

# Joint Language and Domain Engineering

Tobias Walter<sup>1,2</sup>, Fernando Silva Parreiras<sup>1</sup>, Steffen Staab<sup>1</sup>, and Jürgen Ebert<sup>2</sup>

<sup>1</sup> Institute for Web Science and Technology, University of Koblenz-Landau  
Universitätsstrasse 1, Koblenz 56070, Germany

<sup>2</sup> Institute for Software Technology, University of Koblenz-Landau  
Universitätsstrasse 1, Koblenz 56070, Germany  
{walter, parreiras, staab, ebert}@uni-koblenz.de

**Abstract.** In domain-specific development model-driven development environments play an important role. Most of these environments only provide support for language engineering, but do not consider the second dimension which is concerned with domain engineering. In this paper, we join the concerns of language engineering and domain engineering towards a new comprehensive approach of domain-specific development. It allows domain designers to build domain models containing both, types and instances, and it allows language designers for defining language metamodels. Furthermore, based on the integrated description logics the environment provides services for productive modeling in domain and language engineering.

## 1 Introduction

Today, domain-specific development is based on model-driven development (MDD) [1]. In [2] we have presented an environment called *OntoDSL* which allows for developing and using description logic-based domain-specific languages. The environment supports both language designers and language users. A *language designer* provides domain-specific languages (DSL) to language users by defining an abstract syntax in the form of a metamodel, a concrete syntax (e.g. of a textual or visual kind) and semantics. All three steps are related to *language engineering*. The *language user* makes use of the DSL and builds *domain models* by creating instances of elements like classes and associations of the metamodel.

In *OntoDSL*, we consider metamodel hierarchies to describe the specification and the use of DSLs. At the *M3 layer*, a metametamodel is defined. At the *M2 layer*, the language is specified by defining a metamodel. Its elements are instances of elements in the metametamodel. At the *M1 layer*, the specified language can be used by creating a domain model, which is a linguistic instance of the DSL metamodel. For example, a class *Device* at the M2 layer allows for creating linguistic instances like *cisco* at the M1 layer.

In contrast to language engineering, which is based on hierarchies related by linguistic instantiation, another important dimension of model-driven development is the engineering of the domain, where hierarchical layers are related by ontological instantiation. [3].

In *domain engineering* the role of a *domain designer* is involved. A domain designer has the task to formally describe an existing or new domain. The result of domain engineering is a domain model which consists of both *domain instances* and *domain types* which classify the instances. Since a domain designer can create both instances and types in one domain model and can assign types to instances by a *hasType*-relationship, domain engineering requires ontological instantiation. For example, the domain designer wants to explicitly define that the domain instance `cisco7603` has the domain type `Cisco`.

Our work is based on the work of Atkinson and Kühne [3, 4]. They claim, that meta-modeling is an essential foundation for model-driven development but does not meet all the technical requirements for MDD environments. However, it can be extended to provide the full support for language engineering with an *instanceOf*-relation and domain engineering with a *hasType*-relation.

The MOF (Meta Object Facility) language [5] and its derivatives mainly provides linguistic instantiation where some parts of language engineering are supported. The ontological *hasType*-relation may be defined, but it would be just a simple UML association. Its meaning would remain implicit and would not be recognized and supported by the tools. Furthermore, the use of DSLs, where a language user builds domain models containing linguistic instances of concepts in the DSL metamodel, is separated from domain engineering, where a domain designer creates domain models, which consist of domain type and instance definitions.

## 1.1 Challenges

To accomplish the definition of a *hasType*-relation with explicit semantics and a joint design of domain models, using a DSL together with the facilities of domain engineering, we have to deal with the following challenges:

1. **Explicit modeling ontological and linguistic instantiation relationship:** One challenge in today's model-driven development environments is that they should allow for explicitly modeling both, ontological and linguistic, instantiation relationship to support both, domain and language engineering [3]. To create elements in a domain model domain designers and language users require a (domain-specific) language represented by a metamodel. This language should prescribe the design of domain models and provide a linguistic instantiation mechanism for designing types and instances in domain models. In addition domain designers require explicit modeling of an ontological instantiation relationship. It allows for assigning a domain type to domain instances in the domain model.
2. **Combination of Language Engineering and Domain Engineering:** A second challenge is related to the joint use of linguistic and ontological instantiation. The problem in using pure DSLs which only allow for creating linguistic instances of elements in the metamodel is a lack of flexibility in dynamically extending the set of domain types in domain models [3]. Domain designers might identify domain types and instances, where language designers formalize them by creating a DSL for language users to create models describing e.g. products or components for software systems [6]. Domain designers call for the capability iteratively to define

or extend the set of domain types for modeling domain instances. This requires the simultaneous definition of types and instances in one domain model. Here the need of an appropriate language metamodel is needed which provides concepts to allow for defining both, types and instances. On the other side, since pure domain engineering allows to create arbitrary domain types, different domain models of the same domain could have different types which do not fit together. Here some prescribing language for domain models can be necessary to make them comparable and capable of being integrated.

