myjournal manuscript No.
(will be inserted by the editor)

An Ontology-Based Framework for Domain-Specific Modeling

Tobias Walter · Fernando Silva Parreiras · Steffen Staab

Received: date / Revised version: date

Abstract Domain-specific languages (DSLs) provide abstractions and notations for better understanding and easier modeling of applications in a special domain. Current shortcomings of DSLs include learning curve and formal semantics. This paper reports on a framework that allows the use of ontology technologies to describe and reason on DSLs. The formal semantics of OWL together with reasoning services allows for addressing constraint definition, progressive evaluation, suggestions, and debugging. The approach integrates existing metamodels and concrete syntaxes in a new technical space. A scenario in which domain models for network devices are created illustrates the framework.

1 Introduction

Domain-specific languages (DSLs) are used to model and develop systems in application domains. Such languages are high-level and provide abstractions and notations for better understanding and easier modeling of applications in a special domain. A variety of different domain-specific languages are used to develop one large software system. Each domain-specific language focuses on different problem domains and as far as possible on automatic code generation [1].

Domain-specific languages are designed by DSL designers. They design abstract syntax, at least one concrete syntax and semantics. The developed DSL is used by the DSL users to build domain models modeling the domain of a software system.

WeST - Institute for Web Science and Technology
Universitätsstrasse 1, Koblenz 56070, Germany
E-mail: {walter, parreiras, staab}@uni-koblenz.de

We have identified the following challenges given by current approaches for domain-specific modeling [2]:

Tooling: DSL tooling provides abstraction and assistance for the domain to be modeled. The purpose of abstraction is to reduce software descriptions and implementations. Tool assistance aims at guiding the development process (debuggers, testing engines) [3]. *(challenge (1))*

Interoperability: In industrial projects, multiple languages are considered to model a software system. The facilities for language interoperability allow DSL users to model different static and behavioral aspects of the system and to freely shift between domain-appropriate languages. *(challenge (2))*

Formal semantics and constraints: To describe the meaning of concepts semantics are defined and to restrict the use of a concept constraints are defined. DSL semantics and constraints are often not defined explicitly but hidden in tools. To fix specific formal semantics and constraints for DSLs, they should be defined precisely in the language specifications. *(challenge (3))*

Learning curve: At the beginning, users of DSLs have not a deep understanding of the concepts a DSL provides. They need suggestions and guidance to use the DSLs. Suggestions and guidance facilitate the learning effort of a DSL. *(challenge (4))*

Domain analysis: The purpose of the domain analysis is to select and define the domain of focus and to collect relevant domain information and integrate it into a coherent domain model [4]. *(challenge (5))*

Some of the challenges depend on each other. Improving tooling enhances the user experience and the learning curve of DSLs. Formal semantics and constraints are the basis for interoperability and formal domain analysis. Formal semantics are required to provide tooling like debugging. Nevertheless, addressing the challenges is crucial for the successful adoption of domain-specific languages.

Modeling issues like formal semantics or interoperability motivated the development of ontology languages. Formal semantics constrain the meaning of models, such that their interpretations are the same for different persons (or machines). Description Logics [5] underpin the W3C standard Web Ontology Language (OWL) [6] and provide the foundations for ontology languages. OWL together with automated ontology and reasoning technologies provide a powerful solution for formally describing domain concepts.

Considering that some of the main challenges of domain-specific modeling motivated the design of OWL and ontology technologies, the following two questions arise: Which characteristics of ontology technologies may help in addressing current DSL challenges? What are the building blocks of a solution for applying ontology technologies in DSLs?

Recent work has explored ontology technologies to address some domain-specific modeling challenges. Tairas et al. [7] apply ontologies in the early stages of domain analysis to identify domain concepts (*challenge (5)*). Guizzard et al. [8] propose the usage of an upper ontology to design and evaluate domain concepts (*challenge (3)*) whereas Bräuer and Lochmann [9] propose an upper ontology for describing interoperability among DSLs (*challenge (2)*). Nevertheless, the application of ontology languages and technologies to address the remaining *challenges (1) and (4)* remains an open issue.

France and Rumpe present in [10] a research road map for model-driven development of complex software. The open challenges for MDE presented in their work completely cover our *challenges (1) and (4)*. France and Rumpe ask for the integration of formal techniques with MDE approaches to design and use formal (domain-specific) modeling languages.

Besides the integration, the complexity of the formal language being integrated should be hidden. Hence language designers and users may use the modeling languages they are familiar with as much as possible. If their language provides an insufficient expressivity they should be able to use the integrated formal language (*challenges (1)*).

Besides the integrated use of formal language, the analysis tools should be integrated with existing modeling language tools (*challenges (4)*).

1.1 Proposed Solution

To provide support for the *challenges (1)* and *(4)* we suggest an ontology-based framework for domain-specific modeling. For the support of tooling and learning of a DSL, it provides automated services that base on ontology technologies.

The framework provides tool support to DSL designers and DSL users. DSL users are guided during the modeling process, and are able to validate incomplete domain models. The correctness of the domain-specific language in development is important for DSL designers. They use services to *check the consistency* of the developed language, or they might exploit information about *concept satisfiability*, checking if it is possible for a concept in the metamodel to have any instances. When DSL users verify whether all restrictions and constraints imposed by the DSL metamodel hold, they use a reasoning service to *check the consistency* of domain models. If the domain models, created by users during the modeling process, are inconsistent, they rely on debugging support and inconsistency management for *detecting*, *diagnosing*, and *handling* the inconsistency [11].

To facilitate the learning curve, the framework supports DSL users with suggestions during building domain models and progressive evaluation of domain constraints. At the beginning, DSL users often are not familiar with the specific concepts a DSL provides. They often use generic concepts, but it is important for a domain model, that its elements have the most specific type (for example for generating the most specific code). DSL users select a model element and use a reasoning service for *dynamic classification.* Dynamic classification allows for determining the concepts which model elements belongs to dynamically, based on the instances descriptions.

1.2 Modeling Approach

To help DSL designers to define sound and valid languages whose use is supported by the aforementioned services, we create a new technological space. In [12] a technological space is defined as a *working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities.* Since we are describing a framework for domain-specific modeling, we will concentrate on the concepts and tools available for DSL designers and users.

The framework implementing this new technological space is arranged according to the OMG's layered architecture depicted in Figure 1. In addition, the figure refers to the sections where the corresponding parts are explained. The roles DSL user and DSL designer are assigned to the different layers they are responsible for.

The Ecore-based metamodeling language described by the M3 metametamodel is used by DSL designers at the M2 layer. Here new metamodels are designed. The language designers define the general structure of the language at the M2 layer and extend it by additional constraints. The framework allows for using the familiar Java-like KM3 syntax [13,14] to a very large extent. It is used to textually implement Ecore-based metamodels. If DSL designers recognize that it is not expressive enough they are able to define constraints within the metamodel in an integrated manner (cf. Section 5.1). To annotate Ecore-based metamodels with additional constraints, an integrated metamodeling language at the M3 layer is needed (cf. Section 5.2).

DSL users consider the metamodel (the abstract syntax definition of the DSL) and create instances at the M1 layer. These instances build the domain models (cf. Section 2).

Services in the framework like the ones mentioned above are provided to both, DSL designers and DSL users (cf. Section 5.3). The services instruct and guide DSL users during building domain models. Services allow to check the consistency of domain models, domain concepts should be suggested to DSL users, incomplete parts in the models should be detected and redundancies should be removed.

In the scope of this paper, a DSL framework is a tool for model-driven development of DSLs at the M2 layer and to support domain modeling at the M1 layer. Section 5 gives more details about the framework, its concepts and services. Here we will extend Figure 1.

In [15] we presented a framework for using ontologies with UML class-based modeling. In [16,17] we presented the idea of ontology-based design of domain-specific language. In this paper we are going to strengthening the approaches given in [15–17] by enriching domain-specific modeling with ontology technologies. While in [15] the integration of ontology languages and a general class-based modeling language was presented, in this paper we concentrate more on the design and use of domain-specific modeling languages. We are going to show, how DSL designers and users benefit from the advantages of ontology technologies. While [16] just presented the idea and challenges for the design and use of DSLs together with ontology technologies, in this paper we are strengthening the integration of DSLs and ontologies and the services which are provided to DSL designers and users.

We organize the remaining sections as follows: Section 2 describes the running example used through the paper and analyzes the DSL challenges to be addressed with ontology technologies. In [18], an ontology is defined as a formal, explicit specification of a shared conceptualization. In this paper we concentrate more on the ontology technologies usable for DSL design and use. Therefore, Section 3 describes the ontology language OWL as a software language by presenting a concrete syntax, a metamodel and its semantics. In addition we comment in this section on the reasoning technologies, which consider an ontology designed by the ontology language. The framework is described in Section 5 by presenting the language used to design DSL metamodels and the services it provides to designers and users. In Section 7 we present two possible implementations for a framework for ontology-based domain specific modeling. We revisit the running example in Section 8 and analyze related work in Section 6. Section 9 finishes the paper.

2 Scenario

Comarch[1], one of the industrial partners in the *MOST project*[2], has provided the running example used in this paper. It is a suitable simplification of the user scenario being conducted within the MOST project.

The company is specialized in designing, implementing and integrating IT solutions and services. For software development, Comarch uses model-driven methods with different kinds of domain-specific languages being deployed during the modeling process.

Comarch uses a domain-specific language to model physical network devices. DSL designers at Comarch design a DSL to specify devices from a specific family (e.g. the Cisco 7600 family) at the M2-layer. Here, the goal of DSL designers is to formally define the logical structures of devices and restrictions over these structures (which leads to the below listed *requirement (1)*, listed at the end of this Section).

DSL users use the DSLs defined by DSL designers to create domain models that describe concrete configurations of physical devices (M1-layer) [19].

The general physical structure of a Device consists of a Bay which has a number of Shelfs. A Shelf contains Slots into which Cards can be plugged. Logically, a Shelf with its possible Slots and Cards is stored as a Configuration.

Figure 2 depicts four steps of the development of a configuration of a Cisco device. Firstly (*step 1*), the DSL user starts with an instance of the general concept Cicso_Dev representing the physical device. A Cisco device requires at least one configuration. Thus he plugs in a Cisco_Configuration_760x element into the device.

In *step 2*, the DSL user adds exactly three slots to the device configuration. At this point, the DSL user verifies whether the configuration satisfies the DSL restrictions (*requirement (2)*). Although the domain model is incomplete it is not inconsistent. Thus queries against it are possible asking if the domain model consists of at least one configuration with at least one slot.

After adding three slots to the configuration of the physical device, the DSL user plugs in some cards to complete the end product (*step 3*). He may insert two SPA Interface Cards for 1-Gbps broadband connections and a controller for swapping cards at runtime (Hot Swappable OSM).

[1] `http://www.comarch.com/`

[2] `http://www.most-project.eu/`

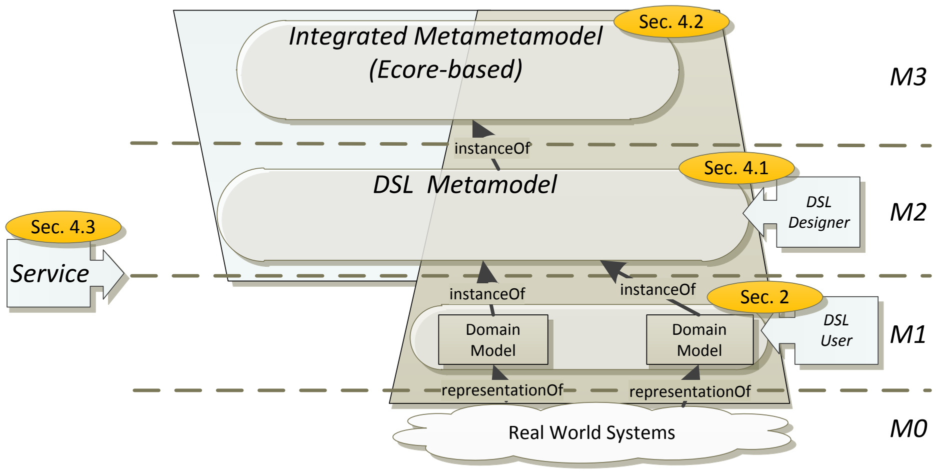


Fig. 1 DSL Designer and DSL User in the OMG four-layered architecture

At this point, the DSL user wants to check the consistency of his configuration by just invoking the corresponding functionality. This functionality is implemented by a reasoning service. In our example, the domain model in *step 3a* is inconsistent. Here debugging comes into play, since a DSL user needs an explanation why his model is inconsistent and how to correct it (*requirement (3) and (4)*). In *step 3b* an explanation service would explain that every configuration of Cisco requires a Supervisor card to control the device and that one of the three cards must be replaced by it.

The DSL defines the knowledge of which special types of cards are provided by a specific configuration. Having the information that a configuration in *step 4* is connected with three slots in which a Supervisor card and at least a Hot Swappable OSM or SPA Interface Card is plugged in, the refinement of the Cisco_Configuration_760x type by the more specific type Cisco_Configuration_7603 should be recommended to the DSL user (*requirement (3)*) by the framework. This recommendation is the result of a reasoning service.

Since it has been inferred that the Cisco device has the Cisco_Configuration_7603, in *step 4*, the recommendation service also suggests to change the type of the Cisco_Dev to the more specific type Cisco7603_Dev.

