax calculations.

We identify the class `SalesOrder` as *context*, *Tax* as *concept*, and the classes `USTax` and `GermanTax` as *variants* of tax calculation. Since tax calculation varies according to the country, the Strategy Pattern allows for encapsulating the tax calculation, and letting them vary independently of the *context*. The resulting class diagram is depicted in Fig.4.

In general, the Strategy Pattern solves the problem of dealing with variations. However, as already documented by [Gamma et al., 1995], the Strategy Pattern has a drawback. The clients must be aware of variations and of the criteria to select between them at runtime.

When combining the Strategy and the Abstract Factory Pattern, the problem of choosing among the variants of the `AbstractFactory` remains almost the same. Indeed, the Abstract Factory Pattern just groups the families of strategies. Hence, the client must still be aware of

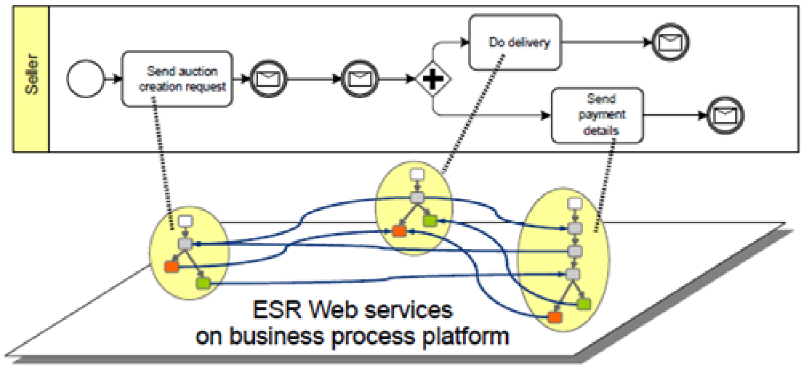


Figure 5: Grounding of NetWeaver BPM on ESR

variations.

Thus, a solution that reuses the understanding of the variations without increasing the complexity is desirable[Silva Parreiras et al., 2008b]. Furthermore, such a solution should allow to decide on the appropriate variants as late as possible. Separating the base of decision from the decision itself will provide an evolvable and more modular software design. In section 3.5 we describe how an OWL-based approach can provide such a mechanism.

3.5 Scenario 5: Web service orchestration specification

For specifying and defining web services orchestration scripts are used. Indeed, a programming language is not applicable since only the business logics are of interest. Thus, this scenario deals with a language which is able to capture business logics of web service orchestration and capable to use reasoning capabilities over web service descriptions.

Example SAP uses a tool called NetWeaver BPM to create business process models. From these models at business layer the grounding of web services are specified (cf. figure 5). The user should be supported by a language that gives some guidance for example to identify which web service operations are potentially needed to implement a certain process task (discovery). Also orchestrating different behavioral requirements of interacting web services should be supported (orchestration). The identifying and consistently resolving alternative, undesired operation responses among all participating web services (exception management) should also be a feature of this language.

3.6 Scenario 6: Ontology based systems

We use our case study in the context of semantic multimedia tool as practical running example in this paper. The K-Space Annotation Tool (KAT)[1] is a framework for semi-automatic and efficient annotation of multimedia content providing the following features:

- GUI framework;

[1]`http://isweb.uni-koblenz.de/Research/kat`

- Plug-in Infrastructure (analysis plug-ins and visual plug-ins);
- Formal model based on the Core Ontology for Multimedia (COMM)[Arndt et al., 2007];
- Support to semantic file system SemFS [2].

Analysis plug-ins provide functionalities to analyze content, e.g., to semi-automatically annotate multimedia data like images or videos, or to detect structure within multimedia data. However, as the number of available plug-ins increases, it becomes confusing for users to choose appropriate plug-ins to perform over multimedia data.

For example, K-Space EU project partners[3] provide Machine Learning based classifying, e.g., Support Vector Machines (SVM) for pattern recognition. There are different recognizers (object recognizers, face detectors, speaker identifiers) for different themes (sport, politics, art), for different types of multimedia data (image, video) and for different formats (JPEG, GIF, MPEG, etc.). Moreover the list of recognizers is continuously extended and, like the list of multimedia formats, it is not closed, but by sheer principle it needs to be open. Hence, end-users may easily misunderstand the suitability of recognizers to multimedia data.

Instead of hard-embedding class descriptions using OCL expressions, a more expressive and extensible manner of modeling data is more appropriate. Actually, we request flexible ways to describe classes and, based on such descriptions, we want to infer typing.

Therefore, one requires a logical class definition language that is more expressive than UML2 class-based modeling. Among ontology languages, the Web Ontology Language (OWL) [McGuinness and van Harmelen, 2004] is the most prominent for Semantic Web applications.

[2]`http://isweb.uni-koblenz.de/Research/SemFS`
[3]`http://www.k-space.eu/`

4 Use Cases

The purpose of this deliverable is the definition of a combined metamodel for UML and OWL. Since the form and content of this metamodel heavily depends on its purpose, we first assemble the typical use cases for this metamodel. They are derived from the scenarios presented in section 3.

The use case diagram depicted in figure 6 gives an overview about all use cases and their relations. It also gives an overview of both actors and their relation to the use cases. The architect works all most on the M2-layer and is responsible for extending the metamodel of some application. The designer who all most works on the M1-layer interacts with designing new class diagrams, verifying and transforming them.

We separate the use cases by using two different use case packages. Use case package Design contains all use cases that deal with designing models mostly at the M1-layer. Use case package Extension contains these use cases that deal with extending the metamodel.

After describing the use cases in the following sections, we map these use cases onto scenarios in Table 1.

4.1 Use Case Package 1: Design

This section defines all use cases belonging to use case package *Design*.

4.1.1 Use Case 1.1: Write Model Transformation with Ontology Support

- **Actor:** Designer
- **Use Case Description:** This use case covers the support to model transformations, which are supported by ontologies, between different instances (models) of it.
- **Input:** A source metamodel, a source model conforming with the source metamodel, a correspondent reference ontology, a target metamodel.
- **Output:** Target model generated.
- **Normal course of events:**
    1. The developer writes the transformation script using ontological rules.
- **Exceptions**: None

4.1.2 Use Case 1.2: Design Hybrid Class Diagram

- **Actor:** Designer
- **Use Case Description:** This use case covers the creation and visualization of hybrid models of different origin, like UML or OWL and with a concrete syntax to depict the model in one diagram.
- **Input:** Two or more diagrams, UML profiles for MOST
- **Output:** One hybrid diagram
- **Normal course of events:**

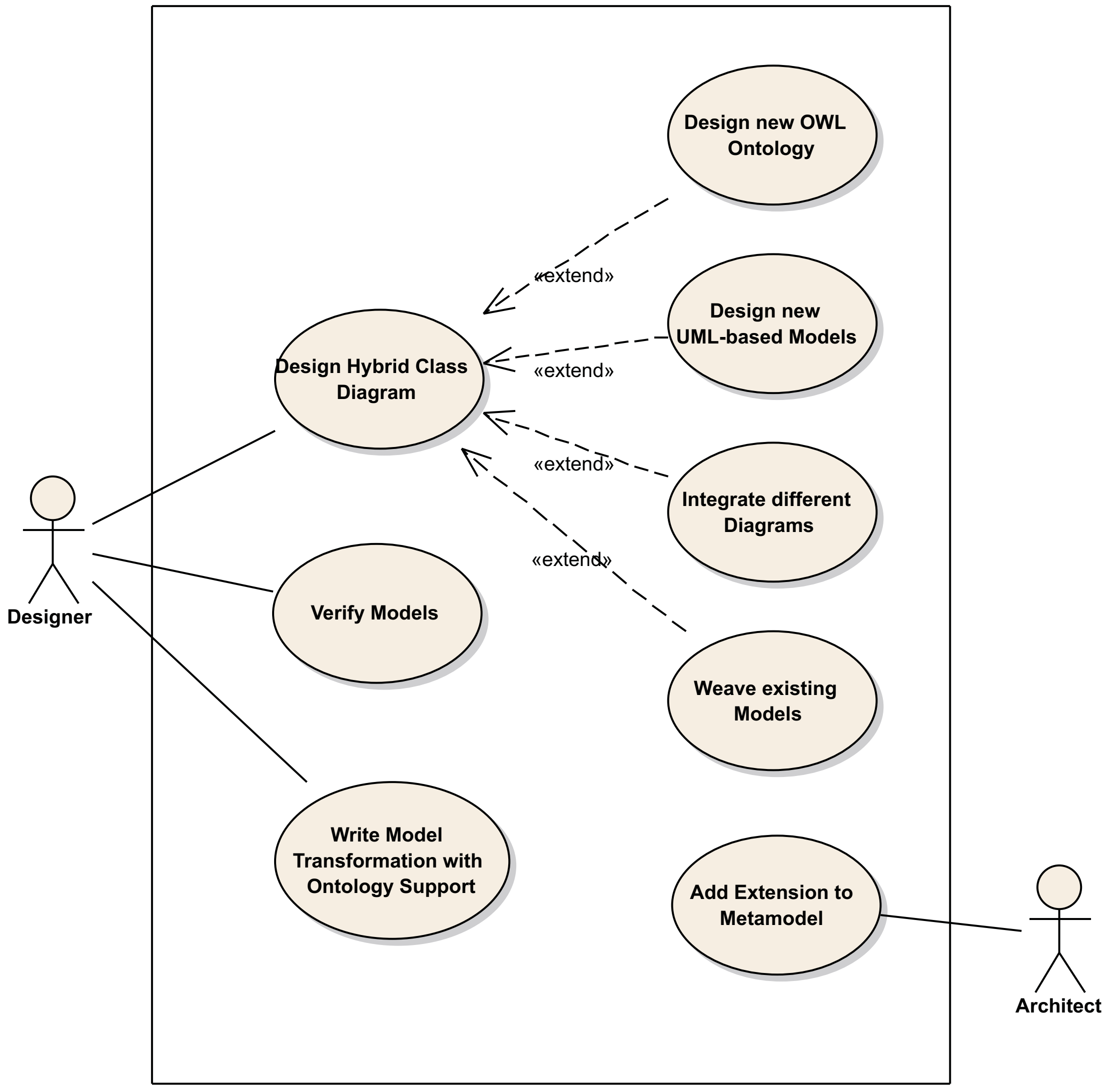

Design new OWL Ontology
«extend»
Design new UML-based Models
«extend»
Design Hybrid Class Diagram
«extend»
Integrate different Diagrams
«extend»
Designer
Verify Models
Weave existing Models
Write Model Transformation with Ontology Support
Add Extension to Metamodel
Architect

Figure 6: Use case model

1. The designer models the hybrid class diagram using the proper profile.

- **Exceptions:** None

4.1.3 Use Case 1.3: Design new OWL ontology

- **Actor:** Designer
- **Use Case Description:** This use case covers the design of ontologies. On the one hand ontologies should be created by any visual editor, on the other hand alternatively by textual syntaxes (like Manchester OWL Syntax).
- **Input:** none
- **Output:** New OWL ontology
- **Alternate courses:**
  1. visual: The designer uses UML profiles to model the OWL ontology.
  2. textual: The designer uses Manchester OWL Syntax to model the OWL ontology
- **Exceptions:** none

4.1.4 Use Case 1.4: Design new UML-based models

- **Actor:** Designer
- **Use Case Description:** This use case covers creation of UML-based models by using UML visual editors.
- **Input:** none
- **Output:** New UML-based model
- **Normal course of events:**
  1. The designer draws UML elements.
- **Exceptions:** none

4.1.5 Use Case 1.5: Verify Hybrid Models

- **Actor:** Designer
- **Use Case Description:** This use case covers the verification of models. First constraint enforcement is claimed, so that model checking on the models can be executed.
- **Input:** Model enriched by constraints
- **Output:** Verified model
- **Normal course of events:**
  1. The diagram is parsed into hybrid model
  2. The hybrid model is validated against well formed rules
- **Exceptions:** none

4.1.6 Use Case 1.6: Integrate OWL into different Diagrams

- **Actor:** Designer
- **Use Case Description:** This use case covers the integration of different diagrams for which integrated syntaxes are needed.
- **Input:** Given different diagrams with different syntaxes
- **Output:** Diagram with OWL support
- **Normal course of events:**
  1. Designer associates OWL classes and restrictions with diagram elements like classes, activities, constraints, etc.
- **Exceptions:** none

4.1.7 Use Case 1.7: Weave existing OWL Ontology and UML-based Models

- **Actor:** Designer
- **Use Case Description:** This use case covers the integration of models. In a particular case OWL ontologies and UML-based models should be weaved to one integrated model.
- **Input:** Given OWL ontology and UML-based models
- **Output:** Weaving model
- **Normal course of events:**
  1. The Designer links OWL elements and UML elements by means of a weaving model.
- **Exceptions:** none

4.2 Use Case Package 2: Extension

This section defines all use cases belonging to use case package *Extension*.