3. **Services and Constraints:** The validity of domain models is an important challenge. If models are invalid, domain designers and language 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. The MDD environment should be able to provide suggestions to language users and domain designers. In the case of building domain models, language users normally start the modeling with general and abstract concepts, since they have not the complete knowledge of all constructs provided by the DSL or they want to keep variability in extending the model [7]. Hence they want to classify conforming model elements according to concept descriptions in the language metamodel. In the case of domain engineering, domain designers want to classify existing domain instances. Since often domain instances exist without any domain type, domain designers want to get suggestions of possible types automatically. To define the validity an appropriate constraint language is needed. Language designers have to define constraints to restrict the use of concepts in the metamodel. Domain designers have to define constraints to refine the domain description. Furthermore, constraints for domain designers must cover both, instance and type layer.

This paper is structured as follows: In section 2, we sketch the application context and show the differences between linguistic and ontological instantiation. We present a running example for joint language and domain engineering illustrating the three challenges. After some foundations of description logics and its need in software engineering in section 3, we present the architecture of an environment which provides both, language engineering with linguistic instantiation and domain engineering with ontological instantiation in section 4. At the end of this paper, we compare our approach with the challenges (section 5) and with related work (section 6).

## 2 Running Example

In this section, we first start with an introduction of the application context which gives an idea of the different dimensions of metamodeling. Afterwards we present a simple running example where we show a domain-specific language and its domain model which allows for defining an ontological instantiation relation.

### 2.1 Application Context

Generally, *language designers* using MDD environments require the facility to define the abstract syntax, at least one concrete syntax and the semantics of the language to be

designed [3]. The abstract syntax can be defined by a metamodel. The concrete syntax can be specified by textual or visual notations. Semantics may be defined by a natural language specification or may be captured (partially) by logics (e.g. description logics [8, 9]).

From the language engineering perspective and with regard to figure 1(a) linguistic instantiation supplies a linear metamodeling hierarchy [4]. The metamodel is instantiated by the language designer to define the metamodel. The metamodel itself is instantiated by a language user to build domain models. For example, the **metatype** elements and the **metainstance** elements in the metamodel are linguistic instances of the metamodel element **class**. **type** elements and **instance** elements in the domain model are linguistic instances of the **metatype** element and **metainstance** element at the M2 layer. In figure 1(b) the elements in figure 1(a) are exemplified by concrete model elements from the domain of network devices (cf. section 2.2). Here, **Device** is a metatype and a possible linguistic instance of **Device** is **Cisco**. On the right side of figure 1(b), we have **DeviceInstance** as a metainstance. A linguistic instance of **DeviceInstance** is **cisco7603** at the M1 layer.

At the M1 layer, a domain designer is able to define at least two ontological layers (*O2* and *O1*) within his domain model. He is able to define **type** elements (at *O2*), corresponding **instance** elements (at *O1*) and connecting them by an ontological *hasType*-relation. The relation itself is defined in the metamodel which strongly prescribes the design of domain models (e.g. types cannot be connected to other types via *hasType*). The M0 layer represents the real world objects, e.g. concrete devices and its categories. With regard to figure 1(b) **Cisco** is a domain type which has a domain instance called **Cisco7603** via an ontological *hasType* relation. The *hasType* relation between a **Device** and a **DeviceInstance** is defined at the M2 layer. Furthermore, a domain designer can specialize domain types by creating *subclass*-relationships, or vice versa subsume given domain types by one super type. For example, domain type **Cisco** in figure 1(b) could have the specialization **CiscoWAN** or **CiscoLAN**.

While metamodels of DSLs on the one side are used to prescribe the design of domain models, on the other side, initial domain models are used to extract the metamodel of a language. At first, a domain designer would create types and instances in his domain model. A language designer considers the domain model and extracts relevant concepts for the metamodel [10].

In the following, we are going to present an example which is provided by one of the industrial partners of the MOST project<sup>3</sup>. This example exemplifies all *challenges* (1) to (3) introduced in section 1. With regard to figure 1 the following example depicts how the elements **metatype**, **metainstance**, **type** and **instance** are concretely defined in a metamodel and a domain model and how the instantiation relations are modeled.