In the following, we concentrate on the requirements derived from the scenario and from the challenges mentioned in Section 1. The requirements may be classified with regard to the two actors concerned: DSL designer and DSL user. First we present the ones for the DSL designer:

1. *Constraint Definition* (*challenge (3)*). The DSL development framework should allow for defining constraints over the DSL metamodel. DSL designers have to define formal semantics of the DSL in development to describe constraints and restrictions the domain models have to fulfill.

The following requirements concern the DSL user:

2. *Progressive verification* (*challenges (1), (4)*). Even with an incomplete model, the DSL development framework must provide means for verifying constraints. For example, in *step 2* the DSL user may want to validate constraints.
3. *Suggestions of suitable domain concepts to be used* (*challenge (4)*). DSL users need suggestions of domain concepts to be used because they might not be familiar of all concepts the DSL provides. DSL users normally start the modeling with generic concepts like Cisco_Dev or Cisco_Configuration_760x. The framework suggests the refinement of elements to the most suitable ones, like Cisco_Configuration_7603 or Cisco-7603_Dev (*step 4*). Such classifications together with explanation help novice DSL users to understand how to use the DSL.
4. *Debugging (reasoning explanation)* (*challenges (1), (4)*). DSL users want to debug their domain models to find errors inside them and to get an explanation how to correct the model. They want to have information about consequences of applying given domain constructs. In this scenario, DSL users want to know that they have to replace an SPAInterface card with a Supervisor card (*step 3*).
5. *Different ways of describing constructs (syntactic sugar)* (*challenge (4)*). DSL users always are not familiar with all specific concepts a DSL provides. In the aforementioned scenario, DSL users do not have the complete knowledge of the Cisco Configuration 7603. Thus, they use an alternative way to describe an instance of this concept (*step 2 and 3*).

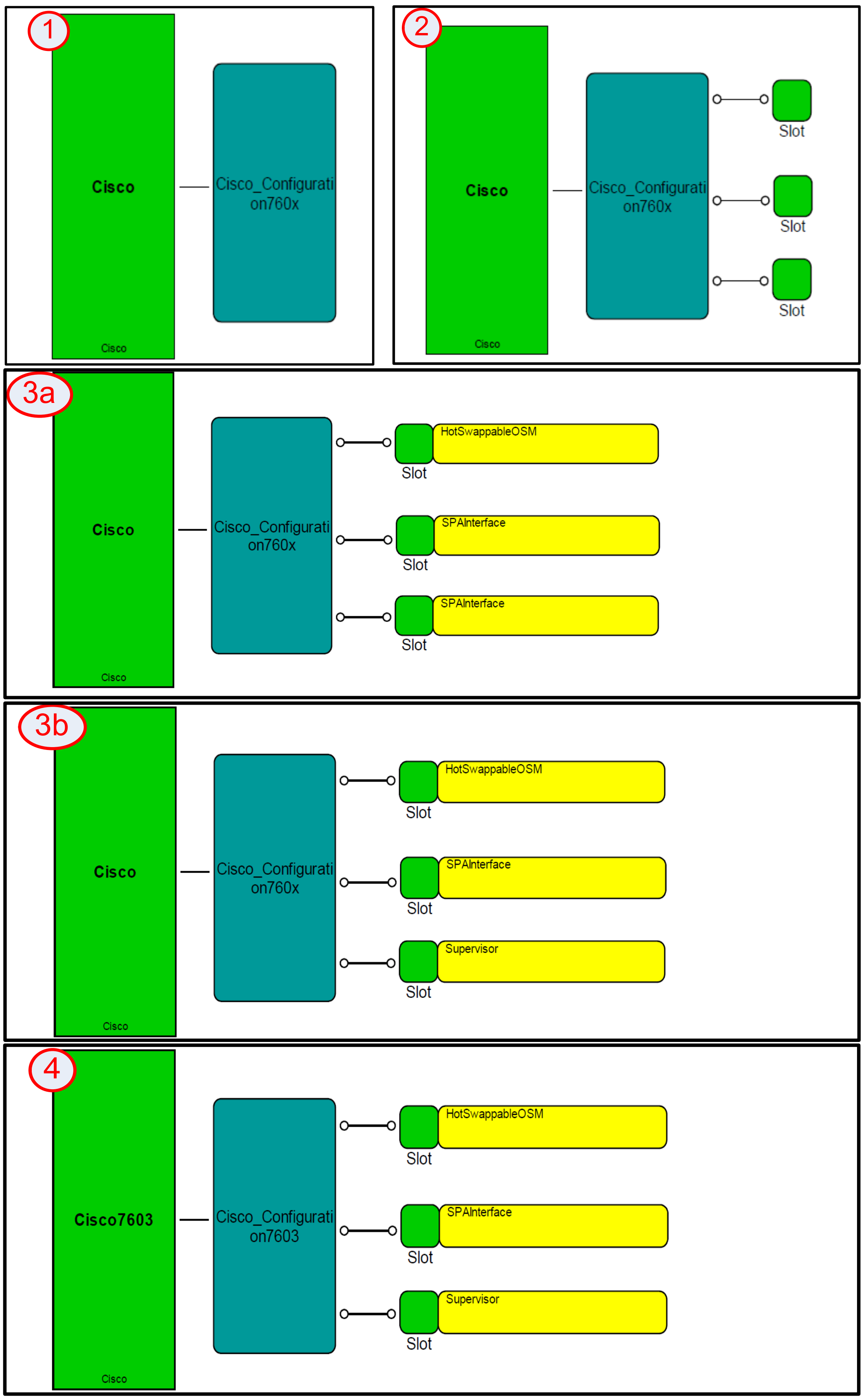


Fig. 2 Modeling a physical device in four steps (M1 layer)

Providing such alternative ways of designing domain models might improve productivity.

3 Ontology Languages and Technologies

In this section we are going to introduce ontology technologies.

Among ontology languages, we build on the W3C standard OWL. OWL actually stands for a family of languages with increasing expressiveness. OWL2 [6], the emerging new version of OWL, is a highly expressive language that allows for sound and complete calculi that are decidable as well as practically efficient.

In general, ontologies are used to define sets of concepts that describe domain knowledge and allow for specifying classes by rich, precise logical definitions. The difference between OWL and modeling languages such as UML class diagrams is the capability to describe classes in many different ways and to handle incomplete knowledge. These OWL features increase the expressiveness of the metamodeling language, making OWL a suitable language to formally define classes of modeling languages.

OWL2 features many types of axioms and thus provides different constructs to restrict classes or properties.

3.1 Ontology Example

In Figure 3 we give an example of an OWL2 ontology in functional-style syntax describing a part of the domain of physical devices presented in Section 2. Each line of the ontology presents one axiom and it represents the class descriptions (modeled in the terminological box (TBox) where the knowledge is declared), the relation descriptions (modeled in the relation box (RBox)), as well as the instance descriptions (modeled in the assertional box (ABox)).

In the first part, the TBox (Terminological Box) of an ontology is depicted. In the TBox all necessary classes are declared and their descriptions refined by additional class axioms. The class Cisco7603_Dev is subclass of Cisco_Dev and equivalent with an anonymous class just defining that it is linked via the property hasConfiguration with some specific Configuration7603. Configuration7603 is defined as specialization of Configuration760x. In addition, the classes Slot and Card are declared.

In the second part the RBox (Relationship Box) is depicted. In the RBox the object properties hasConfiguration, hasSlot, and hasCard are declared and bound

```
Ontology(DeviceOntology

// TBox: Axioms on Classes
Declaration(Class(Cisco_Dev))
Declaration(Class(Cisco7603_Dev))
EquivalentClasses(Cisco7603_Dev ObjectSomeValuesFrom(
        hasConfiguration Configuration7603))
SubClassOf(Cisco7603_Dev Cisco_Dev)
Declaration(Class(Configuration760x))
SubClassOf(Configuration760x Configuration7603)
Declaration(Class(Configuration7603))
Declaration(Class(Slot))
Declaration(Class(Card))
SubClassOf(Configuration760x ObjectMinCardinality(1 hasSlot Slot))
EquivalentClasses(Thing ObjectOneOf(_cisco7603_dev _config _slot1 _card))
DisjointClasses(Cisco_Dev Configuration760x Slot Card)

// RBox: Axioms on Relations ("Properties")
Declaration(ObjectProperty(hasConfiguration))
ObjectPropertyDomain(hasConfiguration Cisco_Dev)
ObjectPropertyRange(hasConfiguration Configuration760x)
Declaration(ObjectProperty(hasSlot))
ObjectPropertyDomain(hasSlot Configuration760x)
ObjectPropertyRange(hasSlot Slot)
Declaration(ObjectProperty(hasCard))
ObjectPropertyDomain(hasCard Slot)
ObjectPropertyRange(hasCard Card)

// ABox: Axioms on Inviduals
Declaration(Individual(_cisco7603_dev))
ClassAssertion(_cisco7603_dev Cisco_Dev)
Declaration(Individual(_config))
ClassAssertion(_config Configuration7603)
ObjectPropertyAssertion(hasConfiguration _cisco7603_dev _config)
Declaration(Individual(_slot1))
ClassAssertion(_slot1 Slot)
ObjectPropertyAssertion(hasSlot _config _slot1)
Declaration(Individual(_card))
ClassAssertion(_card Card)
ObjectPropertyAssertion(hasCard _slot1 _card)
DifferentIndividuals(_cisco7603_dev _config _slot1 _card)
)
```

Fig. 3 Example of an OWL2 ontology

via a domain and range axiom to the classes they come from and go to.

In the third part the ABox (Assertional Box) is depicted. In the ABox the individuals _cisco7603_dev, _config, _slot1, _card are declared, assigned by their respective types using a class assertion and linked by an object property assertion.

3.2 OWL2 Syntax and Model-theoretic Semantic

OWL2 provides model-theoretic semantics [20] which are defined by an interpretation function and are related to the semantics of description logics [5].

An interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ consists of a non-empty set $\Delta^{\mathcal{I}}$, the interpretation domain, and a mapping $\cdot^{\mathcal{I}}$, which associates each class description C with a set $C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$, each object property description P with a binary relation $P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$, and each individual i with an element $i^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$.

In Table 1 we present a set of OWL2 constructs to model the TBox, RBox and ABox. We highlight their

OWL2 Construct	**OWL2 Functional-Syntax**	**Semantics**
TBox class axioms and expressions		
Top Concept	Thing	$\mathsf{Thing}^{\mathcal{I}} = \Delta^{\mathcal{I}}$
Class Declaration	Declaration(Class(C))	$C^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$
Subclass Definition	SubClassOf(C D)	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$
Equivalent Classes	EquivalentClasses(C D)	$C^{\mathcal{I}} = D^{\mathcal{I}}$
Disjoint Classes	DisjointClasses(C D)	$C^{\mathcal{I}} \cap D^{\mathcal{I}} = \emptyset$
Class Intersection	ObjectIntersectionOf(C D)	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$
Class Union	ObjectUnionOf(C D)	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$
Class Complement	ObjectComplementOf(C)	$\Delta^{\mathcal{I}} \backslash C^{\mathcal{I}}$
Existential Quantification	ObjectSomeValuesFrom(P C)	$\{x \mid \exists y : (x,y) \in P^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\}$
Universal Quantification	ObjectAllValuesFrom(P C)	$\{x \mid \forall y : (x,y) \in P^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
At Most Restriction	ObjectMaxCardinality(n P C)	$\{x \mid \sharp\{y \in \Delta^{\mathcal{I}} \mid (x,y) \in P^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \leq n\}$
At Least Restriction	ObjectMinCardinality(n P C)	$\{x \mid \sharp\{y \in \Delta^{\mathcal{I}} \mid (x,y) \in P^{\mathcal{I}} \wedge y \in C^{\mathcal{I}}\} \geq n\}$
Enumeration	ObjectOneOf($i_1 ... i_n$)	$\{i_1^{\mathcal{I}} ... i_n^{\mathcal{I}}\} \subseteq \Delta^{\mathcal{I}}$
RBox object property axioms and expressions		
Object Property Declaration	Declaration(ObjectProperty(P))	$P^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$
Object Property Domain	ObjectPropertyDomain(P C)	$\{x \mid \exists y : (x,y) \in P^{\mathcal{I}}\} \subseteq C^{\mathcal{I}}$
Object Property Range	ObjectPropertyRange(P C)	$\Delta^{\mathcal{I}} \subseteq \{x \mid \forall y : (x,y) \in P^{\mathcal{I}} \rightarrow y \in C^{\mathcal{I}}\}$
ABox individual axioms and expressions		
Individual Declaration	Declaration(Individual(i))	$i^{\mathcal{I}} \in \Delta^{\mathcal{I}}$
Class Assertion	ClassAssertion(i C)	$i^{\mathcal{I}} \in C^{\mathcal{I}}$
Object Property Assertion	ObjectPropertyAssertion(P i j)	$(i^{\mathcal{I}}, j^{\mathcal{I}}) \in P^{\mathcal{I}}$
Different Individuals	DifferentIndividuals(i_1 i_n)	$i_k^{\mathcal{I}} \neq i_j^{\mathcal{I}}$ $(1 \leq k < j \leq n)$

Table 1 OWL2 constructs and semantics (excerpt)

syntax and describe their semantics for a given interpretation $\mathcal{I}$. In [21] a complete list of OWL2 constructs and its semantics is described.

3.3 Open World vs. Closed World and Domain Assumption

While the semantics of MOF-based modeling and OCL adopts the closed world assumption (CWA), description logics and OWL adopt the open world assumption (OWA) by default. The closed-world assumption states that the elements and their relations in the model are all known.

The open world assumption assumes incomplete information as default and allows for validating incomplete models which are still in the design phase.