4.2.1 Use Case 2.1: Add Extension to Metamodel

- **Actor:** Architect
- **Use Case Description:** This use case covers the extension of metamodels e.g. the ability to plug additional metamodels in the proposed framework, since the framework is expected to serve different languages like model transformation languages, process modeling languages, etc.
- **Input:** Given metamodel
- **Output:** Extended metamodel
- **Normal course of events:**
  1. The architect adds a new metamodel into the metamodel architecture
- **Exceptions:** Violation of well formedness rules.

Scenarios	Sec. 3.1	Sec. 3.2	Sec. 3.3	Sec. 3.4	Sec. 3.5	Sec. 3.6
Use Cases						
Sec. 4.1.1	X		X		X	
Sec. 4.1.2		X	X	X		
Sec. 4.1.3	X	X	X	X	X	X
Sec. 4.1.4		X	X	X	X	X
Sec. 4.1.5		X	X			
Sec. 4.1.6	X	X	X	X	X	X
Sec. 4.1.7	X	X	X	X	X	X
Sec. 4.2.1	X	X	X	X	X	X

Table 1: Traceablity Matrix: Mapping Use Cases to Scenarios

4.3 Mapping Use Cases to Scenarios

After describing the use cases in section 4, we mapped these use cases to scenarios in Table 1. The table identifies which scenario is instance of a particular use case and gives advice where a sequence of steps through a use cases could be found.

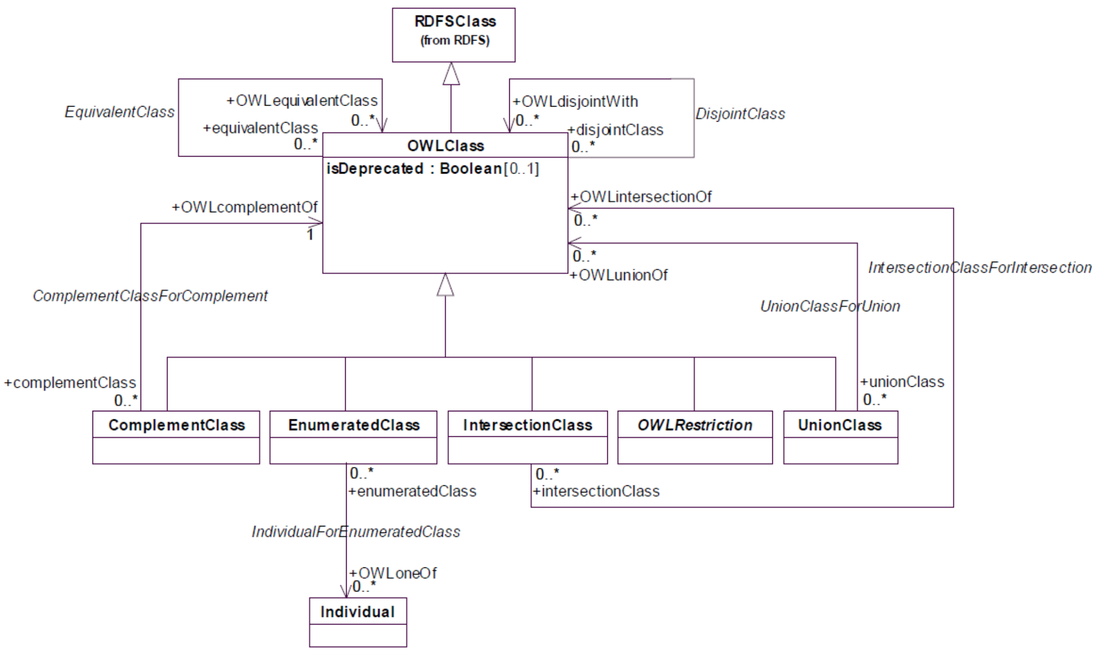


Figure 7: The OWL Class Descriptions Diagram of the OMG OWL Metamodel[OMG, 2007a]

5 OWL Metamodels

The following section presents a short description of the three most prominent OWL metamodels, namely the *OMG OWL Metamodel* [OMG, 2007a], the *Neon OWL Metamodel* [Brockmans et al., 2004] and the *OWL2 Metamodel* [Motik et al., 2008].

Our aim is not to describe all those metamodels. Instead, we concentrate on two central constructs easily comparable and understandable: classes and properties. Please refer to the citations for more details.

5.1 State of the Art

5.1.1 OMG OWL Metamodel

The *OMG OWL Metamodel* is part of the *Ontology Definition Metamodel OMG Adopted Specification*. It has a high number of classes, since it imports the *OMG RDFS Metamodel*. Thus some relations between classes are described in the RDFS Metamodel and reused in the OWL Metamodel.

For example, Figure 7 and Figure 8 depict the Class Description Diagram and the Properties Diagram respectively. The Domain and range of properties are specified in the RDFS Metamodel, depicted in Fig.9.

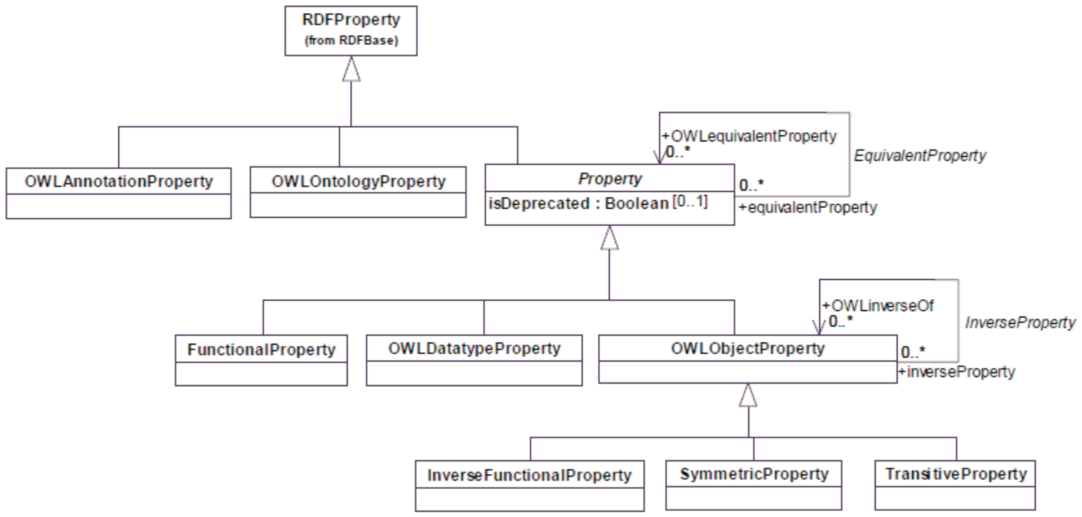


Figure 8: The OWL Properties Diagram of the OMG OWL Metamodel [OMG, 2007a]

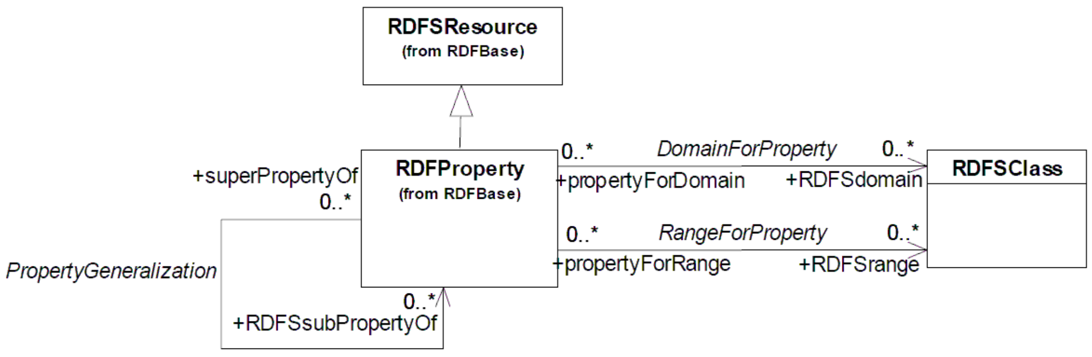


Figure 9: RDFS Package, The Properties Diagram of the OMG OWL Metamodel [OMG, 2007a]

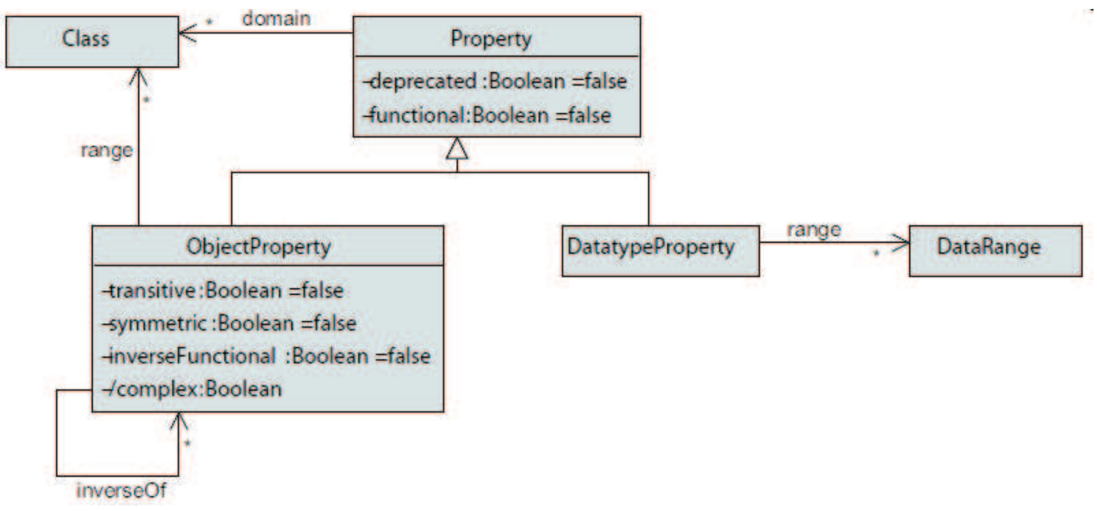

Class
domain
*
Property
-deprecated :Boolean =false
-functional:Boolean =false
range
*
ObjectProperty
-transitive:Boolean =false
-symmetric :Boolean =false
-inverseFunctional :Boolean =false
-/complex:Boolean
*
inverseOf
DatatypeProperty
range
*
DataRange

Figure 10: The OWL Class Descriptions Diagram of the NEON Metamodel

5.1.2 NEON OWL Metamodel

The *Neon Metamodel* [Brockmans et al., 2004] is a concise metamodel able to cover the OWL-DL functional syntax. Figure 10 and Figure 11 depict the OWL class hierarchy and the property diagram respectively. The relationship between Class and Property is direct, since the Neon OWL Metamodel does not provide support for RDFS.