## 2.2 Example

*Comarch*<sup>4</sup>, a polish IT company specialized in software for telecommunication providers, uses different model-driven methods for software development where dif-

<sup>3</sup> <http://www.most-project.eu>

<sup>4</sup> <http://www.comarch.com>

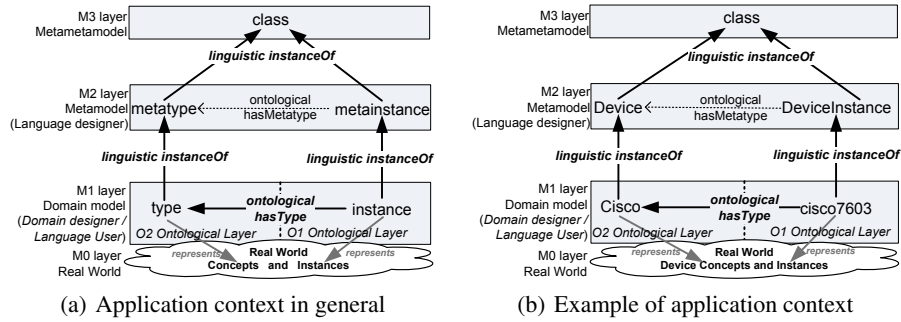


Fig. 1. Linguistic and ontological metamodeling

ferent kinds of domain-specific languages (DSLs) are deployed during the modeling process. Some of the tools that Comarch develops for telecommunication providers are dealing with modeling physical network devices. This is a domain-specific task, since different configurations of network devices have to be modeled. The following language metamodel (figure 2) and domain model (figure 3) are designed by using a textual concrete syntax which is based on an extended KM3-syntax [11].

**1. Explicit modeling ontological and linguistic instantiation relationship:** The domain of physical network devices can be described by a simple DSL, which provides the core metatypes like *Device*, *Slot* and *Card*. Comarch language designers want to provide the facilities of domain engineering to language users and domain designers to create domain models at the M1 layer. Thus, they have to provide a language which allows for creating domain types and instances in domain models. Furthermore, the ontological instantiation relation must be explicitly defined. Metatypes together with the connecting *metareferences* describe the general structure of a network device and are defined in an M2 metamodel which is depicted in figure 2. In the same metamodel the Comarch language designer defines *metainstances* using the *metainstance*-keyword. Here the ontological instantiation relation is defined by the *hasMetatype*-keyword.

A domain model is depicted in figure 3 and consists of linguistic instances of model elements in the metamodel. Here both domain types and instances are defined using the *type*- and *instance*-keyword. Using the *instanceOf*-keyword each domain type and domain instance can be defined as a linguistic instance of a corresponding metatype and metainstance. For example, domain type *Cisco* is a linguistic instance of *Device*, while *supervisor720* is a linguistic instance of *CardInstance*.

A mandatory task in creating domain models is the definition of an explicit *hasType*-relation between instances and domain types. In the example in figure 3, a domain designer wants to use the *hasType*-keyword to define that the ontological instance *supervisor720* has the named type *CiscoCard*. References like *hasSlot* in the type definitions on the one side represent links which are linguistic instances of corresponding references in the metamodel, on the other side, they define new references for links between ontological instances.

Furthermore, constraints based on description logics [9] should be defined in the metamodel. For example, in figure 2 an *equivalentWith*-axiom is used to define that

each device instance must be linked with at least some card via some slot, which cannot be defined by cardinalities, because slots optionally could be empty.

**2. Combination of Language Engineering and Domain Engineering:** To ensure the correctness of domain models Comarch wants to prescribe the design of each domain model. The core domain should be described by a DSL which is used by domain designers und language users to build domain models. So far, the DSLs designed by Comarch do not allow for creating both types and instances in the domain model. To accomplish the prescription of the design of domain models, a Comarch language designer wants to describe DSLs in a way like it is done in figure 2. Here the metamodel of a DSL is depicted which allows for describing the core domain of physical network devices, but as well distinguishes between domain types and instances.

Language users and domain designers get this metamodel and can create linguistic instances, which build the domain model depicted in figure 3. Thus every domain model can consist of domain types (using the `type`-keyword) and corresponding instances (using the `instance`-keyword). Furthermore, each complete device has to follow the given structure of the order *device-slot-card*, and has to contain at least one card, which is prescribed by the DSL. Without a DSL that prescribes the design of domain models a second domain designer would be able to create domain models which describe devices containing elements in the order *device-card*. Such models of the same domain would not be comparable with other domain models and capable of being integrated.

---

```

1 metatype Device {
2   metareference hasSlot [1-*]: Slot;
3 }
4 metatype Slot {
5   metareference hasCard [0-*]: Card;
6 }
7 metatype Card { }
8
9 metainstance DeviceInstance hasMetatype Device, equivalentWith restrictionOn
   hasSlot with some restrictionOn hasCard with some Card {
10  metalink hasSlot [1-*]: SlotInstance;
11 }
12 metainstance SlotInstance hasMetatype Slot {
13  metalink hasCard [0-*]: CardInstance;
14 }
15 metainstance CardInstance hasMetatype Card { }

```

---

**Fig. 2.** M2 metamodel of the core DSL

**3. Services and Constraints:** Language designer and domain designer at Comarch want to define constraints in their language metamodels and domain models. Using a metamodeling language (like KM3 [11]) and in addition some constraint language (like OCL) as yet, maybe would not help Comarch, since the designers want to define constraints that cover at least two layers - one type layer and one instance layer.