3.3.1 Closing the world

Many approaches were developed for closing knowledge bases and allowing reasoners to validate integrity constraints. In general there are two different approaches: those which try to close the knowledge base by additional facts and those which introduce new language constructs for closing the knowledge base.

A first variant of using additional facts proposes the use of closure axioms. A closure axiom on a property consists of a universal quantification that acts along the property to define that it can only be filled by the specified fillers [22].

In [9] a more comprehensive approach is presented, which allows for locally closing the world. Every concept is closed by defining that it is equivalent with the set of all known instances.

An approach that simulates the local CWA by introducing a new language construct is the one presented in [23, 24]. Here a $\mathcal{K}$-operator is introduced which allows for locally closing concepts and roles. The $\mathcal{K}$-operator only considers instances which are known by the knowledge base.

3.3.2 Closing the domain

Although the open world assumption has many advantages we have to ensure that the validation of integrity constraints defined in OWL2 is still possible.

The closed domain assumption (CDA) states that the set of all individuals in the domain coincides with the set of individuals explicitly mentioned in the ontology. The CDA ensures the unique name assumption (UNA) and the type which is explicitly assigned to an individual.

UNA: The unique name assumption (UNA) requires that if instances have different names they are understood as different. In OWA the UNA is not considered, since two instances are not declared as different. Using the *Different Individuals* construct provided by OWL2 it is possible to explicitly declare a set of individuals as different.

Types: To ensure that a given instance only has the asserted type all other concepts are declared as disjoint. This fact is achieved by the use of the *Disjoint Classes* axiom.

Instances: Only the instances which are explicitly defined in the ontology are considered in CDA. Thus the top concept Thing is defined as equivalent with the set of instances the ontology contains. This fact is achieved by using the Enumeration construct.

3.4 Metamodel of OWL2

Figure 4 depicts an excerpt of the OWL2 metamodel that we use subsequently. Each OWL2 ontology consists of a set of axioms. Class axioms for example are the EquivalentClasses axioms or the SubClassOf axioms. The EquivalentClasses axiom is used to describe two or more class expressions as equivalent, whereas the SubClassOf axiom defines exactly one class expression to be the subclass of another. A possible class expression for example is the ObjectSomeValuesFrom expression which describes those individuals which are connected via a given object property to at least an individual of the given class expression.

Beside class descriptions, individuals are part of an ontology. Class assertions are axioms that are used to assert individuals having as type the given class expression.

The ontology in Figure 3 *conforms to* the metamodel in Figure 4 .

3.5 Reasoning

Based on an OWL2 ontology, standard reasoners like Pellet [25] provide reasoning services such as consistency checking, satisfiability checking, and subsumption checking.

An interpretation $\mathcal{I}$ is called a model of the TBox and RBox of an ontology, if it satisfies all its axioms on classes and properties. The interpretation $\mathcal{I}$ is called a model of the ABox of an ontology, if it satisfies all its axioms on individuals.

In the following we describe four standard reasoning services.

Consistency checking: checks, if the ABox is consistent with regard to the TBox and RBox (the ABox is consistent, if it has a model $\mathcal{I}$ which is also a model of the TBox and RBox).

```
SELECT ?i
WHERE {
    ?i rdf:type [
        rdf:type owl:Restriction ;
        owl:onProperty :hasSlot ;
        owl:someValuesFrom :Slot
        ]
    }
```

Fig. 5 SPARQL query

Satisfiability checking: checks, if the a class expression C is satisfiable. C is satisfiable if $C^{\mathcal{I}} \neq \emptyset$ for some model $\mathcal{I}$ of the TBox containing C and the RBox.

Subsumption checking: checks, if C_{sub} is subsumed by C_{sup}. C_{sub} is subsumed by C_{sup} if $C^{\mathcal{I}}_{sub} \subseteq C^{\mathcal{I}}_{sup}$ for all models $\mathcal{I}$ of the TBox and RBox.

Classification Checking: checks, if i is an instance of the class expression C. i is an instance of C, if $i^{\mathcal{I}} \in C^{\mathcal{I}}$ for all models $\mathcal{I}$ of TBox, RBox and ABox.

3.6 Querying

The current W3C recommendation for the SPARQL Protocol and RDF Query Language (SPARQL is a recursive acronym) corresponds to version 1.1 [26]. Originally SPARQL is designed to query RDF graphs.

Below we depict a SPARQL query which asks for all instances which are connected (via hasSlot) with a slot.

SPARQL 1.0 has only be defined as a query language over RDF graphs, not taking into account RDF Schema or OWL ontologies [27].

Answering full SPARQL queries on top of OWL has already preliminarily been addressed so far [28,29] and is proposed to be provided by SPARQL 1.1 [30].

The semantics and their entailment is similar to the one of ontology languages. It is based on interpretations of RDF basic graph pattern which are mapped to OWL axioms defined in an ontololgy.

In [31] a translation of a SPARQLAS query to a SPARQL query is presented.

The problem identified in [31] lies in SPARQL and its syntax for RDF triples. If a user wants to query an ontology which is represented in OWL functional-style syntax, he needs to translate his OWL syntax to the proper representation in RDF triples. This translation, if done manually, can be time consuming and is prone to syntactic as well as to semantic errors.

A simpler syntax is the one of SPARQLAS. The following SPARQLAS query asks for all individuals which have some slot:

```
Query ( ClassAssertion(?i ObjectSomeValuesFrom(hasSlot Slot) ) )
```

Its syntax is adapted to the OWL2 functional-style syntax. Thus similar to the OWL2 functional-style syntax

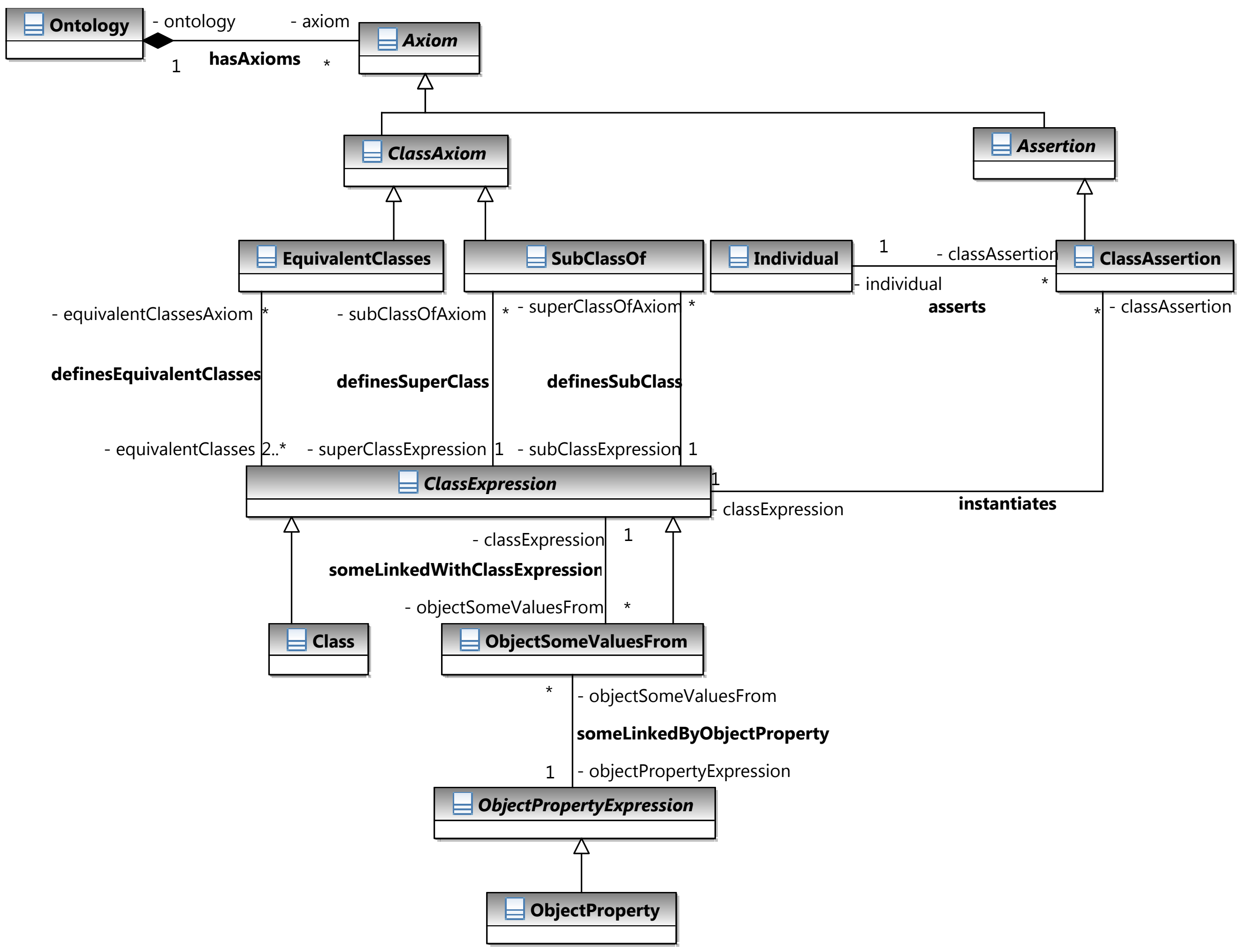


Fig. 4 Excerpt of the OWL2 metamodel [6]

presented in Figure 3. The semantics of the query are the same as the SPARQL query in Figure 5.

4 Relationship between Software Modeling Space and Ontology Space

In Section 1.2 we presented the software modeling approach, where all models are described by metamodels, which in turn are modeled using a metamodeling language like Ecore.

In Section 3 we introduced the *ontology space*. The space allows for designing ontologies, which conform to the ontology language OWL 2.

Before we combine both spaces in Section 5, we are going to discuss its relationship and describe the commonalities and differences.

Figure 6 depicts an overview of the relationship between a software modeling space and an ontology space.

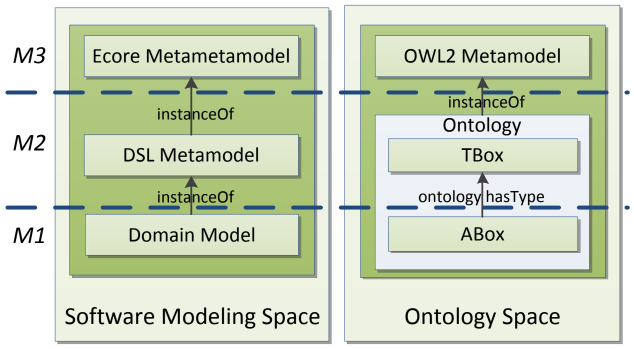


Fig. 6 Comparison of Software Modeling Space and Ontology Space

In the following we are going to compare the Ecore metamodeling language with the ontology language OWL 2. We will establish a mapping between both languages, which represents the common concepts. Additionally, we depict differences of both languages.

4.1 Common Concepts

In this section we are going to the common concepts the respective spaces provide. We consider the concepts of the metamodeling language Ecore and those of the ontology language OWL 2.

OWL 2 allows for modeling instance layer (ABox) and concept layer (TBox). With respect to the software modeling approach and besides Ecore used to model metamodels at the M2 layer, we consider Instance, Link, and Attribute Assignment as the concepts to create a model at the M1 layer.

Comparing the Ecore language and the OWL 2 language, at first glance we intensionally find similar concepts, which are juxtaposed in Table 2.

(1) An Ecore metamodel allows for specifying classes, datatypes and relations like references or attributes. The TBox of an OWL2 ontology as well allows for describing classes and its relations. (2) Classes in Ecore correspond to classes in OWL. With respect to the intensional semantics both class concepts represent a set of instances. (3) The references in Ecore correspond to object properties in OWL. They both represent sets of relations between instances of given types. (4) The attributes in Ecore correspond to data properties in OWL, since they both describe relations between instances and values of a predefined datatype. (5)The datatypes of attributes in Ecore semantically comprise atomic values like the datatypes in OWL.

OWL 2 ontologies additionally allow for modeling an ABox consisting of individuals and assertions. (6) With respect to the intensional semantics instance in a domain model correspond to individuals in an OWL2 ontology. Both are classified by classes. (7) In domain models links are instances of references and represent connections between two instances. In OWL 2 object property assertions are used to define connections between two individuals with respect to an object property. (8) In domain models attribute assignments define the relation between an instance and a value. In OWL 2 data property assertions are used to define the relation between an individual and a value with respect to a data property.

4.2 Differences

After we illustrated commonalities of Ecore and OWL 2, we continue the comparison by depicting differences. For the ontology language OWL 2, we mention those constructs, which are not replaceable by respective counterparts in Ecore-based metamodels. In the case of the Ecore metamodeling language we mention those constructs, which are not directly representable in an ontology.

4.2.1 OWL 2

Besides the description of classes, properties and instances, OWL 2 provides a comprehensive set of class expressions and axioms used to extend the description of modeled data in ontologies.

In OWL 2, class expressions and property expressions are combined to form new class expressions. Class expressions represent sets of individuals by formally describing the properties of individuals. Class expressions in OWL 2, which have no counterpart in Ecore, are the ObjectIntersectionOf, ObjectUnionOf, and ObjectComplementOf for the standard set-theoretic operations on class expressions (in logical languages these are usually called conjunction, disjunction, and negation, respectively). Furthermore, constructs for quantified expressions for the description of classes containing those instances being linked with some individual (using the ObjectSomeValuesFrom class expression) or only with individuals (using the ObjectAllValuesFrom class expression) of a given type, are not provided by Ecore. OWL 2 provides the description of classes by enumerating individuals. A counterpart of the ObjectOneOf class expression is not available in Ecore.