5.1.3 OWL2 Metamodel

Improvements on the OWL language led W3C OWL Working Group to publish working drafts of a new version of OWL: *OWL 2*[Motik et al., 2008]. OWL 2 is fully compatible with OWL-DL and extends the latter with limited complex role inclusion axioms, reflexivity and irreflexivity, role disjointness and qualified cardinality restrictions. Moreover, OWL 2 uses a new XML Serialization and provides a set of profiles with different levels of expressiveness.

As one may note, the OWL 2 Metamodel is considerably different from the available metamodels for OWL. Constructs like Axiom and OWLEntity play central roles and associations between classes and properties are done by axioms. Figures 12 and 13 exemplify such constructs.

5.2 MOST Reference Layer for OWL

The objective of adopting reference layers is to provide extensibility and compatibility with existing metamodels. By doing so, we provide the flexility of choosing the metamodel that best meets partners' requirements. Moreover, it gives an extension point for introducing new metamodels without affect existing applications.

We have conducted a qualitative comparison between the three metamodels described above to select the metamodel with better extensibility and effectiveness quality attributes.

The Neon OWL Metamodel was chosen, because it is the smallest one on number of classes and the simplest one, since it is not attached to the RDF metamodel, as the OMG metamodel.

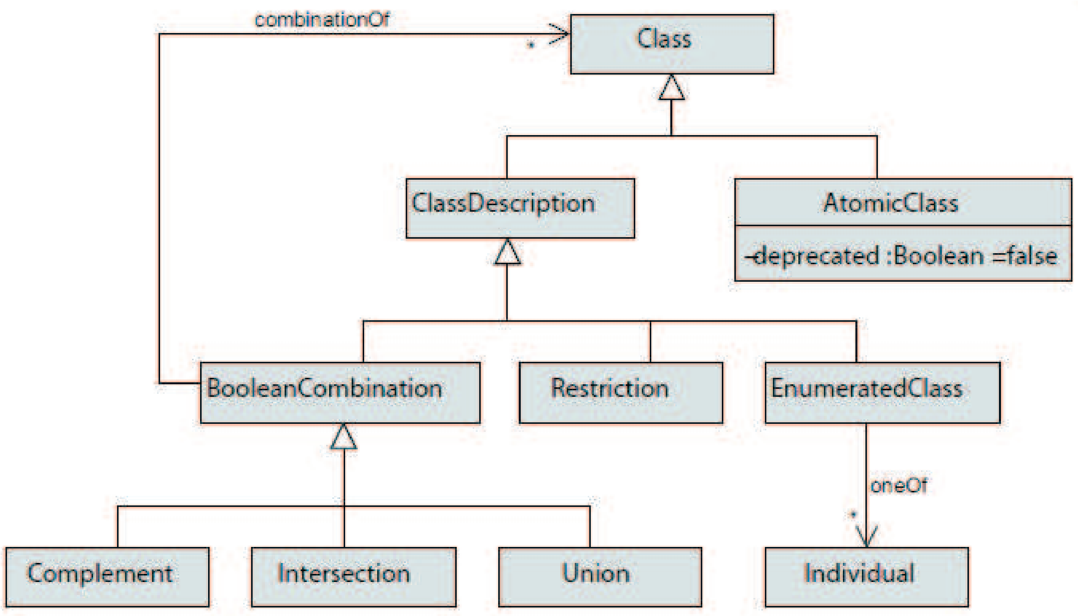


Figure 11: The OWL Properties Diagram of the NEON Metamodel

The OMG metamodel has different qualities like public acceptance as standard and popularity. Nevertheless, working with the OMG metamodel as reference metamodel would introduce unnecessarily the complexity of dealing with RDF without any gain. Furthermore, the usage of the OMG metamodel is enabled as concrete metamodel as explained in Sect. 7.2.2.

OWL 2 is backwards compatible with OWL1 constructs, but, by the time of writing this document, the OWL2 metamodel was not finished yet although it was stable. Notice that the Neon metamodel does not cover OWL 2 constructs and, consequently, neither the MOST reference metamodel for OWL. However it would be possible to extent the reference metamodel to support OWL 2 if such a need emerges as use case requirement. Such changes will be part of the foreseen revision of the combined metamodel.

Because of differences between OMG OWL, Neon OWL and OWL 2, we introduce a Reference Layer that comprises common concepts of OWL that are used by MOST. Based on the extensibility metrics presented by the Neon OWL metamodel, we have adapted it to be used as Reference Layer.

5.2.1 Reference Layer

The *OWL Reference Layer* to be used in MOST is an adaptation of the Neon OWL metamodel. This Reference Layer works as an interface for MOST to access OWL metamodels. In the following paragraphs we detail each package.

OWL::OWLBasic. Figure 14 depicts the class diagram of the package OWL::OWLBasic. Notice that the MOST OWL Reference Layer must be used together with a concrete OWL metamodel.

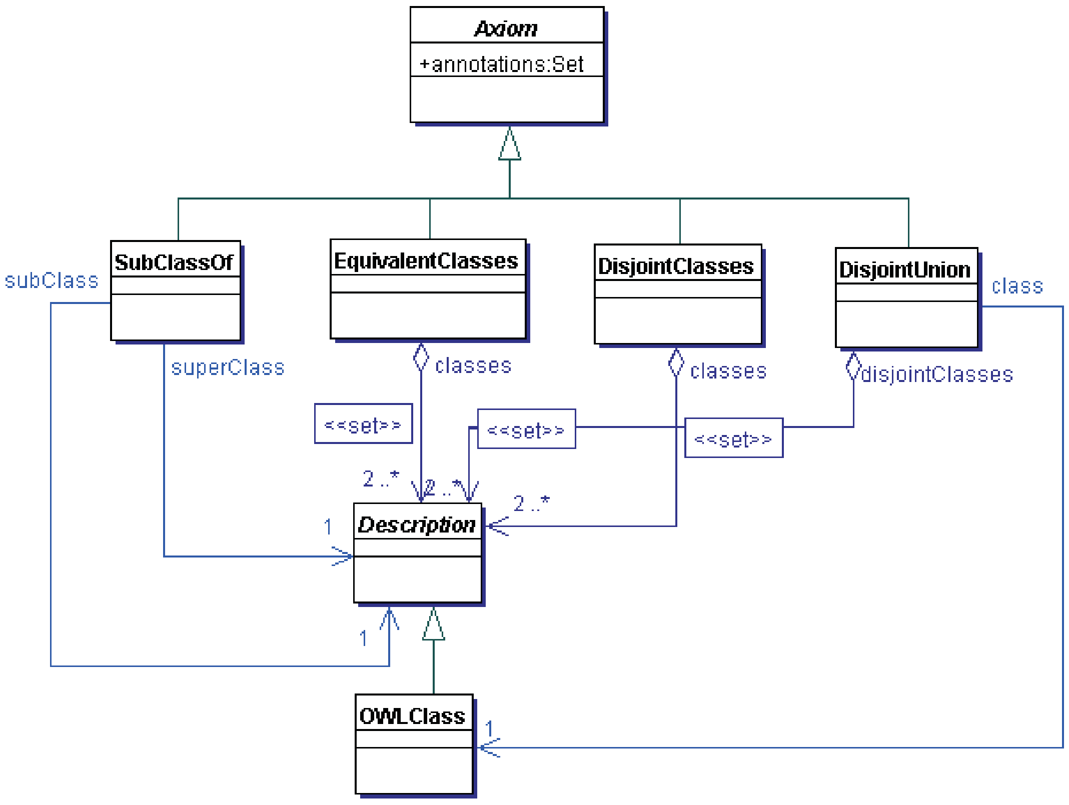


Figure 12: The OWL Class Descriptions Diagram of the OWL 2 Metamodel

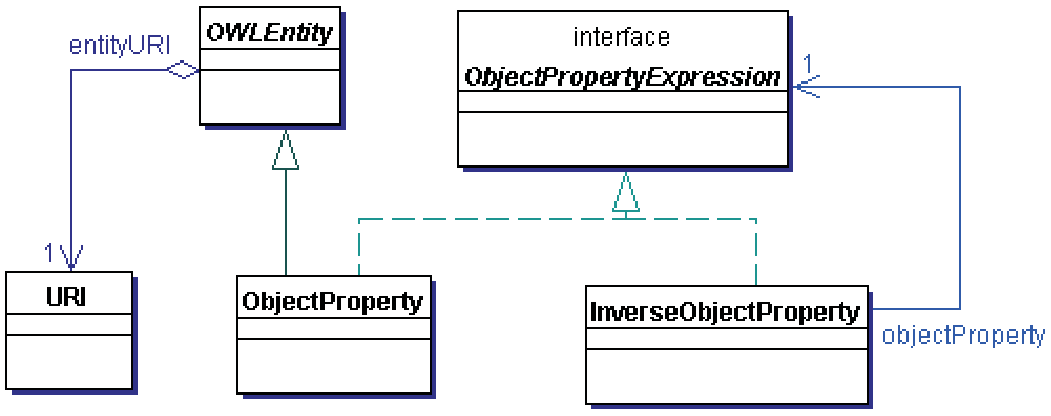


Figure 13: The OWL Properties Diagram of the OWL 2 Metamodel

Whereas the OMG OWL, Neon OWL and OWL 2 metamodels comprise similar classes, associations between classes in these models are specified differently. As well-formedness rules of the MOST metamodel need to navigate along such associations, the Reference Layer introduces additional operations that encapsulate internal navigations.

For example, supposing one wants to adopt MOST with the OMG OWL Metamodel as OWL metamodel, it is required to adapt the classes from the MOST OWL Reference Layer. Figure 15 depicts the example of the classes Class and Property.

Specializing Abstract Classes. Firstly, classes with very similar semantics are chosen and a specialization association is drawn from the abstract class in the MOST OWL Reference Layer to the concrete class.

Implementing Abstract Operations. In the MOST OWL Reference Layer, the class Property has the operation getDomain of type collection of Class. According to our example, the class DatatypeProperty and the class ObjectProperty have the operation getRange with a collection of either Datatype or Class. The class Property inherits from both RDFProperty and BasicOWL::Property.

The class Property specifies the operation getDomain according to the internal associations in the OMG OWL metamodel. Thus, the definition of getDomain looks as follows:

```
context Property
def: getDomain(): Set(BasicOWL::Class)
    self.RDFSDomain.oclAsType(Class)->asSet()
```

The definition of the operation getRange for the DatatypeProperty and the ObjectProperty is similar:

```
context OWLDatatypeProperty
def: getRange(): Set(BasicOWL::Datatype)
    self.RDFSDomain.oclAsType(Datatype)->asSet()

context OWLObjectProperty
def: getRange(): Set(BasicOWL::Class)
    self.RDFSDomain.oclAsType(Class)->asSet()
```

The same process as described above is then used for extending the remaining classes and defining their methods.

SWRL Since the SRWL metamodel is compatible with the Neon OWL metamodel, it can be used together with the MOST OWL Reference Layer to provide support to OWL Rules. For more details about the SWRL metamodel please see [Brockmans et al., 2006]. Our work involves only converting associations between classes in operations(Fig. 16).