The following class expression, which is defined as equivalent to the class Device describes those individuals, which are linked via the property hasConfiguration with at least one individual of type ComplexConfiguration or with one of the individuals config7603 or config7604.

```
EquivalentClasses(Device ObjectSomeValuesFrom(hasConfiguration
        ObjectUnionOf(ComplexConfiguration ObjectOneOf(config7603
        config7604))))
```

OWL 2 provides an extensive set of axioms used to state what is true in the domain [6]. In particular OWL 2 provides the use of class axioms and property axioms.

Class axioms are used to express relationships between two or more class expressions. The EquivalentClasses axiom states that two class expressions describe the same set of individuals, while the DisjointClasses axiom states that the sets of individuals described by both class expressions are disjoint. In the listing above the EquivalentClasses axiom states the equivalence of the set of individuals described by Device and the set described by the ObjectSomeValuesFrom class expression. Ecore allows for relating classes by specialization relationships. Additional relations between the sets of instances described by classes are not possible.

♯	Ecore Concept		OWL 2 Concept	
1	Metamodel	Metamodel	TBox	Ontology
2		Class		Class
3		Reference		ObjectProperty
4		Attribute		DataProperty
5		Datatype		Datatype
6	Model	Instance	ABox	Individual
7		Link		ObjectPropertyAssertion
8		AttributeAssignment		DataPropertyAssertion

Table 2 Mapping between Ecore concepts and OWL 2 concepts.

Object property axioms are used to characterize and establish relationships between object property expressions. An OWL 2 object property transitivity axiom describes that an object property expression is transitive. Furthermore, OWL 2 allows for stating that two object properties are equivalent or disjoint. If two object properties are equivalent or disjoint, the sets of relations between individuals they describe are equivalent or disjoint. Ecore does not provide constructs to relate the sets of links described by an reference.

OWL 2 allows for composing two or more object properties to one object property chain being the specialization of another object property. The object property hasDeviceCard in the following is defined as the composition of hasCard and hasDeviceCard. In Ecore the composition of references is not possible.

```
SubObjectPropertyOf(SubObjectPropertyChain(hasConfiguration hasCard)
    hasDeviceCard)
```

4.2.2 Ecore

The main distinction of the Ecore metamodeling language compared to OWL 2 is the definition of packages. Ecore allows for (sub-)packaging classes and data types. The hierarchical (sub-)packaging concepts are not available for ontologies.

Metamodeling languages like CMOF [32] or grUML[33] compared to OWL 2 allow for the definition of attributes for associations that connect two classes. Hence, links in models may be attributed. OWL 2 does not allow for defining attributes (or data properties) assigned to object properties. In OWL 2 only classes can be defined as a domain of a data property.

4.3 Other Relationships between Software Models and Ontologies

In [8,17] ontologies are used as a pure domain model with layers for domain types and domain instances, respectively. Here, ontologies play the role of conceptual domain models, which are used in domain engineering to describe the problem domain a software system should support. Conceptual domain models are of descriptive nature. It consists of domain instances (individuals), which describe the system instances in the real world. All domain instances and their relations build one layer (ABox), which is part of the ontology. To classify domain instances, ontologies consist of domain types (classes) lying in a separate layer (TBox).

In [34,35] Gasevic et al. use UML class diagrams with a specific UML profile for OWL to model ontologies. In general, one class diagram represents an ontology with TBox and ABox, i.e. UML classes are used to model OWL classes, UML associations and attributes are used to model properties, and UML instances are used to model OWL individuals.

In contrast to other works, we consider ontologies as one single representation for metamodel and domain model. Here, the TBox of the ontology consists of language concepts typically defined in the metamodel of a DSL and the ABox consists of respective instances.

5 Languages and Services for Ontology-Based Domain-Specific Modeling

In this section we are going to present the ontology-based framework for domain-specific modeling.

Figure 7 enriches Figure 1 and represents the framework in more detail. It shows the layered architecture with additional technical details of the framework.

The framework supports both, DSL designers and DSL users.

DSL designers define domain-specific languages at the M2 layer. They require a concrete syntax to model the metamodels together with constraints. Our framework provides a combination of the Java-like KM3-syntax [13] (a concrete syntax for Ecore [36]) and an OWL syntax. DSL designers are able to describe classes in DSL metamodels seamlessly integrated with OWL axioms and expressions. In Section 5.1 we exemplify the design of integrated metamodels.

To build metamodels, DSL designers require a metamodeling language whose abstract syntax is described by a metametamodel. This metametamodel is defined at the M3 layer. A metametamodel which provides a tool-ready reusable implementation is the Ecore metametamodel [36]. To support the DSL designer in enriching metamodels by constraints and formal semantics we integrate the metametamodel with a language that is more expressive than Ecore. We consider an integration of OWL2 to describe integrated metamodels with seamlessly embedded OWL2 axioms and expressions. The integration of Ecore and OWL2 is presented in Section 5.2.

DSL users may then use the developed DSL with additional benefits. Results are domain models (M1-layer) like the ones in Section 2. Having formal semantics of the DSL, different reasoning services are available. Services are presented in Section 5.3.

DSL users build domain models (cf. Section 2) using the DSL developed by the DSL designer. Based on the formal semantics and constraints restricting the use of the DSL, different reasoning services are available. In Section 5.3 we specify a set of services. They describe an interface of the framework for DSL designers and DSL users. All services rely on ontology technologies. Services get as input an ontology which is a projection of metamodel (to the ontology TBox) and domain model (to the ontology ABox).

5.1 Integrated Metamodeling

The static structure of domain models as well as extended model-theoretic semantics are defined by DSL designers in integrated metamodels.

In Figure 8, we see an excerpt of an M2 metamodel that is created by a DSL designer using the integrated metamodeling language. Using the KM3 syntax, he defines that a Cisco_Dev has Cisco_Configurations, a Cisco7603_Dev is a specialization of Cisco_Dev, each Cisco_Configuration has Slots and in each Slot one to many cards can be plugged in. All specific cards are specializations of Card.

The DSL designer defines additional formal semantics and constraints using an embedded variant of the OWL2 Manchester style concrete syntax [37], which is integrated with the existing KM3 syntax. In Figure 8, the designer states that every Cisco7603_Dev device has at least one Cisco_Configuration7603. All possible configurations must have a slot in which a Supervisor card is plugged in. A Cisco_Configuration760x is a Cisco_Configuration7603 if and only if it has exactly three slots in which either a HotSwappableOSM card or a SPAInterface card is plugged in. Furthermore, if a Supervisor card is part of a configuration, the same configuration cannot have VoiceInterface card.

```
class Cisco_Dev {
  reference hasConfiguration [1−∗]: Cisco_Configuration760x;
}
class Cisco7603_Dev extends Cisco_Dev, equivalentWith hasConfiguration
        min 1 Cisco_Configuration7603 {
}
class Cisco_Configuration760x extends (hasSlot min 1 Slot) and (hasSlot
        some (hasCard some Supervisor)) {
  reference hasSlot [1−∗]: Slot;
}
class Cisco_Configuration7603 extends Configuration, equivalentWith (
        hasSlot exactly 3 Slot) and (hasSlot some (hasCard some (
        HotSwappableOSM or SPAInterface))) {
}
class Cisco_Configuration780x extends (hasSlot min 1 Slot) and (hasSlot
        some (hasCard some VoiceInterface)) {
  reference hasSlot [1−∗]: Slot;
}
class Slot {
  reference hasCard [0−∗]: Card;
}
class Card {
}
class Supervisor extends Card and inv(hasCard) only inv(hasSlot) only
        hasSlot only hasCard only (not VoiceInterface) {
}
class SPAInterface extends Card {
}
class HotSwappableOSM extends Card {
}
class VoiceInterface extends Card {
}
```

Fig. 8 Example of an M2 metamodel

5.2 Integration of Ecore and OWL

To design integrated metamodels like the one in Figure 8 DSL designers need an integrated metamodeling language. In the following we are going to discuss how to integrated the metamodeling language Ecore and OWL.

We have chosen Ecore for the integration with OWL because it represents the metametamodel of the Eclipse Modeling Framework [36] a technological space which provides a set of freely available modeling frameworks, tools, and implementations. Nevertheless the integration approach presented in the following is similar to the integration of OWL with other class-based metamodeling languages (e.g. with MOF [32] or grUML [33]).

The integration is established in three steps: *mapping of concepts*, *integration of concepts* and *projection.*

Step 1: Mapping based on intensional knowledge. Before an integration bridge between Ecore and OWL can be established, the different concepts the two languages provide must be compared and related. The relation of constructs is based on the intensional semantics and the knowledge framework developers have of the languages Ecore and OWL. The result of a relation is depicted in Table 2 in Section 4.

Step 2: Integration of concepts. The integration of different constructs relies on some basic integration tasks, which are informally described in the following:

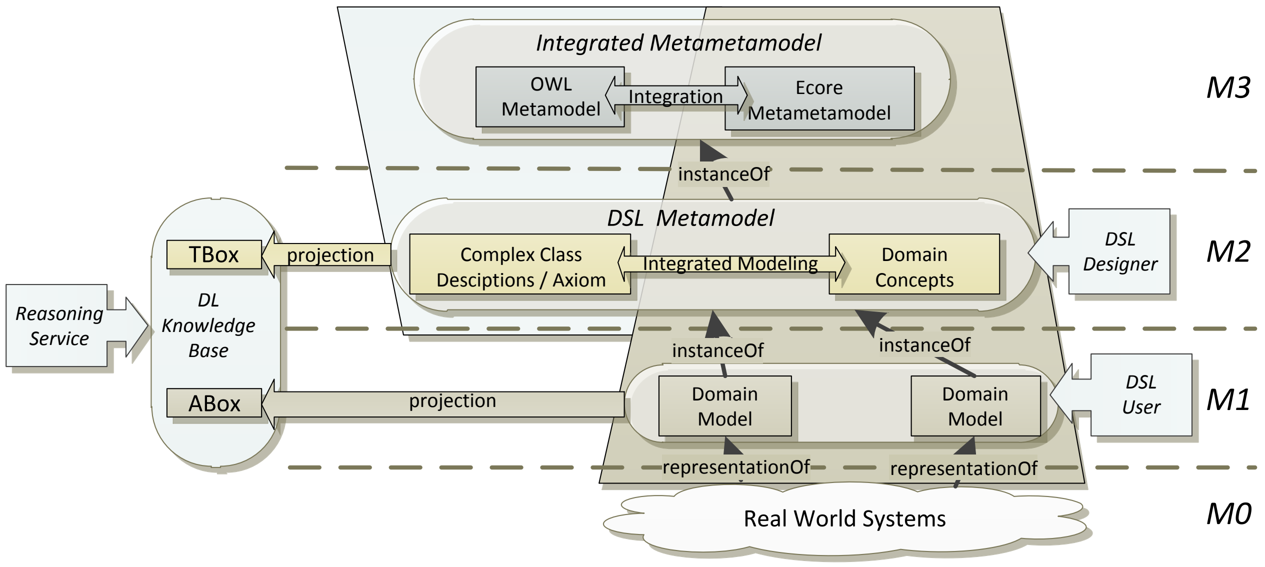


Fig. 7 Bridging Ecore and OWL

Merge of Concepts: If two concepts have the same intensional meaning they are merged. All incident associations, specialization relations and all nested attributes of the classes to be merged are moved to the new class.

Specialization of Concepts: If one concept has a more specialized meaning than some other concept, they are related by a specialization relationship.

Relation between Concepts: If two concepts are related to each other they are connected by an association.

For each correspondence between two concepts in the mapping in Table 2, one of the integration tasks is performed. The result of the integration tasks applied on the Ecore metametamodel and the OWL2 metamodel is a new, integrated metametamodel. It is depicted in Figure 9.

(1) Specialization of Ontology Concept: The Ecore concept Metamodel is declared as a specialization of Ontology. This allows for adding OWL2 axioms to Ecore metamodels.

(2) Merge of Class Concepts: The Ecore concept Class is merged with the OWL2 concept Class. Hence, all classes within an integrated metamodel can be involved in class axioms, like the equivalent classes axiom.

(3) Specialization of Object Property Concept: A specialization relationship between Reference and ObjectProperty is created. This integrations allow Ecore references being involved in different object property axioms and class expressions.

(4) Specialization of Data Property Concept: A specialization relationship between Attribute and DataProperty is created. This integrations allow Ecore attributes being involved in different object property axioms and class expressions.

(5) Merge of Datatype Concepts: The two classes for data types in Ecore and OWL2 are merged because they intensionally have the same meaning. Both represent a set of values.

Since the Ecore metametamodel does not provide explicit concepts for the design of instances and their links, there is no integration with concepts of OWL 2. Instances and their links are considered by a projection service, which translates these elements to respective OWL 2 elements according to the mapping in Table 2.

The metametamodel in Figure 9 allows for describing metamodels like the one depicted in Figure 8. The metamodel conforms to the integrated metametamodel.

Step 3: Projection. For language designers and users the interoperability with other tools is important. In particular, language designers and users having created a metamodel or domain model want to benefit from ontology technologies. These technologies perform reasoning tasks in the OWL2 technological space. Hence, our tools project metamodels and models to an ontology such that they can serve as input for reasoning tools.

We propose the implementation of a projection service. A projection service for a given integrated language is used to extract those parts of a hybrid model conforming to the integrated metamodel which are built by constructs of the given language to be integrated.

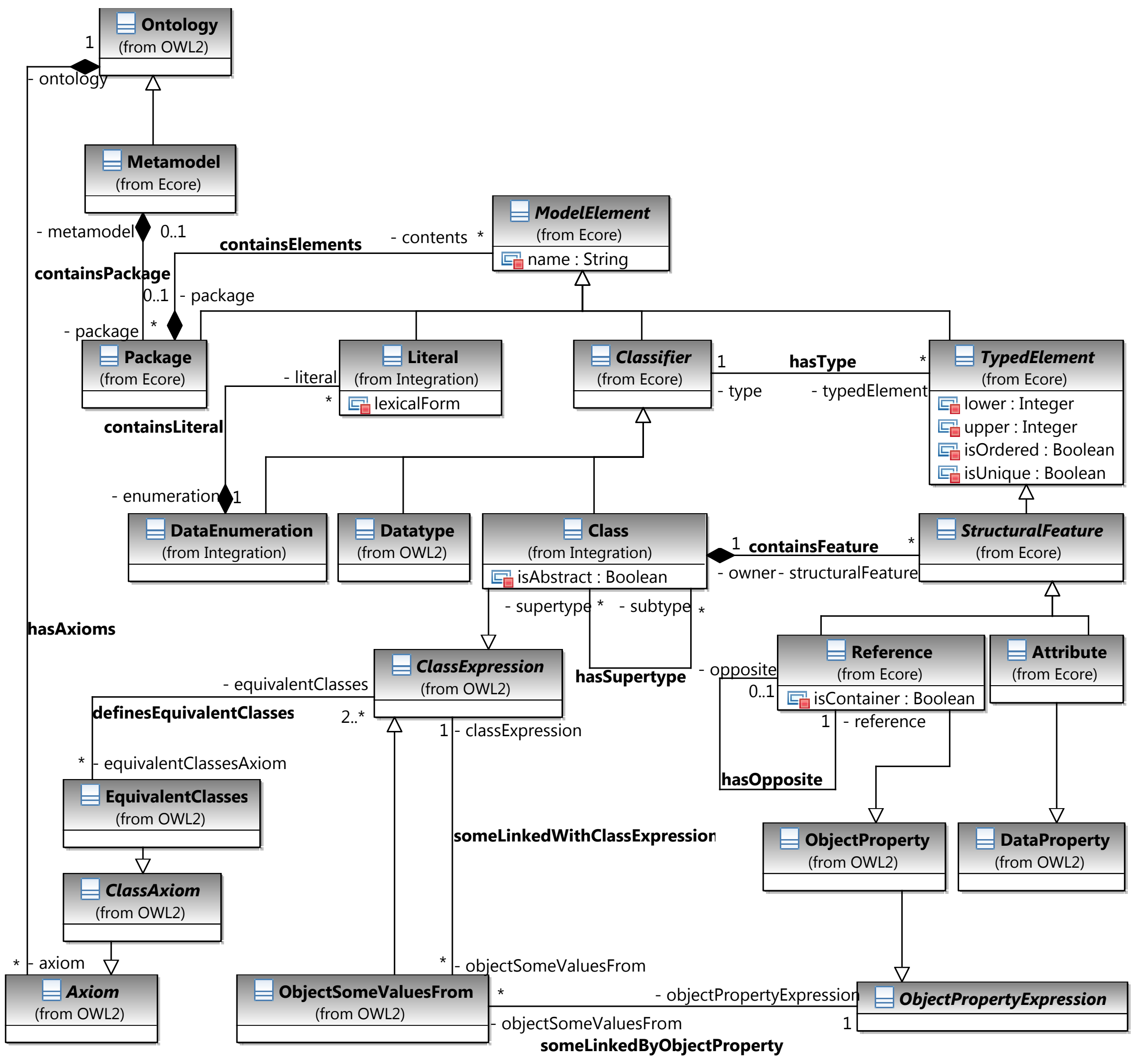


Fig. 9 Integrated Metametamodel Ecore+OWL

The projection service for OWL transfers elements built by ontology constructs and keeps their relations to other ontology elements. The OWL projection service returns a pure OWL2 ontology. Figure 10 depicts a specification of a projection service getting as input a (hybrid) metamodel and a domain model. The service returns one ontology which conforms to the OWL2 metamodel. It is one formal representation of both metamodel and domain model.

5.2.1 Concrete Syntax

Figure 8 gives an example of using the concrete syntax to define a metamodel.

The concrete syntax of our framework is based on KM3. The motivation is that DSL designers should use the Java-like KM3 syntax as much as they can. To benefit from OWL, they should be able to annotate elements of their DSL metamodel in a textual manner. Hence, we have extended the grammar of the KM3 concrete syntax by new non-terminals which are defined in grammars of a textual OWL2 concrete syntax.

In Figure 8 the Cisco7603_Dev and the Cisco_Configuration7603 class are annotated by an OWL2 equivalent classes axioms.

We have extended the KM3 syntax by an adaptation of the OWL2 Manchester Syntax [37] to get a natural controlled language for coding OWL2-based annotations. We have chosen the Manchester Syntax because its notations are closely related to the notations of KM3. The mapping between abstract and concrete syntax is solved using the *EMFText framework* for textual concrete syntax specification of DSLs [38].

Name	*Projection Service*
Signature	Ontology project(Metamodel mm, Model m)
Description	Creates a new ontology o which only consists of those parts of mm which conform to ontology constructs defined in the metamodel of an ontology language. In addition, the model m is projected into the same ontology o: (6) Projection of Instances: for each instance in a model m conforming to some class in mm, o is extended by an individual having as type the projection of the class. (7) Projection of Links: For each instance in a model m conforming to some reference in mm an object property assertion is created in o connecting the individuals which are projections of source and target instance in m. (8) Projection of Attribute Assignments: For each instance in a model m conforming to some attribute in mm a data property assertion is created in o assigning the value to the individual, which is a projection of the corresponding instance.

Fig. 10 Projection of a metamodel and a domain model to an ontology

An editor for designing metamodels with the combined textual concrete syntax is provided by the TwoUse Toolkit[3] (cf. Section 7).

5.3 Services for DSL Designers and Users

In the following we are going to present services for DSL designers and users. The services are composed of standard reasoning services, services for inconsistency management and querying services.

We specify the services by giving their name, their signature and a description of their activity.

The standard reasoning services presented in the following specify the services provided by standard reasoners (cf. Section 3.5).

Services for inconsistency management are provided to DSL users to diagnose, detect and handle the inconsistency in domain models.

DSL users use a query service for predefined queries developed by the DSL designer.

[3] `http://code.google.com/p/twouse`

Name	*Consistency Checking*
Signature	boolean isConsistent(Ontology o)
Description	The service returns true, if the ABox $\mathcal{A}$ of the ontology o is consistent with regard to the TBox $\mathcal{T}$. Otherwise, it returns false.

Fig. 11 Reasoning Service: Consistency Checking

Name	*Satisfiability Checking*
Signature	boolean isSatisfiable(Ontology o, ClassExpression c)
Description	The service returns true if the class expression c is satisfiable. Otherwise, it returns false. (c is satisfiable if some instance of c can be created in o and o does not become inconsistent)

Fig. 12 Reasoning Service: Satisfiability Checking

5.3.1 Reasoning Services

In the following we are going to present a set of reasoning services available for language designers and users.

Consistency Checking. DSL users having created a domain model want to check the consistency of the model with respect to the metamodel. Our framework supports DSL users by providing a consistency checking service. Before invoking the service, the framework projects both, the metamodel and the domain model, by the projection service presented in Figure 10 into an OWL2 ontology. The domain model is part of the ABox, where the metamodel with all axioms and expressions lies in the TBox of the ontology. The service specified in Figure 11 gets as input an ontology model, which conforms to the OWL2 metamodel, and returns, whether the domain model is consistent.

A DSL user, who checks the domain model in Figure 2 (3a), gets the answer, that the model is not consistent (because the supervisor card is missing).

Satisfiability Checking. The task of language designers is to build a metamodel, describing the abstract syntax of domain-specific modeling language. To validate the metamodel, the language designer wants to check if its classes are instantiable. Our framework supports language designers with a satisfiability checking service. Before the service can be applied, the metamodel (and an empty domain model) must be projected to an ontology. The satisfiability checking service gets as input the ontology and an OWL2 class expression which is a projection of the class to be validated.

The class definition of Cisco_Dev in Figure 13 is not satisfiable because on the one side it must have at least two configurations, on the other side it must have at most one configuration. The reasoning service specified

```
class Cisco_Dev equivalentWith (hasConfiguration max 1
      CiscoConfiguration760x) and (hasConfiguration min 2
      CiscoConfiguration760x) {
  reference hasConfiguration [1..*]: CiscoConfiguration760x;
}
```

Fig. 13 Unsatisfiable Class

Name	*Classification*
Signature	boolean classifies(Ontology o, ClassExpression c, Individual i)
Description	The service returns true if i is an instance of the class expression c. Otherwise it returns false. i is an instance of c if it fulfills all properties and restrictions given by the class c.

Fig. 14 Reasoning Service: Classification

in Figure 12 allows for detecting the class Cisco_Dev as an unsatisfiable class.

Classification. DSL users, who are not familiar with the language, often start with generic concepts. In the example given in Figure 2, the DSL user starts modeling a Cisco_Dev with a Cisco_Configuration760x, although he wants to model a specific device, since he adds a specific set of cards to the device.

In Figure 14 a classification service is specified. After a projection of metamodel and model to an ontology, the service can be used to check whether an individual has as a type the given class expression.

In domain modeling the service can be used to refine the type of a given model element. In Figure 2 (3) the DSL user asks for the most specific type of the Cisco_Dev and CiscoConfiguration760x element. Based on the service given in Figure 14 the framework computes all possible types the elements might have (by iterating through all named classes in the ontology and only listing those for which the classifies service returns true). In step (4) in Figure 2, the DSL user replaces the types to the valid ones Cisco7603_Dev and Cisco-Configuration7606, because they are the most specific ones.

Inconsistency Explanation. In Figure 2 (3a) the DSL user describes a Cisco_Dev device with a missing supervisor card. The domain model is inconsistent which simply is detected by the isConsistent service specified in Figure 11.

If models are inconsistent, DSL users require an explanation. This explanation is computed by the service given in Figure 15. The service adopts the projection of metamodel and model, the ontology, and computes for each inconsistency in the domain model a minimal

Name	*Inconsistency Explanation*
Signature	Set<Set<Axiom>> inconsistencyExplanation(Ontology o)
Description	The service returns a set S of minimal sets of axioms for each inconsistency. If at least one axiom of each set $s_i \in S$ is removed from o, o becomes consistent.

Fig. 15 Reasoning Service: Inconsistency Explanation

```
CiscoConfiguration760x equivalentTo hasSlot some hasCard some
      Supervisor
_hotswappable type HotSwappableOSM
_spainterface1 type SPAInterface
_spainterface2 type SPAInterface
```

Fig. 16 Inconsistency Explanation by Reasoners

Name	*Satisfiability Explanation*
Signature	Set<Set<Axiom>> explanation(Ontology o, ClassExpressions c)
Description	The service returns a set S of sets of axioms where each set $s_i \in S$ of axioms is minimal and entails the unsatisfiability of c (c is unsatisfiable if c equivalentWith Nothing.

Fig. 17 Reasoning Service: Satisfiability Explanation

set of axioms. If at least one axiom of the ontology is removed, it becomes inconsistent.

For the inconsistency in the model in Figure 2 (3) the service returns axioms given in Figure 16 (each line consists of one axiom):

The axioms in Figure 16 are rendered in the natural readable OWL2 Manchester syntax. A DSL user sees that each CiscoConfiguration760x must have some slot in which some supervisor card is plugged in. Problems in the domain model are the instances for the HotSwappableOSM card and the two instances of SPAInterface. If one of those types is changed to Supervisor the model becomes consistent.

In Section 5.3.2 we show how the framework evaluates these sets and gives suggestions to DSL users how to repair their models.

Satisfiability Explanation. A language designer uses the service specified in Figure 12 to check whether a class is satisfiable. Having detected an unsatisfiable class the language designer needs an explanation why it is unsatisfiable. The service specified in Figure 17 returns for a given unsatisfiable class a minimal set of axioms explaining the unsatisfiability.

Figure 18 depicts the explanation for the class Cisco-7603_Dev, because it is unsatisfiable too. The reason for the unsatisfiability is that Cisco7603_Dev is a subclass of Cisco_Dev which must have at most one configuration and at least two configurations at the same time.

```
Cisco_Dev equivalentTo hasConfiguration max 1 CiscoConfiguration760x
      and hasConfiguration min 2 CiscoConfiguration760x
Cisco7603_Dev subClassOf Cisco_Dev
```

Fig. 18 Unsatisfiability Explanation by Reasoners

5.3.2 Inconsistency Management

Syntactic consistency ensures that a specification conforms to the metamodel of the modeling language, specified by the language designers. This guarantees that the model is well-formed [39].

In [40] inconsistency management is defined as the process by which inconsistencies between software models are handled to support the goals of the stakeholders concerned. The process of inconsistency management consists of activities for *detecting*, *diagnosing*, and *handling* inconsistency [11].

Detecting: Detection of inconsistencies is the activity of checking for inconsistencies in instance models with regard to a metamodel. Different approaches for the detection of inconsistencies are possible [41]. In this work, we consider a logic based approach with detecting logical inconsistency [5], where models together with metamodels are projected to ontologies which are expressed as description logics knowledge bases and which are consistent if an interpretation exists.

Diagnosing: The diagnosis of inconsistencies is concerned with the identification of the elements causing an inconsistency [41]. The diagnosis is a basic for inconsistency handling. Several methods are available for debugging ontologies and identifying inconsistent parts [42]. Since models specified by DSL users are graphs composed of instances of classes and links, in the following we present a service delivering the instances in a model causing the inconsistency.