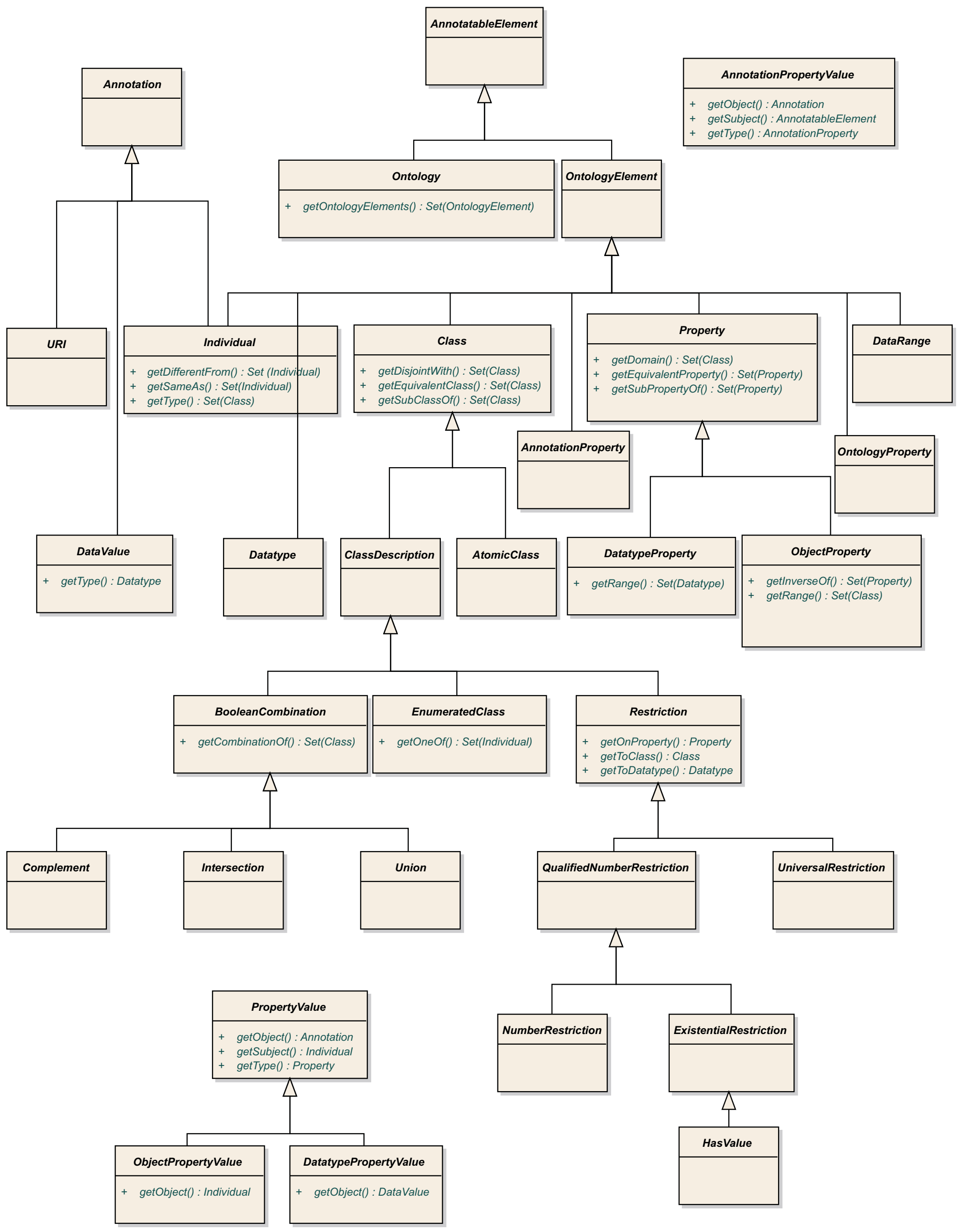

AnnotatableElement
AnnotationPropertyValue
+ getObject() : Annotation
+ getSubject() : AnnotatableElement
+ getType() : AnnotationProperty
Annotation
Ontology
+ getOntologyElements() : Set(OntologyElement)
OntologyElement
URI
Individual
+ getDifferentFrom() : Set (Individual)
+ getSameAs() : Set(Individual)
+ getType() : Set(Class)
Class
+ getDisjointWith() : Set(Class)
+ getEquivalentClass() : Set(Class)
+ getSubClassOf() : Set(Class)
Property
+ getDomain() : Set(Class)
+ getEquivalentProperty() : Set(Property)
+ getSubPropertyOf() : Set(Property)
DataRange
AnnotationProperty
OntologyProperty
DataValue
+ getType() : Datatype
Datatype
ClassDescription
AtomicClass
DatatypeProperty
+ getRange() : Set(Datatype)
ObjectProperty
+ getInverseOf() : Set(Property)
+ getRange() : Set(Class)
BooleanCombination
+ getCombinationOf() : Set(Class)
EnumeratedClass
+ getOneOf() : Set(Individual)
Restriction
+ getOnProperty() : Property
+ getToClass() : Class
+ getToDatatype() : Datatype
Complement
Intersection
Union
QualifiedNumberRestriction
UniversalRestriction
PropertyValue
+ getObject() : Annotation
+ getSubject() : Individual
+ getType() : Property
NumberRestriction
ExistentialRestriction
HasValue
ObjectPropertyValue
+ getObject() : Individual
DatatypePropertyValue
+ getObject() : DataValue

Figure 14: MOST OWL Reference Layer - Package OWL::BasicOWL.

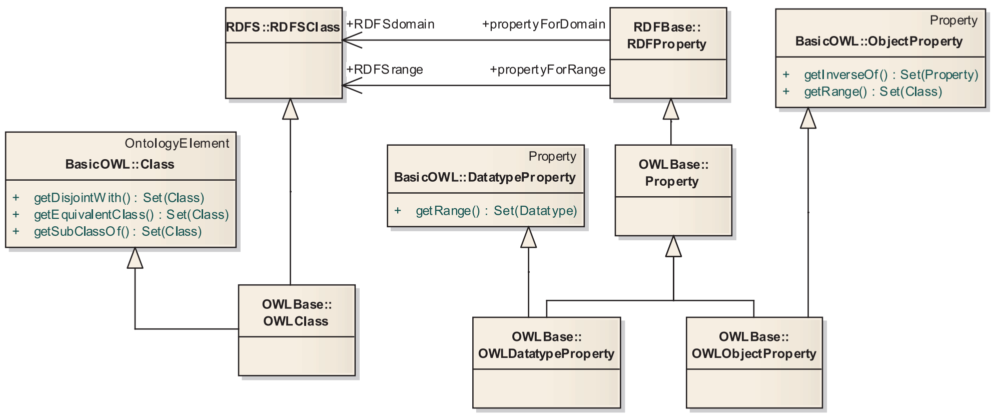


Figure 15: Adapting the MOST OWL Reference Layer to OMG OWL metamodel.

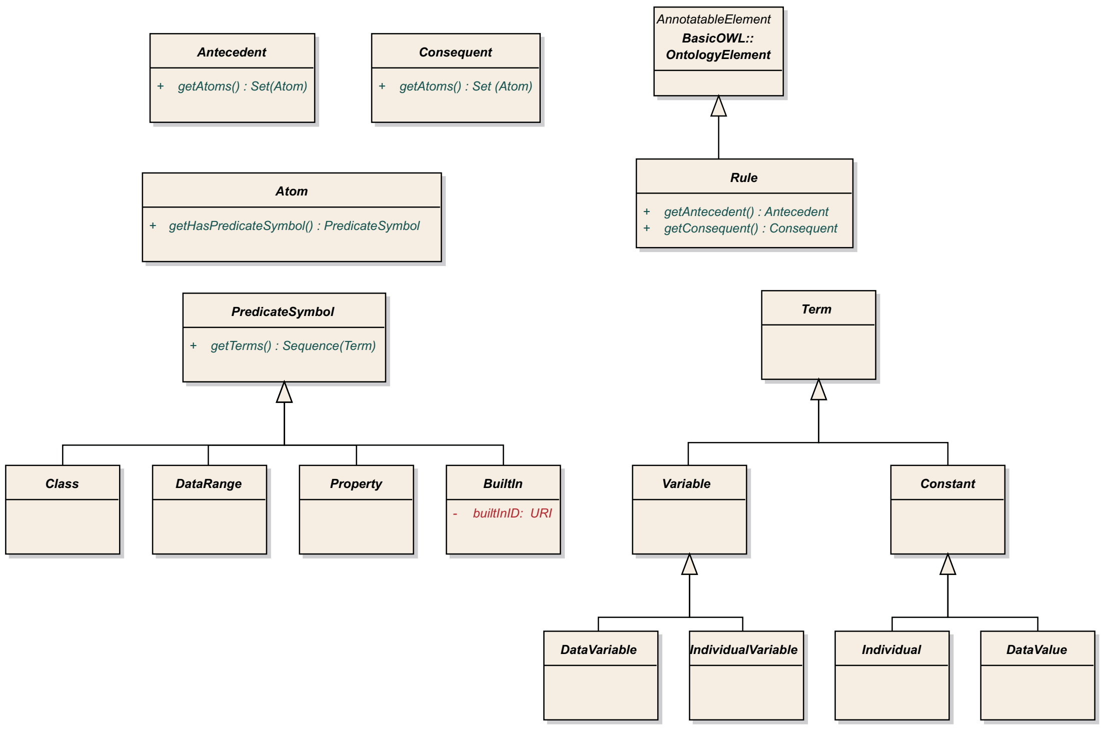


Figure 16: SWRL Package.

6 UML-Based Common Metamodels

Following the rationale of UML Class modeling, several Metamodels are centered in constructs like Class, Property and Attribute. Examples of such metamodels are Ecore, MOF, KM3 and UML class modeling packages. Additionally, metamodels that import or depend on the former metamodels like QVT, ATL and SysML are also organized around these constructs as well as Business Process languages like BPEL or BPMN.

Therefore it is feasible to identify common constructs among the above metamodels and group them forming an reference layer. This section explains such a strategy.

6.1 MOST Reference Layer for UML-based Metamodeling

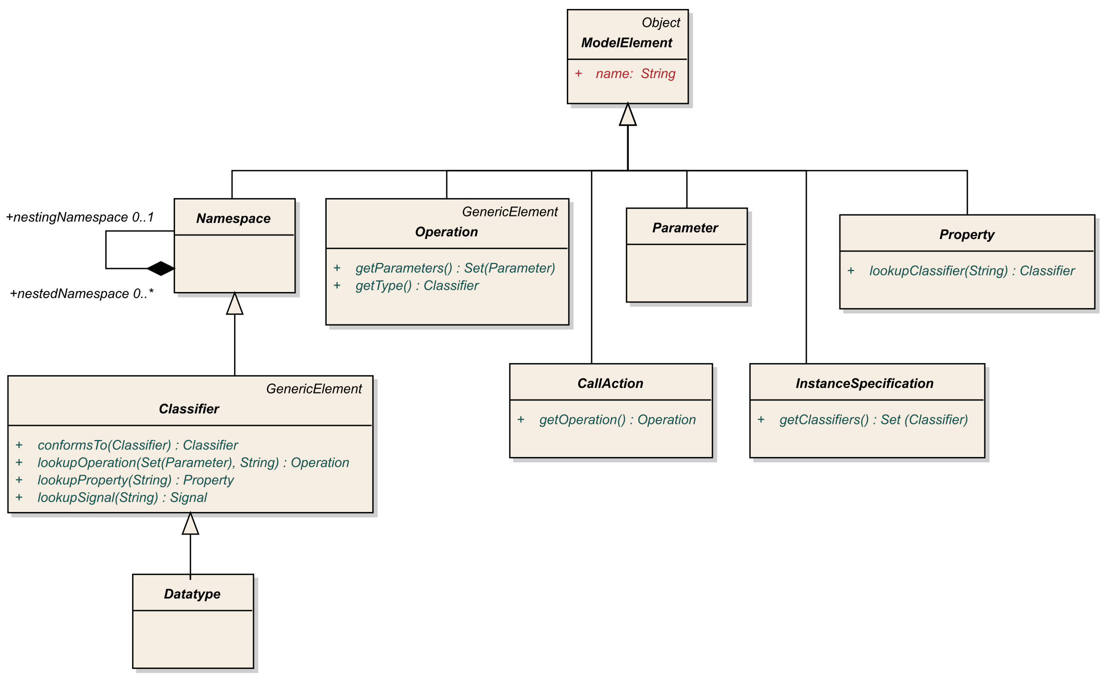
Following the rationale of MOST OWL Reference Layer, we provide a Reference Layer for UML-based Metamodeling as depicted in Fig. 17. Under UML-based metamodeling we group modeling approaches that use constructs as class, property and operation as essential constructs. The MOST UML-based Metamodeling Reference Layer comprises the Kernel package with common constructs to UML, MOF and EMOF.

Kernel. The Kernel package comprises common core classes to UML, MOF and Ecore and necessary classes to possibly support OCL. Common constructs among these UML-based approaches have been already investigated by [Akehurst and Patrascoiu, 2004], [Loecher and Ocke, 2004], [Bräuer and Demuth, 2007]. We have adapted the work of Akehurst and Patrascoiu [Akehurst and Patrascoiu, 2004] to provide the Reference Layer for UML-based Metamodeling.

Akehurst and Patrascoiu [Akehurst and Patrascoiu, 2004] have successfully mapped most of the classes of the Kernel package to classes from the UML 1.X (which the metamodel is originally based on), to the metamodel for Java and to the ECore Metamodel. According to their work, it would not be a problem mapping the classes to the UML 2.0 metamodel or MOF metamodels. One might think of using UML 2.0 directly, but it would introduce unnecessary complexity regarding the size and hierarchies.

The class ModelElement allows to tag model elements with names. Namespace is a ModelElement that can own other ModelElements, like Classifiers or Packages. An Operation is a functioning objects request to effect behavior. An operation is typed by a Classifier and has a signature describing actual Parameters.

A CallAction is "an action resulting in an invocation of an operation on an instance" [OMG, 2001]. An action points to the operation that is invoked when the Action is carried out. A Property designates attributes for containing classifiers. Finally, An InstanceSpecification is a model element that specifies existence of an instance. It can be classified by Classifiers.



Object
ModelElement
+ name: String
+nestingNamespace 0..1
Namespace
+nestedNamespace 0..*
GenericElement
Operation
+ getParameters() : Set(Parameter)
+ getType() : Classifier
Parameter
Property
+ lookupClassifier(String) : Classifier
GenericElement
Classifier
+ conformsTo(Classifier) : Classifier
+ lookupOperation(Set(Parameter), String) : Operation
+ lookupProperty(String) : Property
+ lookupSignal(String) : Signal
CallAction
+ getOperation() : Operation
InstanceSpecification
+ getClassifiers() : Set (Classifier)
Datatype

Figure 17: MOST Reference Layer for UML-Based Metamodeling.

7 MOST Metamodeling Architecture

In spite of their different purposes, Ontologies and UML-based metamodeling share some similar constructs. Recent work presents similarities between MOF and RDF [Gašević et al., 2007], between OWL/RDF and Object-Oriented Languages [Knublauch et al., 2006] and between UML and OWL [OMG, 2007a] [Falkovych et al., 2003]. We roughly summarize them in Table 2. For the subtleties, please refer to the cited papers.

Some of these constructs are based on others. For example, subclass, enumeration and disjointness are based on Classifier; domain and range are based on the relationship between classifier and property whereas cardinality is based on Property.

Thus, providing a seamless integration of the core constructs is the foundation for an integrated approach. The next sections present an extensible approach for integrating OWL ontologies and UML-based metamodeling.

7.1 MOST Conceptual Architecture

With the intention of summarizing and providing a complete view of MOST Conceptual Architecture, Fig. 18 presents a model-driven view of MOST, using modeling spaces [Djurić et al., 2005a]. Figure 18 shows the Concrete Syntax and Abstract Syntax according to the MOF modeling space. Two modeling levels according to the OMG's Four layered metamodel architecture are shown: the metamodel level (M2) and the model level (M1). The relationships inside each quadrant show dependency, the ones stereotyped with flow are transformations, the ones that cross the horizontal borderline denote instantiation.

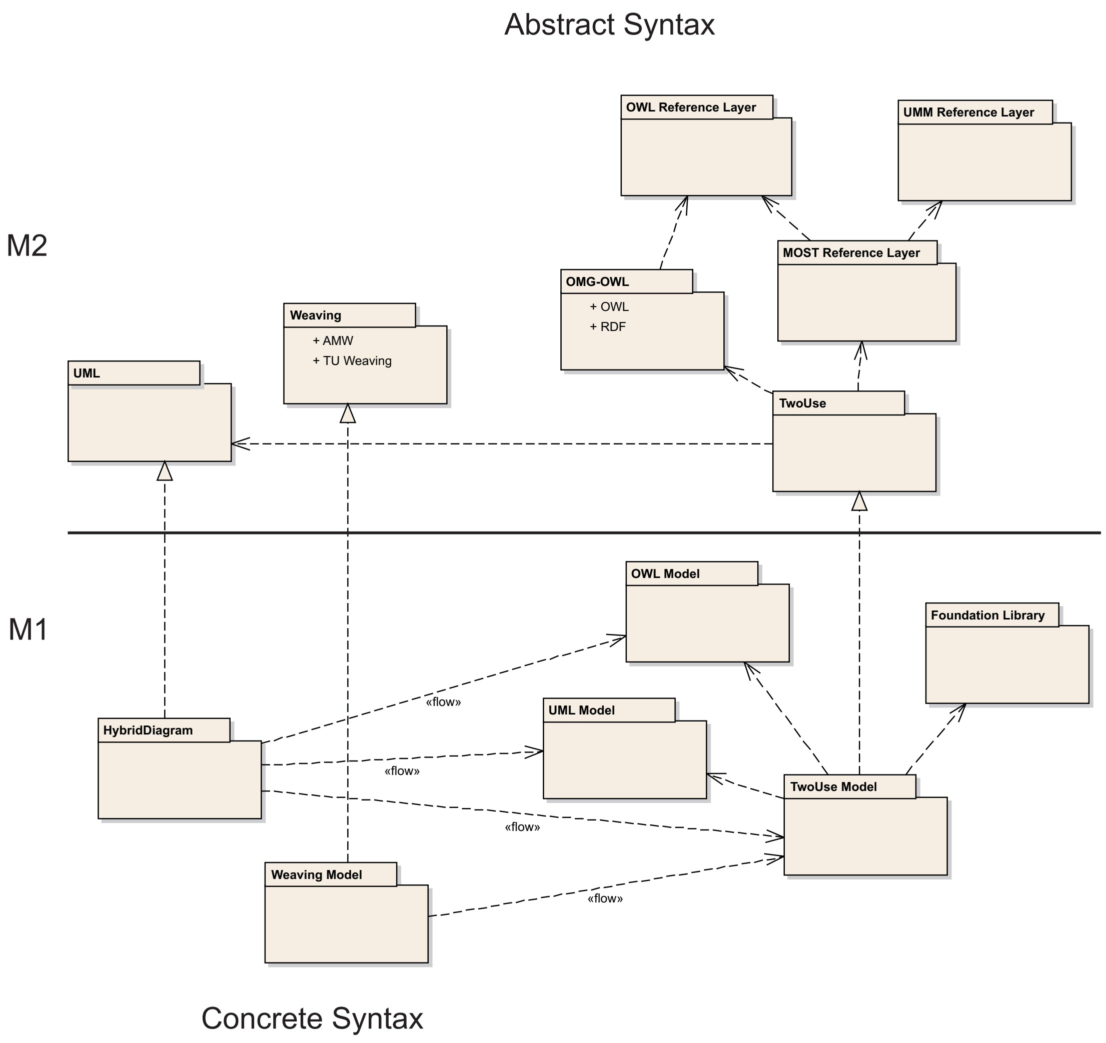


Figure 18: Layered metamodels and modeling spaces of the TwoUse Approach.

OWL	UML Class-based Modeling
Ontology	Package
Class	Class, Classifier
Individual and values	Instance and Attribute Values
Object Property, Datatype Property	Association, Attribute
Data Types	Data Types
Subclass, Subproperty	Subclass, Generalization
Enumeration	Enumeration
Domain, Range	Navigable, Non-Navigable
Disjointness, union	Disjointness, cover
Cardinality	Multiplicity

Table 2: Comparable Features of OWL and UML Class-based Modeling

A concrete implementation of the MOST Reference Layer is illustrated here by TwoUse [Silva Parreiras et al.,]. Indeed, TwoUse represents our initial efforts at integrating OWL and UML. In the context of MOST, TwoUse profits from the MOST Reference Layer.

As Concrete Syntax UML profiles of the HybridDiagram are illustrated: an UML profile for OWL (Sect. 7.3.1) and another for MOST, which contains only stereotypes for Opaque expressions and properties (Sect. 7.3.1).

The resulting package from designing a hybrid model using an UML tool are shown at M1. The hybrid diagram is composed of: pure UML elements with their OCL expressions; elements stereotyped by the UML profile for OWL; and the opaque expressions stereotyped by the UML profile for MOST.

The Abstract Syntax presents the organization of the packages in the MOST. It comprises the MOST OWL Reference Layer (Sect. 5.2), the UML-based metamodeling Reference Layer (Sect. 6.1), the MOST Reference Layer (Sect.7.2.1), the implementation of the MOST Reference Layer by integrating of OMG-OWL and UML provided - TwoUse. The *TwoUse package* imports the UML and OWL metamodels and specializes elements from both, as shown in Sect. 7.2.2.

The packages of models generated from the hybrid diagram are represented at M1. UML elements are copied, because they don't need to be transformed. The package `HybridDiagram` serves as basis for two transformations: (1) the generation of the OWL model, instance of OWL metamodel and (2) the generation of TwoUse classes in the TwoUse package, as explained in Sect. 7.4.

Additionally, a Weaving model (Sect. 7.5) is provided for composing existing OWL ontologies and UML-based Models.

7.2 Abstract Syntax

MOST will enable developers to describe M1 classes with UML and OWL in a platform independent way. An overview of the packages comprised by the MOST Abstract Syntax is depicted in Fig. 19.

The Abstract Syntax comprises a set of packages of different types: packages for the Reference Layer and packages for the implementation. The Reference Layer consists of MOST OWL Reference Layer, MOST UML-based Metamodeling Reference Layer and MOST Reference Layer. Concrete implementations of these layers are illustrated by TwoUse, which con-

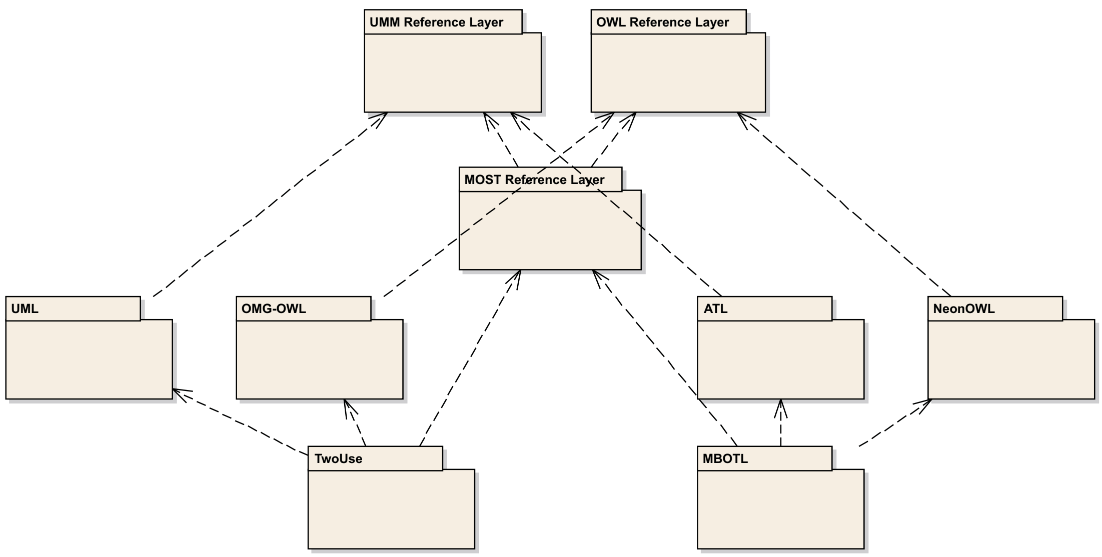


Figure 19: Overview of Packages in the MOST Metamodel.

cretely integrates UML and OMG OWL, and MBOTL [Silva Parreiras et al., 2008a], which is a language for transforming Ontologies that can profit from the MOST Reference Layer as well. Notice that the MOST Conceptual Architecture allows for plugging and adapting metamodels based on any of the three Reference Layers.