Handling: Inconsistency handling is concerned with identifying possible actions for dealing with an inconsistency [41]. For ontologies several repair strategies have been developed [42]. In the following we use a service suggesting valid types for instances involved in an inconsistency.

In the following we describe inconsistency management services for DSL user having built a domain model. They are going to detect that their domain model is inconsistent, they are going to diagnose the domain model to find the parts involved in the inconsistency and they are going to repair the domain model.

The metamodel considered by DSL users to build instances in the domain model is assumed as valid and all concepts are satisfiable. Input of all services is a metamodel and a corresponding domain model.

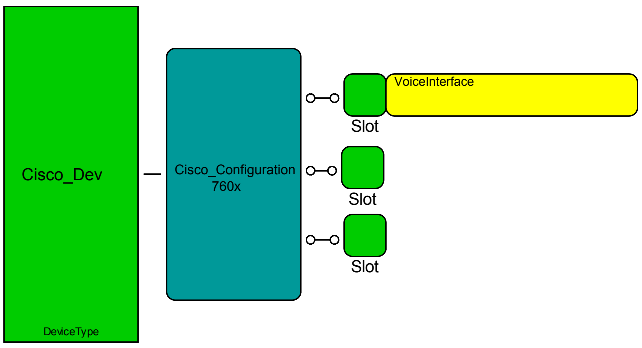


Fig. 19 Inconsistent Domain Model (M1 layer)

Name	*Inconsistent Elements Service*
Signature	Set<Set<Instance>> inconsistentElements(Metamodel mm, Model m)
Description	Project mm and m to an ontology o=project(mm, m). If isConsistent(o)=true, return null. If isConsistent(o)=false, compute an explanation E=inconsistencyExplanation(o). For each sets in E put those instances into one set in S which are involved in an axiom in the given set in E.

Fig. 20 Inconsistent Elements Service

Services for Inconsistency Management With respect to the DSL metamodel in Figure 8, the domain model in Figure 19 is not consistent.

As mentioned above, Figure 2 (3) depicts an inconsistent domain model.

For the detection of the inconsistency the domain model is projected together with the metamodel to an ontology. The ontology is used as an input for the consistency checking service specified in Figure 11. For the domain model in Figure 19 and the metamodel in Figure 8 the service isConsistent returns false.

Having an inconsistency detected, DSL users may want to diagnose the inconsistency. To identify which elements in the domain models are involved, the DSL users require a service which returns a set of instances being part of the domain model he has created. The service, specified in Figure 20, considers the explanation service for inconsistent ontologies. The reason for the model not being consistent depends on the Supervisor card type which excludes the VoiceInterface (cf. the metamodel in Figure 8). Although the Supervisor card is not part of the current configuration, the reasoner assumes this fact, since it is required for each Cisco_Configuration760x. In an open world reasoners assume missing facts in models as default, which are described in metamodels.

The service for inconsistent elements works on the explanation given in Figure 16. Since the model in Figure 19 has only one inconsistency the service for incon-

Name	*Type Suggestion Service*
Signature	Set<Class> typeSuggestion(Metamodel mm, Model m, Instance i)
Description	Project mm and m to an ontology o=project(mm, m). An Ecore class c defined in mm is put into the result set s, if o' is consistent. o' is defined by removing all class assertions for i from o, so that o' becomes consistent. If o' is still consistent after adding the class assertion ClassAssertion(i c) then c is put into the set s.

Fig. 21 Type Suggestion Service

Name	*SPARQL Query Answering*
Signature	ResultSet sparqlQuery(Ontology o, Query q)
Description	Evaluates the SPARQL query q on the ontology o and computes the answer a. If there is no answer a is null.

Fig. 22 SPARQL Querying Service

sistent elements returns only one set of elements. It returns configuration element of type Cisco_Configuration760x, the card element of type VoiceInterface and the respective slot element linking the card with the configuration.

Several strategies for ontology debugging and repairing are developed. A simple solution for handling inconsistencies and to repair domain models is to provide for one instance being involved in a class assertion axiom leading to an inconsistency a set of valid types. Replacing the type of the given instance by a suggested one leads to a consistent domain model. This is realized by a *type suggestion service*, specified in Figure 21.

5.3.3 Querying Services

DSL designers are able to define their own services by describing their functionality as queries. The framework allows for building SPARQL queries. In Figure 22 we specify a SPARQL query answering service. It gets as input a SPARQL query and returns an answer set.

Services defined by Queries DSL users which are not familiar with the concepts the DSL provides need suggestions. A DSL user having created an instance of Cisco-7603_Dev in his domain model wants to know which elements of which types can be connected via the hasConfiguration reference with the Cisco7603_Dev instance. The DSL user relies on a service which is implemented by the DSL designer, defining the signature of the service and a query defining the functionality of the service. The service is presented in Figure 23. It queries simultaneously metamodel and model and returns all

Name	*Target Type Service*
Signature	Set<Class> targetType(Metamodel mm, Model m, Reference r, Instance i)
Description	The target type service returns a list l of classes of mm which are valid types for an instance t in m where i refers via r to t. The service projects mm and m to an ontology o=project$_{GS_{OWL}}$(mm, g). It returns the query result of the following SPARQL query q="Query(ObjectPropertyAssertion(i r _:t) Type(_:t ?C))" by using the querying service with sparqlQuery(o, q).

Fig. 23 Target Type Service

types of target instances which are linked via an object property assertion (the projection of the instantiated reference) with the instance whose type offers the reference.

6 Related Work and Advantages

In the following, we group related approaches into two categories: approaches with formal semantics and constraints and approaches for model-based domain-specific language development.

Among approaches with formal semantics, one can use languages like F-Logic or Alloy to formally describe models. In [43], a transformation of UML+OCL to Alloy is proposed to exploit analysis capabilities of the Alloy Analyzer [44]. In [45], a reasoning environment for OWL is presented, where the OWL ontology is transformed to Alloy. Both approaches show how Alloy can be adopted for consistency checking of UML models or OWL ontologies. F-Logic is a further prominent rule language that combines logical formulas with object oriented and frame-based description features. Different works [46] have explored the usage of F-Logic to describe configurations of devices or the semantics of MOF models.

The integration in the cases cited above is achieved by transforming MOF models into a knowledge representation language (Alloy or F-logic). Thus, the expressiveness available for DSL designers is limited to MOF/OCL. Our approach extends these approaches by enabling DSL designers to specify class descriptions *à la* OWL together with MOF/OCL, increasing expressiveness.

Many other approaches are dealing with inconsistency management. In [39] the detection and resolution of inconsistencies in UML models with description logics (DL) is presented. Here, UML models are translated to (encoded as) DL knowledge bases, which allow for querying and reasoning. Similar to our approach of metamodeling, [39] presents an approach for checking structural inconsistencies of UML models with regard to its metamodel. Here, the DL knowledge base ABox represents the user model, where the TBox encodes the UML metamodel.

The integration between UML and DL in the approach cited above is established by a transformation which precisely defines which UML metamodel elements are translated to DL constructs. The transformation is adequate since [39] is dealing only with the UML language. Generally, for domain specific languages with arbitrary metamodels we propose an integration of the metametamodel with the one of an ontology language. Such an integration allows DSL designers to define their own constraints embedded within an arbitrary DSL metamodel.

There is quite a large number of works in the field of assisting modeling and debugging models. Different works [47,48] are dealing with tool support for creating feature models. Here the prescription of valid feature configuration is based on OCL constraints [49]. They provide the propagation of configuration choices, auto-completion of configurations and the debugging of incorrect configurations. [50] is dealing with model intelligence where existing constraint specifications in OCL [49] are used to query for valid endpoints of relationships in models. Such queries guide users toward correct solutions. [51,52] present a modeling editor for process models with syntax-based assistance. The editor provides the completion and correctness preserving of models. The syntax of the process modeling language is formally defined by graph grammars.

6.1 Advantages

In this section we concentrate on the advantages of metamodeling and reasoning approaches. We compare our solution with respective other approaches.

6.1.1 Constraint definition and formal semantics

Considering our comparison in Section 4 we may state that OWL 2, compared to usual metamodeling languages like Ecore, provides a rich set of primitives for the conceptual description of domains.

In contrast to Ecore, OWL 2 allows for refining classes by using additional axioms and several OWL 2 class and object property expressions. In Figure 8 a DSL designer builds a metamodel using integrated Ecore+OWL metamodeling language. Besides the structure of devices, the DSL designer also defines constraints in the metamodel. Therefore, he uses the integrated OWL 2 language, which is integrated with the existing KM3 syntax.

Furthermore, Ecore adopts the formal semantics of the integrated ontology language. Hence, elements in a Ecore-based metamodel have a formal meaning. A class in a metamodel describes a set of instances in the domain model. A reference in a metamodel describes a set of links between instances of a corresponding type. Attributes in a metamodel describe links between instances and values in the domain model.

6.1.2 Reasoning

For the suggestion of suitable concepts to be used (guidance) we use ontology reasoning technologies.

Technologies in the software modeling space mainly do not provide reasoning facilities. Reasoning on software models is principally enabled after a translation to a logic-based representation, e.g to Alloy [43], Description Logics [53], OWL 2 [54], or Object-Z [55,56]. When using such formal representations, one could reason on modelware models and formally prove properties through inference and make implicit knowledge of interest explicit [53]. Description Logics reasoners (such as Pellet [25], or Racer [57]) allow for joint as well as for separate sound and complete reasoning at both, the schema and the instance layers.

Nevertheless, we consider ontology technologies to exploit reasoning facilities. For example, OCL and respective tools do not allow for reasoning and inferring new facts based on the facts defined by language designer or user.

Schema Reasoning Schema reasoning considers all concept descriptions in ontologies independent of their instances. Based on the descriptions in the schema (TBox), schema reasoning allows for inferring new facts, which might be queried, e.g., using SPARQL, or detected by reasoning services.

Language designers creating metamodels may possibly be interested in computing the classes and references, which are not satisfiable, i.e., classes, which cannot be instantiated without the model becoming inconsistent. The following OCL constraint is not satisfiable because it simultaneously forbids and requires that instances of Configuration have a successor.

```
context Configuration
inv: not(hasSlot−>exists(t|t.oclIsKindOf(Slot))) and (hasSlot−>exists(t|t.
    oclIsKindOf(Slot)))
```

The tools available in the software modeling space do not allow for detecting the unsatisfiability of elements in metamodels. If we encode the OCL constraint as an axiom of an ontology, we will be able to reason and infer new facts. The listing below represents the OCL constraint above as class description being part of an OWL 2 ontology.

```
SubClassOf(Configuration ObjectIntersectionOf(ObjectComplementOf(
        ObjectSomeValuesFrom(hasSlot Slot)) ObjectSomeValuesFrom(
        hasSlot Slot)))
```

Facts for unsatisfiability of class expressions may be derived by queries or the given reasoning service presented in Section 3.5. The SPARQL query below queries for the fact of unsatisfiability. It uses the ontology with all additional facts inferred by a reasoner as data model:

```
SELECT DISTINCT ?t
WHERE {
        ?t rdfs:subClassOf owl:nothing
}
```

The satisfiability checking service presented in Section 3.5 considers an ontology and may infer the unsatisfiability based on all descriptions. The result of this check is that the OWL class Configuration described above is not satisfiable.

Schema+Instance Reasoning Description logics reasoners allow for joint reasoning on both schema and instance layer. Given an ontoware model describing TBox concepts and ABox instances, reasoners allow for classifying individuals to find their possible types described in the schema.

The following excerpt of an ontoware model depicts a TBox axiom stating that every device is linked via hasConfiguration with some configuration. The corresponding ABox consists of two individuals d and c. d is linked with c, which is of type Configuration.

```
// TBox axiom
EquivalentClasses(Device ObjectSomeValuesFrom(hasConfiguration
        Configuration))

// ABox axioms
Declaration(Individual(d))
ObjectPropertyAssertion(hasConfiguration d c)
Declaration(Individual(c))
ClassAssertion(c Configuration)
```

Based on a common description of schema and instance layer within one ontoware model, reasoners may infer new facts. Based on the descriptions in TBox and ABox, the SPARQL query below asks for all named types, that an individual i has. In the case of the individual d it returns the type Device.

```
SELECT DISTINCT ?t
WHERE {
        i rdf:type ?t
}
```

Using the reasoning service mentioned in Section 3.5, we are able to classify the individual d to find its possible type. The result is the class Device, since d fulfills all descriptions defined by the class expression.

6.1.3 Open World Assumption

The *Open World Assumption* (OWA) assumes incomplete information as default and allows for reasoning on incomplete models, while the *Closed World Assumption* (CWA) assumes all positive to be facts as part of the knowledge base (cf. Section 3.3).

For quantified expressions a reasoner assumes that a given individual is linked with other individuals. Although an individual is not linked with a given number (cardinality) of other individuals, a reasoner would assume by default that cardinality restrictions are fulfilled by assumed individuals in the domain.

The ontology below describes an incomplete knowledge base. In the TBox we define that each device must have a configuration and that each configuration must have a slot. In the ABox we declare the individuals d and c. d is linked with c, which is of type configuration.

```
// TBox axiom
EquivalentClasses(Device ObjectSomeValuesFrom(hasConfiguration
        Configuration))
EquivalentClasses(Configuration ObjectSomeValuesFrom(hasSlot Slot))

// ABox axioms
Declaration(Individual(d))
ObjectPropertyAssertion(hasConfiguration d c)
Declaration(Individual(c))
ClassAssertion(c Configuration)
```

Although the knowledge base is incomplete (c is not linked with a slot), reasoners are able to infer facts based on all descriptions in TBox and ABox. In the example above a reasoner infers that the individual d is of type device, because it is linked with some configuration although the configuration c is not complete (i.e., it is not linked with a slot).