7.2.1 MOST Reference Layer

The objective of having a Reference Layer is to be able to implement a language-independent engine to query OWL ontologies and UML-based models. The MOST Reference Layer comprises extensions of the Kernel package of the MOST UML-based Metamodeling Reference Layer.

MOSTKernel The MOSTKernel package provide the integration between common constructs in OWL and UML-based Metamodeling: Package, Class, Property, Instance and Datatype. Basically, we apply the Adapter Design Pattern [Gamma et al., 1995] to connect classes from the MOST UML-based Metamodeling Reference Layer to classes from the OWL Reference Layer.

Following the nomenclature of Gamma et al. [Gamma et al., 1995], *Target* classes represent the interfaces from the MOST UML-based Metamodeling Reference Layer (Kernel::Namespace, Kernel::Classifier, Kernel::Datatype, Kernel::Property and Kernel::InstanceSpecification). *Adapter* classes are prefixed with the letter M (MPackage, MClassifier, MDatatype, MProperty and MIndividual). *Adaptee* classes are classes from the MOST OWL Reference Layer.

In most of the cases, associations between classes are substituted by operations in order to provide a common interface for different metamodels. This technique enable the concrete metamodels to keep associations between classes independently from the classes at the reference metamodel. An exception is the case of the class MPackage, since a connection between the class MPackage and the class BasicOWL::Ontology in an different way is unlikely.

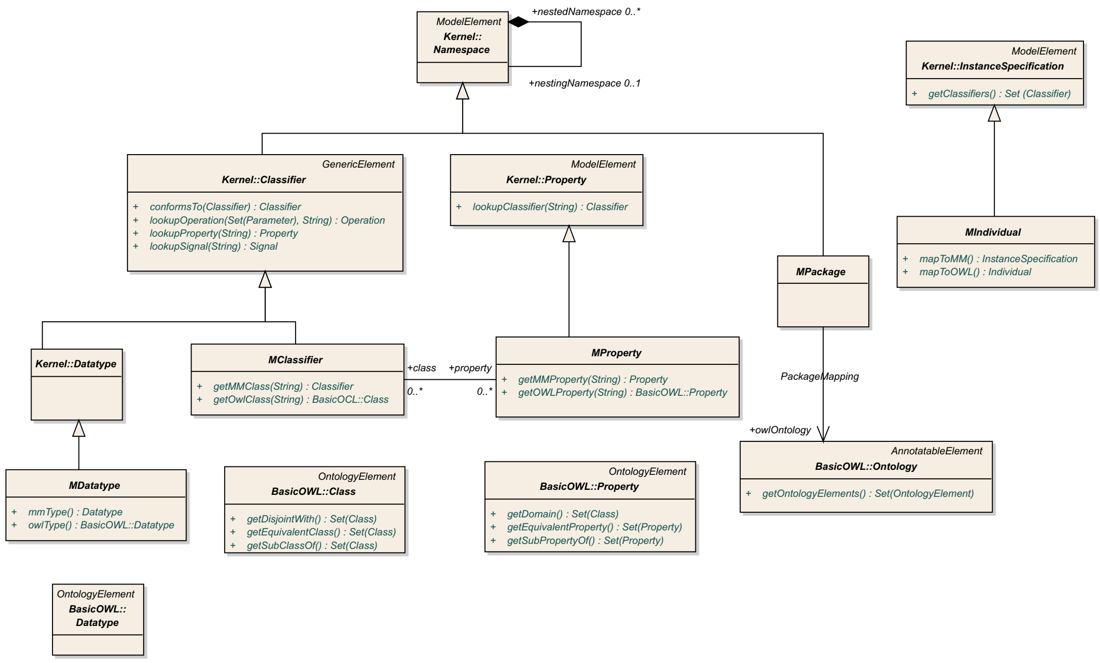

ModelElement
Kernel::Namespace
+nestedNamespace 0..*
+nestingNamespace 0..1
ModelElement
Kernel::InstanceSpecification
+ getClassifiers() : Set (Classifier)
GenericElement
Kernel::Classifier
+ conformsTo(Classifier) : Classifier
+ lookupOperation(Set(Parameter), String) : Operation
+ lookupProperty(String) : Property
+ lookupSignal(String) : Signal
ModelElement
Kernel::Property
+ lookupClassifier(String) : Classifier
MPackage
MIndividual
+ mapToMM() : InstanceSpecification
+ mapToOWL() : Individual
Kernel::Datatype
MClassifier
+ getMMClass(String) : Classifier
+ getOwlClass(String) : BasicOCL::Class
+class
+property
0..*
0..*
MProperty
+ getMMProperty(String) : Property
+ getOWLProperty(String) : BasicOWL::Property
PackageMapping
+owlOntology
AnnotatableElement
BasicOWL::Ontology
+ getOntologyElements() : Set(OntologyElement)
MDatatype
+ mmType() : Datatype
+ owlType() : BasicOWL::Datatype
OntologyElement
BasicOWL::Class
+ getDisjointWith() : Set(Class)
+ getEquivalentClass() : Set(Class)
+ getSubClassOf() : Set(Class)
OntologyElement
BasicOWL::Property
+ getDomain() : Set(Class)
+ getEquivalentProperty() : Set(Property)
+ getSubPropertyOf() : Set(Property)
OntologyElement
BasicOWL::Datatype

Figure 20: MOST Reference Layer Package MOSTKernel.

7.2.2 Connecting Metamodels

In order to implement concrete integration between OWL and UML-Based Metamodeling, the classes provided by the MOST Reference Layer are extended and adapted, according to the example below.

Marrying UML and OMG-OWL: TwoUse TwoUse is a concrete implementation of MOST that integrates UML and OWL [Silva Parreiras et al.,]. It imports the OMG OWL metamodel [OMG, 2007a] and package Core::Basic::Kernel of the UML2 [OMG, 2007b] metamodel.

TwoUse extends the MOST Reference Layer by adding the class TUClass, which specializes MOSTReferenceLayer::MOSTKernel::MClassifier and UML2::Core::Basic::Kernel::Classifier (Fig. 21). The class TUClass has as association ends the UML class and the OWL class. The associations are used to define the operations inherited from the class MOSTReferenceLayer::MOSTKernel::MClassifier like follows:

```
context TUClass
def: getMMClass(): (MOSTKernel::Classifier)
    self.umlClass.oclAsType(MOSTKernel::Classifier)

def: getOWLClass(): (OWL::Class)
    self.owlClass.oclAsType(OWL::Class)
```

The steps for extending the classes Datatype, Property and InstanceSpecification are similar. Figure 22 shows the class TUDatatype extending the class MDatatype.

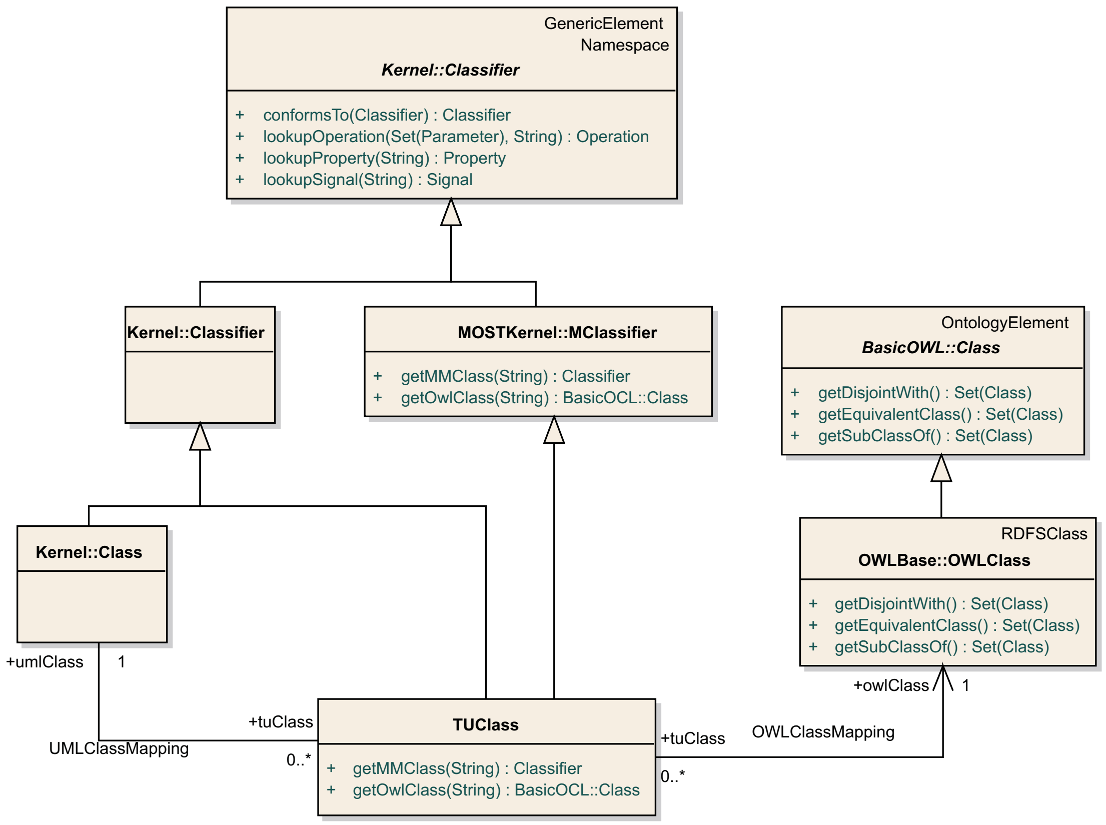

GenericElement
Namespace
Kernel::Classifier
+ conformsTo(Classifier) : Classifier
+ lookupOperation(Set(Parameter), String) : Operation
+ lookupProperty(String) : Property
+ lookupSignal(String) : Signal
Kernel::Classifier
MOSTKernel::MClassifier
+ getMMClass(String) : Classifier
+ getOwlClass(String) : BasicOCL::Class
OntologyElement
BasicOWL::Class
+ getDisjointWith() : Set(Class)
+ getEquivalentClass() : Set(Class)
+ getSubClassOf() : Set(Class)
Kernel::Class
RDFSClass
OWLBase::OWLClass
+ getDisjointWith() : Set(Class)
+ getEquivalentClass() : Set(Class)
+ getSubClassOf() : Set(Class)
+umlClass
1
+owlClass
1
+tuClass
UMLClassMapping
0..*
TUClass
+ getMMClass(String) : Classifier
+ getOwlClass(String) : BasicOCL::Class
+tuClass
OWLClassMapping
0..*

Figure 21: TwoUse Metamodel Extensions.