7 Implementation

In this section we are going to present two implementations of an ontology-based framework for domain-specific modeling. In Section 7.1 we present the *TwoUse toolkit*, implemented at the WeST institute[4]. In Section 7.2 we present the *MOST workbench*, implemented by BOC[5]. While we have developed the TwoUse toolkit for the proof of concept of the approaches given in this paper, the MOST workbench provides a framework usable for industrial domain-specific modeling tasks.

[4] http://west.uni-koblenz.de/
[5] http://www.boc-group.com

From the point of ontology-based reasoning services provided by the two frameworks, they are equivalent. Comparing the usability of both frameworks, the MOST workbench provides a graphical user interface provided to DSL users. This is not available in the TwoUse toolkit.

7.1 The TwoUse Toolkit

In Figure 24 we depict a screenshot of the TwoUse toolkit. The TwoUse toolkit in general aims to filling the gap between MDE and ontology technologies. The TwoUse Toolkit is developed in the Eclipse Platform using the Eclipse Modeling framework [36] and is freely available for download on the project website[6]. In Figure 24 we see the view of a DSL designer modeling an integrated metamodel.

Integrated metamodeling is based on an integration of Ecore and OWL as explained in Section 5.2. We developed a textual concrete syntax combining the KM3 syntax and an adaption of the OWL Manchester syntax.

Ontologies are extracted by a projection service. All services for inconsistency management and guidance base on standard reasoning services provided by Pellet.

7.2 The MOST Workbench

In Figure 25 we depict a screenshot of the MOST workbench. The workbench is developed in the ADOxx platform[7] by the company BOC which is also industrial partner in the MOST project. The tool offers to develop domain-specific languages which may be coupled to different visual concrete syntaxes. A language like the physical device domain-specific language (PDDSL) is developed within the workbench by a DSL designer (or DSL expert). In Figure 25 we see the view of a DSL user modeling configurations of physical network devices.

The integrated modeling is based on the ADOxx generic graphical modelling editor. To enable integrated modeling with OWL descriptions, languages to be integrated as well as the OWL metamodel are defined using the ADOxx M3-metametamodel (called *ADOxx Meta^{2}Model* [58]). Then the integration has been performed following to the integration approach presented in Section 5.2.

[6] `http://code.google.com/p/twouse/`

[7] ADOxx® is the extensible, multi-lingual, multi-os, repository-based platform for the development of modeling tools of the BOC Group. ADOxx® is a registered trademark of the BOC Group, `http://www.boc-group.com`.

The reasoning services are implemented by a separate validation services component Comarch has implemented. The component projects models and metamodels to a format which is readable by OWL reasoners. In addition it implements domain-specific services for inconsistency management and guidance [59].

8 Analysis of the Approach

In this section, we establish the viability of our approach by a proof of concept evaluation. We analyze the approach with respect to the requirements of Section 2. At the end of this section we give comments Comarch provided after evaluating the approaches above.

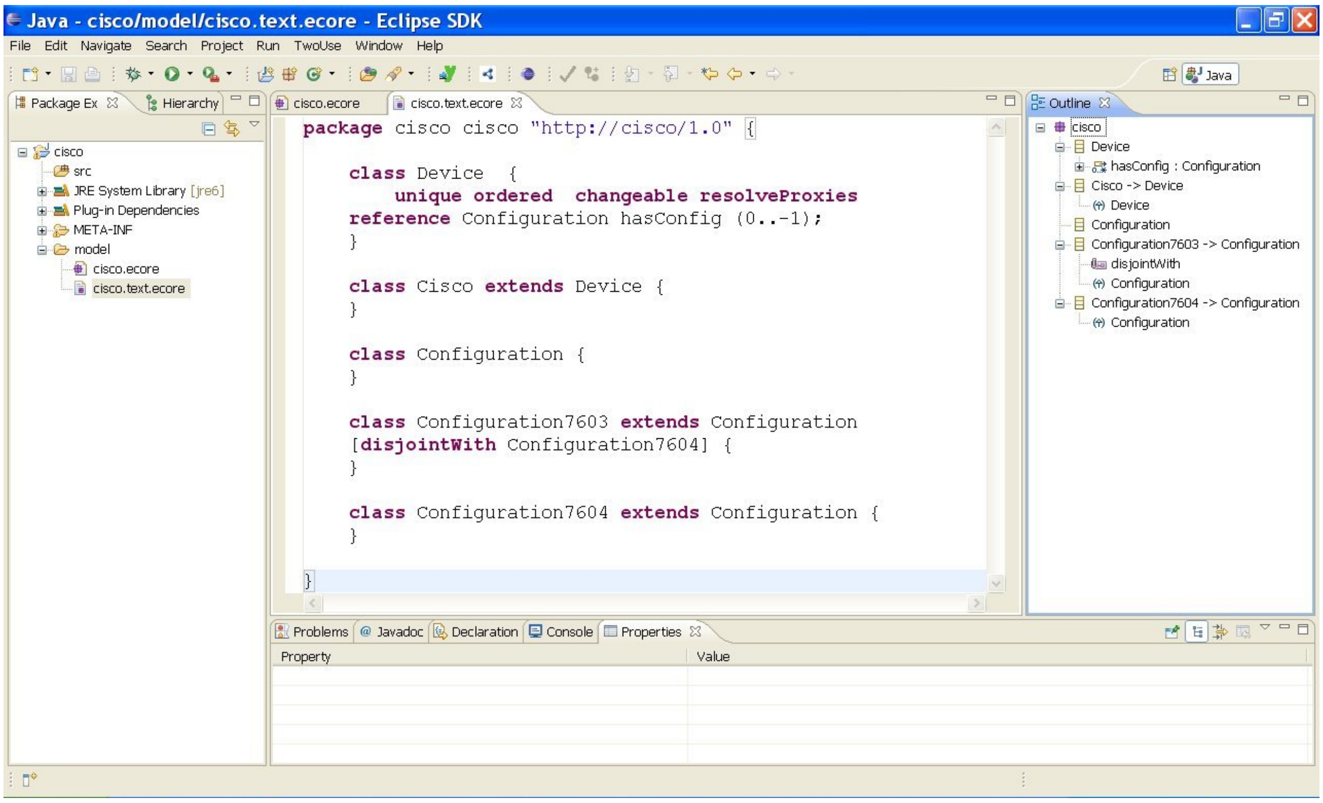
To address formal semantics and constraints (*requirement (1)*), we integrated the EMOF based metametamodel Ecore and its concrete syntax KM3 with OWL, allowing for a formal and logical representation of the solution domain. DSL designers count on an expressive language that allows for modeling logical constraints in DSL metamodels (*requirement (1)*). Reasoners check the consistency of metamodels and constraints and debugging services clarify the inferences (*requirement (4)*).

Formal model-theoretic semantics enable the usage of reasoning services to help DSL users to validate domain models, to detect inconsistencies, to diagnose them and to get suggestions how to repair the models (*requirement (3)*). DSL users may get suggestions of cards to be used in their DSL models based on the configuration of the device.

The expressiveness of OWL enables DSL designers to define classes and properties as equivalent. DSL designers may use this functionality to provide DSL users with different means for declaring objects (*requirement (5)*). A DSL user may describe a Cisco 7603 device in two different ways: by creating an instance of class Device with a configuration with three slots and a supervisor card in one slot; or by creating an instance of class Cisco7603_Dev.

The nature of the open world assumption enables progressive evaluation of domain models (*requirement (2)*). A DSL user may drag a new configuration into a DSL model with three slots but without any cards. The reasoner assumes that at least one of the mandatory cards is part of the configuration. Thus, DSL users can progressively evaluate parts of the domain model without firstly completing it .

DSL users call services defined by DSL designers by a query (*requirement (3)*). These queries are the interface between DSL users and reasoning services. For example, a DSL user may use a reasoning service which is



```
package cisco cisco "http://cisco/1.0" {

    class Device  {
        unique ordered  changeable resolveProxies
    reference Configuration hasConfig (0..-1);
    }

    class Cisco extends Device {
    }

    class Configuration {
    }

    class Configuration7603 extends Configuration
    [disjointWith Configuration7604] {
    }

    class Configuration7604 extends Configuration {
    }

}
```

Fig. 24 View of the DSL designer in the TwoUse Toolkit

implemented as query defined in the DSL metamodel and queries all classes that describe an object in the DSL model.

While solutions provided by DSL development environments for teaching DSL users are usually limited to help the creation of the example models, we have an interactive assisted solution by suggesting concepts and explaining inferences (*requirement (3)*). Nevertheless, addressing the aforementioned requirements lead us to new challenges as well as it demands to consider trade-offs between expressiveness and completeness and soundness, expressiveness and user complexity.

OWL is a logical language and it requires logical expertise of DSL designers. On the other side, the usage of OWL is encapsulated from DSL users. They only use operations to invoke different reasoning services which work on ontologies. Our approach extends the KM3 vocabulary with a controlled natural language as OWL textual concrete syntax to smooth the usage of logical assertions.

Comarch has evaluated the approach on ontology-based domain specific languages with an extension of scenario given in Section 2. Initial experiences showed benefits as well as limitations [59].

Benefits are the simple but expressive metamodeling language to design DSL metamodels and the DSL independent services provided to DSL designers and DSL users.

Limitations concern scalability and a hybrid syntax for DSL designers. Initial experiments revealed that scalability issues are not sufficiently handled within the prototype. The time needed to perform the guidance services is increasing dramatically with the size of the models. DSL designers at Comarch are domain experts who prefer graphical syntaxes to design DSL metamodels. However, the OWL annotations still should be modeled using the Manchester Syntax which is naturally readable.

Comarch showed in a user evaluation [60], that the development of DSL tools is less resource-consuming. The development time is reduced by 92% when comparing with the manually developed solution based on models only. The reason is that DSL tool developers are able to use the freely available basic services for reasoning and inconsistency management. Based on these services new domain-specific services are implemented easily. In addition Comarch showed for the particular case study that the modelling of domain models is less resource-consuming and less error-prone.

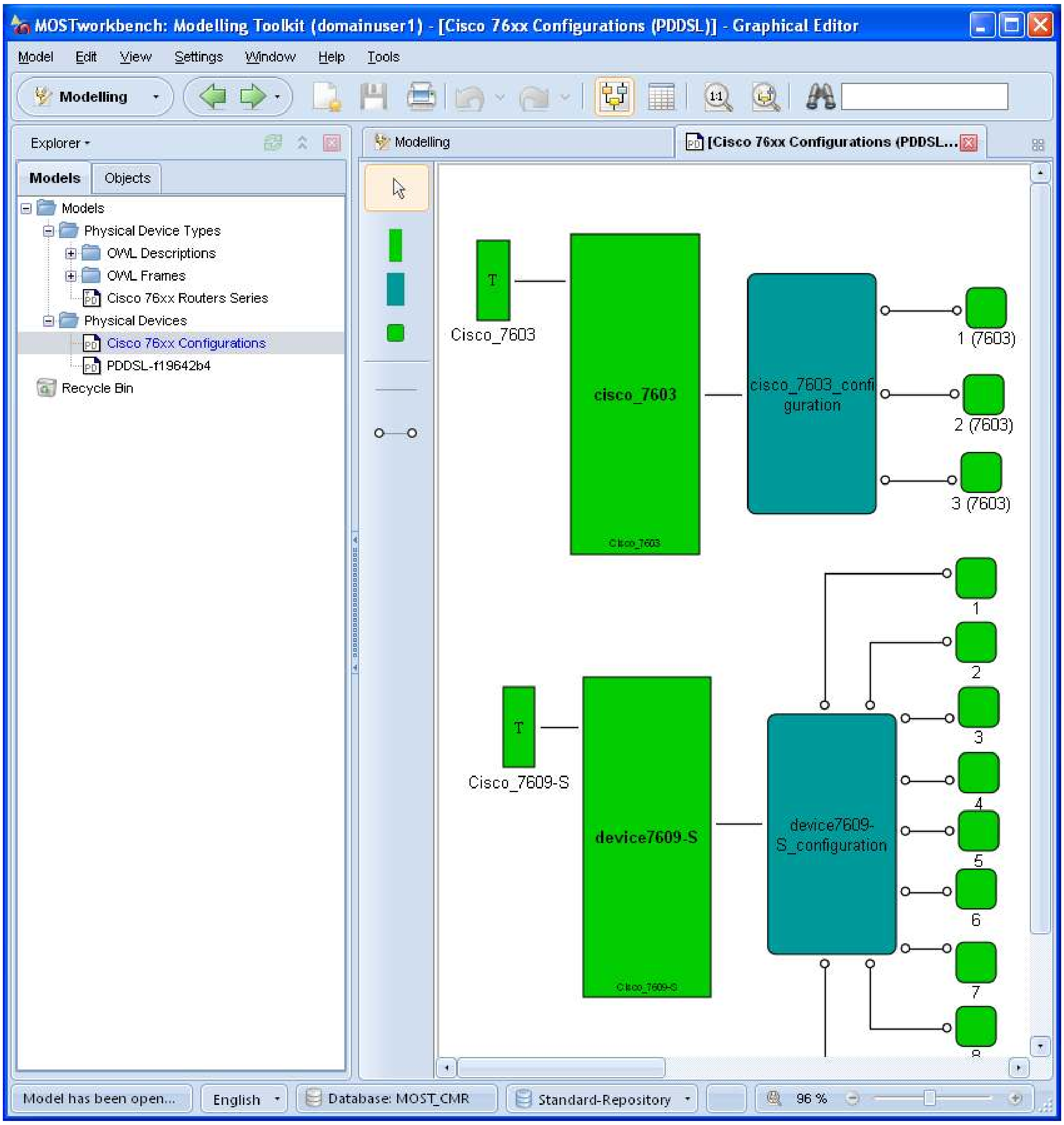


Fig. 25 View of the DSL user in the MOST Workbench

While our approach currently is only used within the company Comarch, it nevertheless is reusable for other stakeholders creating Ecore-based metamodels and models within the Eclipse Modeling Framework.