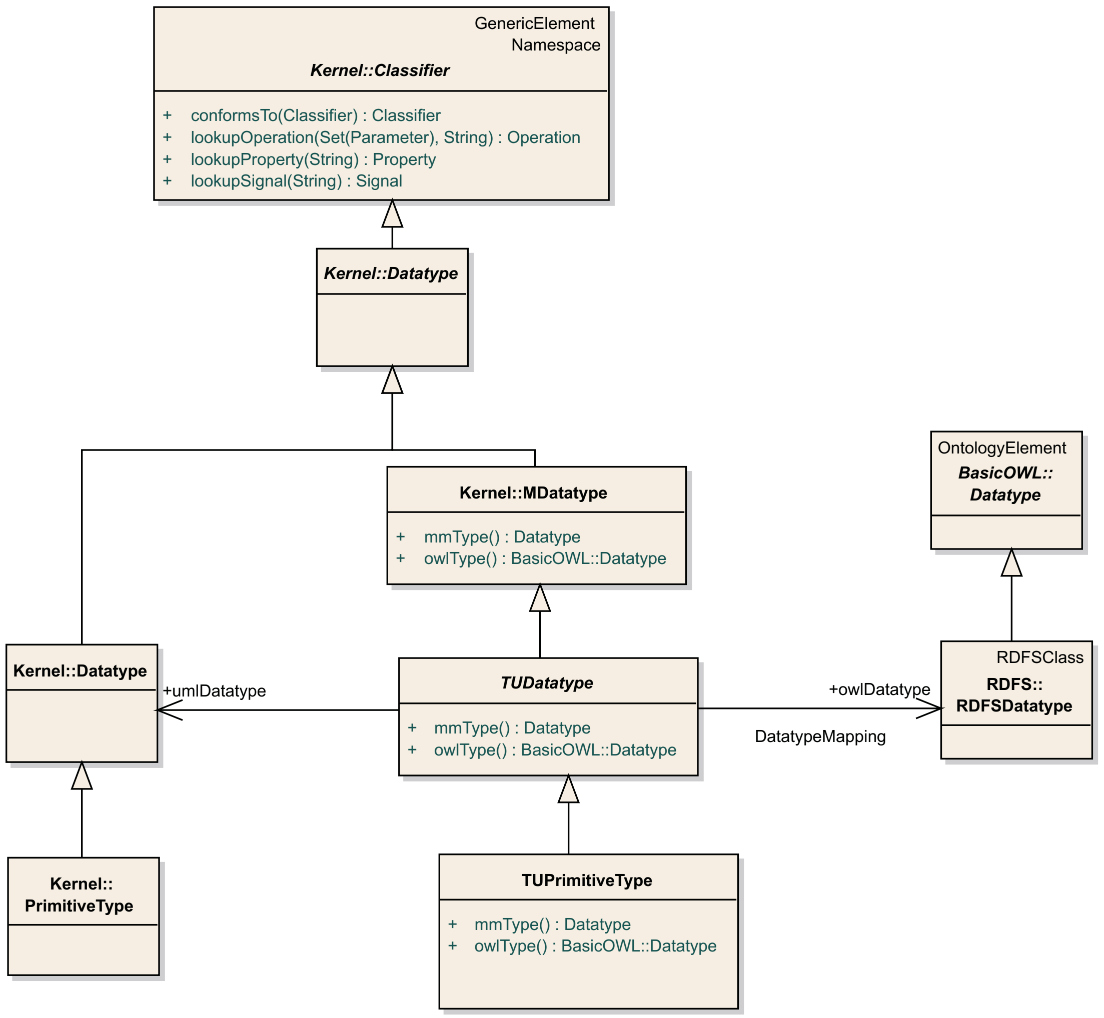


Figure 22: TwoUse Metamodel Extensions.

Additionally, we illustrate how datatypes can be mapped between the different models. The following expressions map the datatypes Boolean and String onto UML and OWL datatypes.

```
context TwoUse::TUPrimitiveType
  inv: self.name = 'Boolean'
         implies umlDatatype.name = 'Boolean'
            and owlDatatype.name = 'boolean'
  inv: self.name = 'String'
         implies umlDatatype.name = 'String'
           and owlDatatype.name = 'string'
  // ...
```

7.3 Concrete Syntax

Notations for modeling OWL ontologies have been developed, resulting in textual notations [Horridge et al., 2006] [Bechhofer et al., 2003] as well as in using UML as a visual notation [Brockmans et al., 2004] [Djurić et al., 2005b] [OMG, 2007a].

We cover the usage of UML Profiles as visual concrete syntax as well as the usage of an weaving approach for integrating existing models.

7.3.1 Visual Concrete Syntax

UML Profiles for OWL Among existing UML profiles for OWL, we use the UML profile for OWL proposed by OMG[OMG, 2007a]. The choice of such profile is justified by the fact that it covers all constructs of OWL-DL and does not require extra adornments, which eases its implementation and adoption. The UML profile for OWL is described in detail in the OMG ODM specification [OMG, 2007a].

UML Profiles for TwoUse We call the UML class diagram with some of the elements stereotyped by an UML Profile for OWL a *hybrid diagram.* The hybrid diagram comprises three different views, illustrated in Fig. 23: (1) the UML view with its OCL expressions, (2) the OWL view and its logical class definitions and (3) the TwoUse view, which integrates UML classes and OWL classes.

Considering the example of Fig. 23, the OWL view consists of nine classes, seven of which are named classes and two are unnamed classes. The restriction classes are required for reasoning on the subclasses `USSalesOrder` and `CanSalesOrder`. Thus, they only reside in the OWL view. The UML View comprises the six classes depicted in Fig.23. Applying the rules described in Sect.7.4, the TwoUse view will contain five classes.

Although we reuse an available UML profile for OWL to map onto TwoUse classes, OWL classes referred to by operations must be TwoUse classes too. To be compatible with tools that support UML2 extension mechanisms, the Opaque expressions must be specified with the stereotype <<DLExpression>>. This stereotype has the property `referredOwlClass [*]` and the values are the referred classes. Such reference is needed to match TwoUse classes later when writing mappings.

So far, these are the extensions to be done to provide mappings onto the TwoUse model. Those mappings are presented in Sect 7.4

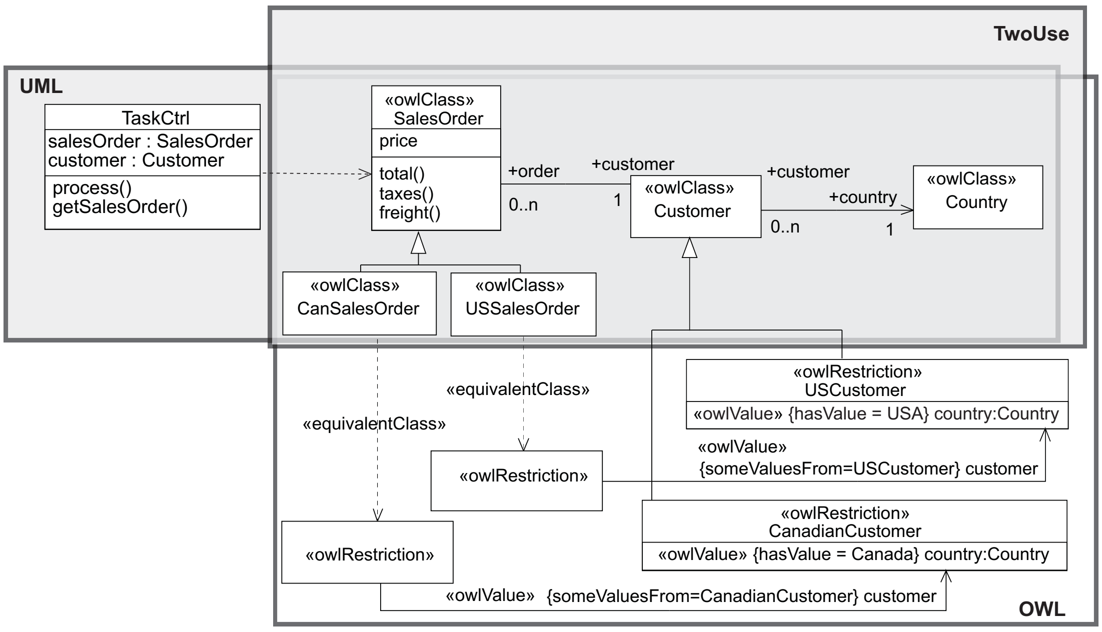


Figure 23: UML Class Diagram profiled with UML Profile for OWL and TwoUse Profile.

7.4 Mappings from UML Profiles onto TwoUse Models

The elements of the TwoUse view in the hybrid diagram (Fig.23) are mapped onto instances of two kinds of elements of the TwoUse metamodel: `TUClass` and `TUPackage`. The elements of the OWL view in the hybrid diagram map onto instances of the OWL metamodel. The elements of the UML view are copied, because they don't need to be changed, since the TwoUse metamodel imports the UML one.

The classes in the hybrid diagram to be mapped onto TwoUse classes can be matched as follows: (1) any class that has the stereotype `<<owlClass>>` and has any operation or any UML property declared or (2) any class with stereotype `<<owlClass>>`, of which the name is a property value of `ReferredOwlClass` property of the stereotype `<<DLExpression>>`.

Any class with the stereotype `<<owlClass>>` and with only properties stereotyped as `<<datatypeProperty>>` or `<<objectProperty>>` and that are not mapped onto TwoUse classes are mapped onto OWL classes.

Any class without any of the above-mentioned stereotypes results in a regular UML class. Properties can be available on OWL and be accessible from the OWL side or can remain only in UML. Properties to be available on both paradigms (UML and OWL) are stereotyped as TUProperty. A TwoUse package is any package that has TwoUse classes.

The relationships among elements from the TwoUse view and elements from the other views are preserved, as the TwoUse metamodel specializes both UML and OWL metamodels. No direct relationship is allowed between OWL and UML entities without being linked by TwoUse entities.

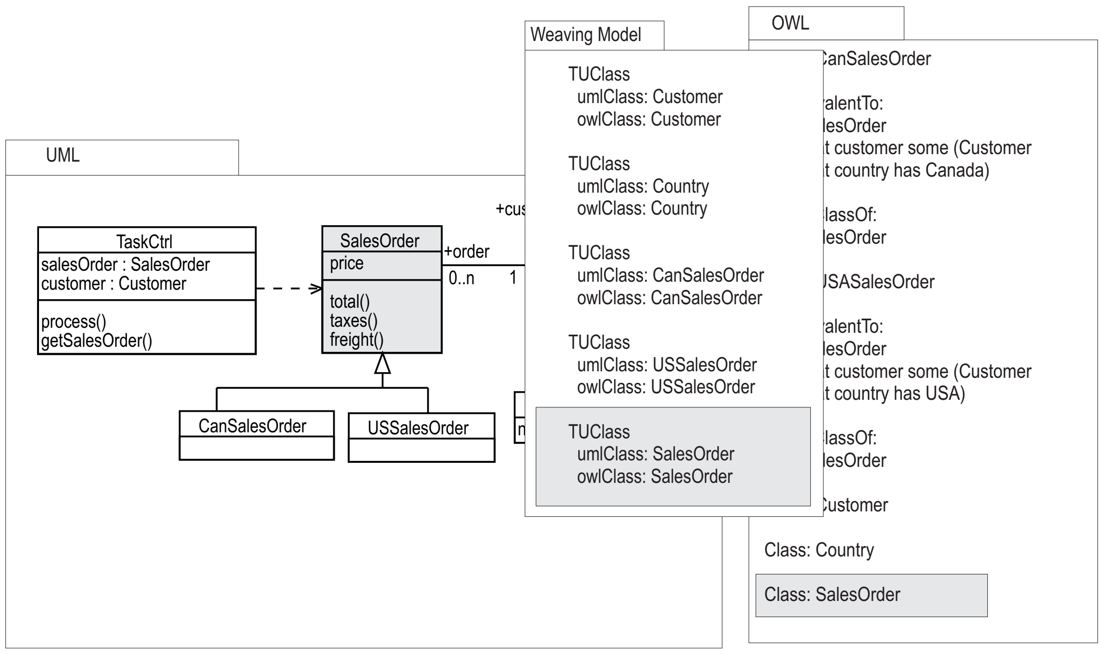


Figure 24: Weaving Model.

7.5 Weaving Model

To integrate existing UML2 models and OWL ontologies, an weaving model described by an weaving metamodel is used. The weaving model serves as input for the model transformation into the TwoUse model. For example, Fig. 24 depicts a weaving model linking the the UML Class SalesOrder and the OWL Class SalesOrder.

To provide a weaving metamodel, we have followed the generic weaving metamodel presented by Didonet Del Fabro et al. [Fabro et al., 2005]. Figure 25 depicts a preliminary metamodel for weaving existing OWL ontologies and UML models.

7.6 Foundation Library

The model libraries define a number of datatypes, class identifiers and operations that must be included in the implementation of MOST metamodel. These constructs are instances of metaclasses at M2 level. The foundation library exists at the M1 level, whereas the abstract syntax (metamodel) exists at M2 level. The foundation library is composed of the XML Schema Datatypes library, the RDF library and the OWL library.

Examples of M1 objects of the XML Schema datatypes library are the datatypes `gDay`, `gMonth` and `gYear`, having the M2 class `RDFS::RDFSDatatype` as metaclass. In the RDF library, for example, the M1 object `nil` has the M2 class `RDFS::RDFList` as metaclass. In the OWL library, interesting M1 objects are `Thing` and `Nothing`, both having the M2 class OWL::OWLClass as metaclass. These three libraries are based on the foundation library for RDF and OWL described in the ODM specification[OMG, 2007a].

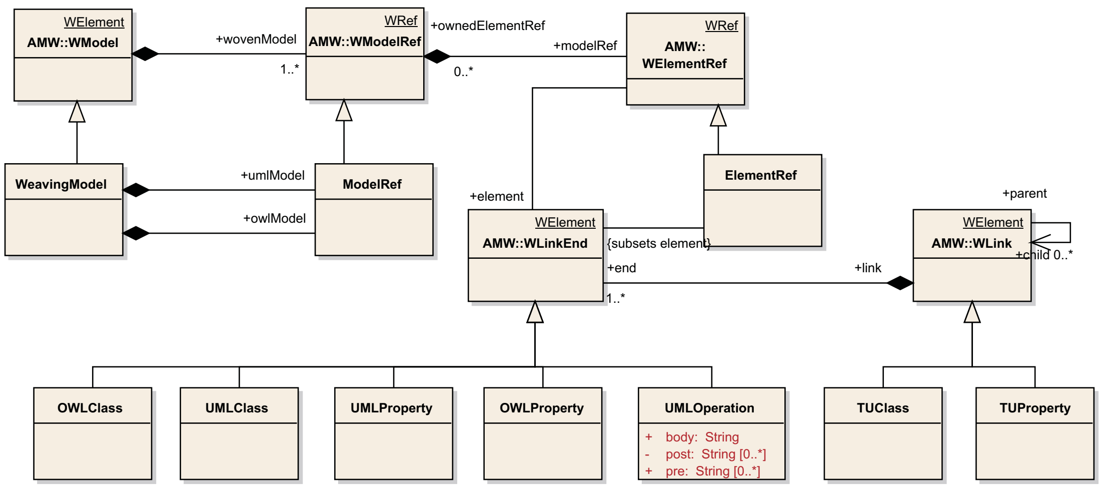


Figure 25: Weaving Metamodel.

The foundation library is not yet implemented is to be developed within the MOST project, according to future requirements of MOST use cases.

8 Validation

In this section we corroborate the metamodeling architecture by pointing where and how the use cases courses defined in Section4 are addressed in this document. Table 3 lists the use cases courses according to each use case and describes how the use case can be accomplished by this deliverable and where to find details in this document.

UC	Course	How to accomplish	Where
4.1.1			
	1	A Model Transformation Language with support to ontologies is not defined, but a guideline of how to extend the Reference Layer including an example is outlined.	7.2.2
4.1.2			
	1	An UML Profile is provided as well as a weaving model.	7.3
4.1.3			
	4.1.3	UML Profiles for OWL are suggested.	7.3.1
4.1.4			
	1	MOST provide compatibility with UML model by the Reference Layer	6.1
4.1.5			
	1	UML Profiles are defined as CS, transformation rules from CS to AS and a set of metamodels as AS.	7.3, 7.2, 7.4
	2	WFR are defined	7.2
4.1.6			
	1	The MOST Reference Layer allows for connecting different elements with the Metaclass MClassifier, which can be a bridge for integrating UML elements of different diagrams like artifacts, classes, signals, etc, since these elements are specializations of Classifier.	7.2.1
4.1.7			
	1	A weaving metamodel is outlined to be used with existing model weaving tools.	7.5
4.2.1			
	1	A Reference Layer and guidelines to implement the Reference Layer are provided.	5.2, 6.1,

Table 3: Use Case Validation

9 Conclusion

This document describes a combined metamodel for ontologies and metamodeling technologies based on initial requirements provided by MOST Use Case Partners. We build on top of the requirements to propose an extensible approach based on design patterns for adaptation and extension of existing metamodels. The resulting metamodels are analyzed from the perspective of requirement fulfilment.

Since Working Package 1 does not contribute with a tool and our approach is heavily based on reuse, minor changes on metamodels are preserved and foreseen during the implementation phase.

Acknowledgement

Thanks to Profs. Steffen Staab and Jürgen Ebert (UoKL) for their valuable comments.

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 7th Framework Programme project MOST number 216691 (cf. `http://most-project.eu`).

List of Figures

List of Tables

References

[com, 2008] (2008). Comarch oss suite - telecomunications, mobile, telecom network software homepage. `http://oss.comarch.com/`.

[Akehurst and Patrascoiu, 2004] Akehurst, D. H. and Patrascoiu, O. (2004). Ocl 2.0 - implementing the standard for multiple metamodels. *Electr. Notes Theor. Comput. Sci.*, 102:21–41.

[Arndt et al., 2007] Arndt, R., Troncy, R., Staab, S., Hardman, L., and Vacura, M. (2007). Comm: Designing a well-founded multimedia ontology for the web. In *Proc. of ISWC 2007 + ASWC 2007, Busan, Korea*, volume 4825 of *LNCS*, pages 30–43. Springer.

[Bechhofer et al., 2003] Bechhofer, S., Patel-Schneider, P. F., and Turi, D. (2003). OWL Web Ontology Language concrete abstract syntax. Available at `http://owl.man.ac.uk/2003/concrete/latest/`.

[Brockmans et al., 2006] Brockmans, S., Haase, P., Hitzler, P., and Studer, R. (2006). A metamodel and UML profile for rule-extended OWL DL ontologies. In *Proc. of 3rd European Semantic Web Conference (ESWC)*, volume 4011 of *LNCS*, pages 303–316. Springer.

[Brockmans et al., 2004] Brockmans, S., Volz, R., Eberhart, A., and Löffler, P. (2004). Visual modeling of owl dl ontologies using uml. In et al., S. M., editor, *Proceedings of the Third International Semantic Web Conference*, pages 198–213, Hiroshima, Japan. Springer.

[Bräuer and Demuth, 2007] Bräuer, M. and Demuth, B. (2007). Model-level integration of the ocl standard library using a pivot model with generics support. In *Proc. of 7th OCL Workshop at the UML/MoDELS Conferences*, Nashville, USA.

[Decker et al., 2005] Decker, S., Sintek, M., Billig, A., Henze, N., Dolog, P., Nejdl, W., Harth, A., Leicher, A., Busse, S., Ambite, J. L., Weathers, M., Neumann, G., and Zdun, U. (2005). TRIPLE - an RDF rule language with context and use cases. In *Rule Languages for Interoperability.*

[Djurić et al., 2005a] Djurić, D., Gašević, D., and Devedžić, V. (2005a). Adventures in modeling spaces: Close encounters of the semantic web and MDA kinds. In *Workshop on Semantic Web Enabled Software Engineering (SWESE 2005)*, Galway, Ireland.

[Djurić et al., 2005b] Djurić, D., Gašević, D., Devedžić, V., and Damjanovic, V. (2005b). A UML profile for OWL ontologies. In *MDAFA*, volume 3599 of *LNCS*, pages 204–219. Springer.

[Fabro et al., 2005] Fabro, M. D. D., Bézivin, J., Jouault, F., Breton, E., and Gueltas, G. (2005). Amw: a generic model weaver. In *Journées sur l'Ingénierie Dirigée par les Modeles (IDM05), pages = 105-114, note = 2-7261-1284-6.*

[Falkovych et al., 2003] Falkovych, K., Sabou, M., and Stuckenschmidt, H. (2003). Uml for the semantic web: Transformation-based approaches. In *Knowledge Transformation for the Semantic Web*, pages 92–106.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. (1995). *Design Patterns.* Addison-Wesley Longman, Amsterdam.

[Gašević et al., 2007] Gašević, D., Djurić, D., and Devedžić, V. (2007). MDA-based automatic OWL ontology development. *Int. J. Softw. Tools Technol. Transf.*, 9(2):103–117.

[Horridge et al., 2006] Horridge, M., Drummond, N., Goodwin, J., Rector, A., Stevens, R., and Wang., H. (2006). The Manchester OWL Syntax. In *OWL: Experiences and Directions (OWLED) 2006*, Athens, Georgia, USA.

[Knublauch et al., 2006] Knublauch, H., Oberle, D., Tetlow, P., and Wallace, E. (2006). A semantic web primer for object-oriented software developers. W3c working group note, W3C.

[Loecher and Ocke, 2004] Loecher, S. and Ocke, S. (2004). A metamodel-based ocl-compiler for uml and mof. *Electr. Notes Theor. Comput. Sci.*, 102:43–61.

[McGuinness and van Harmelen, 2004] McGuinness, D. L. and van Harmelen, F. (2004). OWL Web Ontology Language overview. Available at `http://www.w3.org/TR/2004/REC-owl-features-20040210/`.

[Motik et al., 2008] Motik, B., Patel-Schneider, P. F., and Horrocks, I. (2008). Owl 2 web ontology language – structural specification and functional-style syntax. Working draft, W3C.

[Oberle et al., 2004] Oberle, D., Eberhart, A., Staab, S., and Volz, R. (2004). Developing and managing software components in an ontology-based application server. In *Proc. of Middleware-04*, pages 459–477.

[OMG, 2001] OMG (2001). *Unified Modeling Language Specification Version 1.4*. Object Modeling Group.

[OMG, 2007a] OMG (2007a). *Ontology Definition Metamodel*. Object Modeling Group.

[OMG, 2007b] OMG (2007b). *Unified Modeling Language: Superstructure, version 2.1.1*. Object Modeling Group.

[Shalloway and Trott, 2002] Shalloway, A. and Trott, J. (2002). *Design patterns explained: a new perspective on object-oriented design*. Addison-Wesley, Boston, MA, USA.

[Silva Parreiras et al., 2008a] Silva Parreiras, F., Staab, S., Schenk, S., and Winter, A. (2008a). Model driven specification of ontology translations. In Lia, Q., Spaccapietra, S., and Yu, E., editors, *Conceptual Modeling - ER 2008, 27th International Conference on Conceptual Modeling, Barcelona, Spain, October 23-26, 2008, Proceedings*, volume 5231 of *Lecture Notes in Computer Science*. Springer.

[Silva Parreiras et al.,] Silva Parreiras, F., Staab, S., and Winter, A. TwoUse: Integrating UML models and OWL ontologies. Technical Report 16/2007, University of Koblenz-Landau. Available at `http://isweb.uni-koblenz.de/Projects/twouse/tr162007.pdf`.

[Silva Parreiras et al., 2007] Silva Parreiras, F., Staab, S., and Winter, A. (2007). On marrying ontological and metamodeling technical spaces. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7*. ACM Press.

[Silva Parreiras et al., 2008b] Silva Parreiras, F., Staab, S., and Winter, A. (2008b). Improving design patterns by description logics: A use case with abstract factory and strategy. In Kühne, T., Reisig, W., and Steimann, F., editors, *Modellierung 2008, 12.-14. März 2008, Berlin*, number 127 in LNI. GI.