Since our new technological space represents the union of the Ecore software modeling space and the OWL 2 ontology space, for example several other language metamodels may be loaded.

Subsequently metamodels may be extended by additional constraints and expressions using the integrated ontology language.

On the other side, several technologies, e.g. for language evolution [61,62], which are developed within the Eclipse Modeling Framework may be reused. Additionally several other (textual or graphical) concrete syntaxes may be developed, e.g. using the Graphical Modeling Framework or EMFText.

9 Conclusion

In this paper, we presented an approach on how to address major challenges in the field of domain-specific languages with ontology languages and automated reasoning services.

We presented a new technological space which integrates Ecore and OWL at the M3-layer. The new metamodeling language is provided to DSL designers, who are able to specify metamodels with seamlessly integrated ontology-based expressions and axioms. DSL users simply use the DSL provided by DSL designers to create domain models.

The integrated ontology language gives the developed DSLs a formal model-theoretic semantics. The formal semantics enable applications of reasoning to help DSL designers and DSL users through the development and usage of DSLs. DSL designers benefit from constraint analysis. DSL users benefit from progressive verification, debugging support and assisted modeling.

The approach has been used and tested in the telecommunication domain under EU STReP MOST.

Acknowledgement. We like to thank Krzysztof Miksa from Comarch for providing the use cases. Further, we like to thank Prof. Dr. Jürgen Ebert for comments and improvement remarks. This work is supported by EU STReP-216691 MOST.

References

1. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling. John Wiley & Sons (2007)
2. Gray, J., Fisher, K., Consel, C., Karsai, G., Mernik, M., Tolvanen, J.P.: Panel - DSLs: the good, the bad, and the ugly. In: OOPSLA Companion '08, ACM (2008)
3. Langlois, B., Jitia, C.E., Jouenne, E.: DSL Classification. In: OOPSLA 7th Workshop on Domain Specific Modeling. (2007)
4. Czarnecki, K.: Generative Programming. PhD thesis, Department of Computer Science and Automation Technical University of Ilmenau (1998)
5. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.: The description logic handbook: theory, implementation, and applications. Cambridge University Press New York (2003)
6. Motik, B., Patel-Schneider, P.F., Horrocks, I.: OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax. `http://www.w3.org/TR/owl2-syntax/` (October 2009)
7. Tairas, R., Mernik, M., Gray, J.: Using ontologies in the domain analysis of domain-specific languages. In: Proceedings of the 1st International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering 2008. Volume 395 of CEUR Workshop Proceedings., CEUR-WS.org (2008)
8. Guizzardi, G., Pires, L.F., van Sinderen, M.: Ontology-based evaluation and design of domain-specific visual modeling languages. In: Proceedings of the 14th International Conference on Information Systems Development, Springer (2005)
9. Bräuer, M., Lochmann, H.: An ontology for software models and its practical implications for semantic web reasoning. In: Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications. Volume 5021 of LNCS., Springer (2008) 34–48
10. France, R.B., Rumpe, B.: Model-driven Development of Complex Software: A Research Roadmap. In: Proceedings of the Workshop on the Future of Software Engineering (FOSE). (2007) 37–54
11. Nuseibeh, B., Easterbrook, S., Russo, A.: Leveraging Inconsistency in Software Development. Software Development **33**(4) (2000) 24–29
12. Kurtev, I., Bézivin, J., Aksit, M.: Technological Spaces: An Initial Appraisal. In: CoopIS, DOA'2002 Federated Conferences, Industrial track, Irvine (2002)
13. Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: Formal Methods for Open Object-Based Distributed Systems. Volume 4037 of LNCS., Springer (2006) 171–185
14. ATLAS Group LINA & INRIA, Nantes: KM3: Kernel MetaMetaModel - Manual version 0.3. (2005)
15. Parreiras, F.S., Staab, S.: Using ontologies with uml class-based modeling: The twouse approach. Data & Knowledge Engineering **69**(11) (2009) 1194–1207
16. Walter, T., Parreiras, F.S., Staab, S.: OntoDSL: An Ontology-Based Framework for Domain-Specific Languages. In: Model Driven Engineering Languages and Systems, 12th International Conference, MODELS. Volume 5795 of LNCS., Springer (2009) 408–422
17. Walter, T., Parreiras, F.S., Staab, S., Ebert, J.: Joint Language and Domain Engineering. In: Proceedings of European Conference Modelling Foundations and Applications. Volume 6138 of LNCS., Springer (2010) 321–336
18. Guarino, N., Oberle, D., Staab, S.: What Is an Ontology? Handbook on Ontologies (2009) 1–17
19. Miksa, K., Kasztelnik, M.: Definition of the case study requirements. Deliverable ICT216691/CMR/WP5-D1/D/PU/b1, Comarch (2008) MOST Project, http://www.most-project.eu/.
20. Farrugia, J.: Model-theoretic semantics for the web. In: WWW '03: Proceedings of the 12th international conference on World Wide Web, New York, NY, USA, ACM (2003) 29–38
21. Motik, B., Patel-Schneider, P.F., Grau, B.C.: OWL 2 Web Ontology Language Direct Semantics. `http://www.w3.org/TR/2009/REC-owl2-direct-semantics-20091027/` (October 2009)
22. Horridge, M., Knublauch, H., Rector, A., Stevens, R., Wroe, C.: A practical guide to building OWL ontologies using the protégé-OWL plugin and CO-ODE tools. Technical report (2004)
23. Donini, F.M., Lenzerini, M., Nardi, D., Nutt, W., Schaerf, A.: An epistemic operator for description logics. Artificial Intelligence **100**(1-2) (1996) 225–274

24. Grimm, S., Motik, B.: Closed World Reasoning in the Semantic Web through Epistemic Operators. In: Proceedings of the 1st OWL Experiences and Directions Workshop (OWLED-2005). Volume 188 of CEUR Workshop Proceedings., CEUR-WS.org (2005)
25. Sirin, E., Parsia, B., Grau, B.C., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL Reasoner. Web Semantics: Science, Services and Agents on the World Wide Web **5**(2) (2007) 51–53
26. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. `http://www.w3.org/TR/sparql11-query/` (June 2010)
27. Polleres, A.: SPARQL 1.1: New Features and Friends (OWL2, RIF). In: Web Reasoning and Rule Systems. Volume 6333 of LNCS., Springer (2010) 23–26
28. Sirin, E., Parsia, B.: SPARQL-DL: SPARQL Query for OWL-DL. In: Proceedings of the 3rd OWL Experiences and Directions Workshop (OWLED-2007). Volume 258 of CEUR Workshop Proceedings., CEUR-WS.org (2007)
29. Kremen, P., Sirin, E.: SPARQL-DL Implementation Experience. In: Proceedings of the 4th OWL Experiences and Directions DC Workshop (OWLED-DC-2008). Volume 496 of CEUR Workshop Proceedings., CEUR-WS.org (2008)
30. Glimm, B., Parsia, B.: SPARQL 1.1 Entailment Regimes. `http://www.w3.org/TR/2010/WD-sparql11-entailment-20100126/` (January 2010)
31. Schneider, M.: SPARQLAS – Implementing SPARQL Queries with OWL Syntax. In: Proceedings of the 3rd Workshop on Transforming and Weaving Ontologies in Model Driven Engineering. Volume 604 of CEUR Workshop Proceedings., CEUR-WS.org (2010)
32. OMG: Meta Object Facility (MOF) Core Specification. Object Management Group. (January 2006)
33. Ebert, J., Riediger, V., Winter, A.: Graph Technology in Reverse Engineering, The TGraph Approach. In: Proceedings of Workshop Software Reengineering (WSR). Volume 126 of LNI., GI (2008) 67–81
34. Djuric, D., Gasevic, D., Devedzic, V.: Ontology Modeling and MDA. Journal of Object technology **4**(1) (2005) 109–128
35. Gaševic, D., Djuric, D., Devedzic, V., Damjanovic, V.: Approaching OWL and MDA through Technological Spaces. In: Proceedings of the 3rd Workshop in Software Model Engineering (WiSME 2004). (2004)
36. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework (2nd Edition). Addison-Wesley (2008)
37. Horridge, M., Patel-Schneider, P.F.: OWL 2 Web Ontology Language Manchester Syntax. `http://www.w3.org/TR/owl2-manchester-syntax` (October 2009)
38. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende., C.: Derivation and Refinement of Textual Syntax for Models. In: Proceedings of European Conference on Model-Driven Architecture Foundations and Applications. Volume 5562 of LNCS., Springer (2009) 114–129
39. Van Der Straeten, R.: Inconsistency Management in Model-driven Engineering. An Approach using Description Logics. PhD thesis, Vrije Universiteit Brussel, Belgium (2005)
40. Finkelstein, A., Spanoudakis, G., Till, D.: Managing Interference. In: ISAW '96: Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, ACM (1996) 172–174
41. Spanoudakis, G., Zisman, A.: Inconsistency Management in Software Engineering: Survey and Open Research Issues. Handbook of Software Engineering and Knowledge Engineering **1** (2001) 329–380
42. Kalyanpur, A.: Debugging and Repair of OWL Ontologies. PhD thesis, University of Maryland, College Park (2006)
43. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: Proceedings of Model Driven Engineering Languages and Systems, MoDELS 2007. Volume 4735 of LNCS., Springer (2007) 436–450
44. Jackson, D.: Software Abstractions: logic, language, and analysis. The MIT Press (2006)
45. Wang, H., Dong, J., Sun, J., Sun, J.: Reasoning support for Semantic Web ontology family languages using Alloy. Multiagent and Grid Systems **2**(4) (2006) 455–471
46. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The missing link of MDA. In: Proceedings of 1st International Conference on Graph Transformation. Volume 2505 of LNCS., Springer (2002) 90–105
47. Czarnecki, K., Kim, C.H.P.: Cardinality-based feature modeling and constraints: A progress report. In: Proceedings of International Workshop on Software Factories at OOPSLA'05. (2005)
48. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: Proceedings of the 5th international conference on Generative programming and component engineering, ACM (2006) 211–220
49. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley (2003)
50. White, J., Schmidt, D.C., Nechypurenko, A., , Wuchner, E.: Model intelligence: an approach to modeling guidance. UPGRADE **9**(2) (2008) 22–28
51. Mazanek, S., Minas, M.: Business process models as a showcase for syntax-based assistance in diagram editors. In: Proceedings of Model Driven Engineering Languages and Systems (MoDELS). Volume 5795 of LNCS., Springer (2009) 322–336
52. Mazanek, S., Maier, S., Minas, M.: Auto-completion for Diagram Editors based on Graph Grammars. In: Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computings, IEEE (2008) 242–245
53. Berardi, D., Calvanese, D., Giacomo, G.D.: Reasoning on UML class diagrams. Artificial Intelligence **168**(1-2) (2005) 70–118
54. Walter, T., Schwarz, H., Ren, Y.: Establishing a Bridge from Graph-based Modeling Languages to Ontology Languages. In: Proceedings of 3rd Workshop on Transforming and Weaving Ontologies in Model Driven Engineering (TWOMDE). Volume CEUR of 604., CEUR-WS.org (2010)
55. Evans, A.S.: Reasoning with UML class diagrams. In: Proceedings of 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques, IEEE Computer Society (1998) 102–113
56. Ebert, J., Winter, A., Dahm, P., Franzke, A., Süttenbach, R.: Graph Based Modeling and Implementation with EER / GRAL. In: Proceedings of Conceptual Modeling - ER'96. Volume 1157 of LNCS., Springer (1996) 163–178
57. Haarslev, V., Möller, R.: Description of the racer system and its applications. In: Proceedings of Description Logics Workshop. Volume 49 of CEUR Workshop Proceedings., CEUR-WS.org (2001)

58. Bartho, A., Zivkovic, S.: Modeled software guidance/engineering processes and systems. Deliverable ICT216691/TUD/WP2-D2/D/PU/b1.00, Technial University Dresden, BOC (2009) MOST Project, http://www.most-project.eu/.
59. Miksa, K., Sabina, P., Zivkovic, S.: First demonstrator and report on experiences. Deliverable ICT216691/CMR/WP5-D3/D/PU/b1, Comarch (2010) MOST Project, http://www.most-project.eu/.
60. Miksa, K.: Evaluation of case study. Deliverable ICT216691/CMR/WP5-D4/D/RE/b1, Comarch (2011) MOST Project, http://www.most-project.eu/.
61. Kappel, G., Wimmer, M., Retschitzegger, W., Schwinger, W.: Leveraging Model-Based Tool Integration by Conceptual Modeling Techniques. In: The Evolution of Conceptual Modeling. Volume 6520 of LNCS., Springer (2011) 254–284
62. Kolovos, D.S., Paige, R.F., Polack, F.: Merging Models with the Epsilon Merging Language (EML). In: Proceedings of International Conference on Model Driven Engineering Languages and Systems(MoDELS). Volume 4199 of LNCS., Springer (2006) 215–229