For example, a domain designer restricts the domain type `CISCO` by defining that it must be connected within the domain model in figure 3 with at least one `Supervisor-card` via a slot. The type `Supervisor` is equivalent to a set of two domain instances,

namely `supervisor360` and `supervisor720`. Here a constraint is used, which covers both layers for types and instances. In figure 3, below the definition of domain types, the definition of domain instances occurs. Here the instance `cisco7604` has an anonymous type which restricts that it must be connected with some instance of `Supervisor`.

Language users want to have services for validating domain models with regard to the metamodel. Domain designers also want to validate their domain models and check the consistency of domain instances with regard to the domain types. Furthermore, they require classification of domain instances with suggestions of suitable types to be assigned to instances in the domain model. So far, Comarchs MDD environments do not support validation and classification services cannot be realized based on the current domain models.

The domain instance `cisco7604` leads to an inconsistency. As an explanation, an MDD environment should return a debugging relevant fact which gives the information that a link to a supervisor card is missing. Since not every instance in the domain model is assigned to a domain type, domain designers require suggestions of suitable types. For example, they want to have `CISCO` as possible domain type of `cisco7603` (because it is connected via a slot with some supervisor card).

---

```

1  type Cisco instanceOf Device equivalentWith restrictionOn hasSlot with some
   restrictionOn hasCard with some Supervisor {
2      reference hasSlot [1-*]: CiscoSlot;
3  }
4  type CiscoSlot instanceOf Slot {
5      reference hasCard [0-*]: CiscoCard;
6  }
7  type CiscoCard instanceOf Card { }
8  type HotSwappableOSM instanceOf Card, extends CiscoCard { }
9  type Supervisor instanceOf Card, equivalentWith oneOf(supervisor720, supervisor360
   ) { }
10
11 instance cisco7603 instanceOf DeviceInstance{
12     hasSlot slot1;
13 }
14 instance cisco7604 instanceOf DeviceInstance, hasType restrictionOn hasSlot with
   some restrictionOn hasCard with some Supervisor {
15 }
16 instance slot1 instanceOf SlotInstance, hasType CiscoSlot {
17     hasCard supervisor360;
18 }
19 instance supervisor720 instanceOf CardInstance, hasType CiscoCard { }
20 instance supervisor360 instanceOf CardInstance, hasType CiscoCard { }

```

---

**Fig. 3.** M1 domain model containing types and instances

### 3 Description Logics-based Metamodeling

Description logics[9] are a family of logics for concept definitions that allows for separate as well as for joint sound and complete reasoning at the model and at the instance level given the definition of domain concepts.

OWL2, the web ontology language, is a W3C recommendation with a very comprehensive set of constructs for concept definitions [12] and represents a concrete implementation of a description logic.

In this paper, we use OWL for specifying classes, properties and individuals of a domain, instead of OCL, the Object Constraint Languages [13], which is an expression language to specify constraints for UML diagrams.

### 3.1 Description Logics for language and domain engineering

The domain-specific language engineering process can be divided into different phases [14]: *analysis*, *design* and *implementation*. In this paper, we mainly concentrate on the *design phase* of DSLs. Here a metamodel of the language is specified, together with concrete syntax and semantics. MOF-like metamodels usually describe the metamodels. The semantics of MOF-based metamodels is limited in comparison to the ones of description logics, and the latter one provides a better support for reasoning than MOF-based languages [15]. Description logics-based approaches lead to formal domain-specific metamodels that may be exploited for a variety of services, from consistency checking to semi-automatic engineering and to explanations [2].

As described in [16] the process of domain engineering can be divided into three main parts: *domain analysis*, *infrastructure specification* and *infrastructure implementation*. The domain analysis phase considers the identification and analysis of domain knowledge to be reused in software engineering. The result of domain analysis is a formal *domain model* of the problem domain. In this paper, we consider the task of creating domain models. As described in [16], ontologies can help in the language specification by capturing the problem domain, conceptualizing it, and later constraining the interpretation by further formal axioms.

### 3.2 Example

In the following, we consider the running example again and define the domain model presented in figure 3 as a knowledge base using description logics in figure 4.

The axioms (1) to (4) define the description logics TBox. The TBox is used to specify concepts (corresponding to classes in UML) which denote sets of individuals and roles (corresponding to associations in UML) which define binary relations between individuals. At first the concept `CISCO` is defined as an anonymous class which demands that each individual of `CISCO` is connected with some individual of type `Supervisor` via the `hasSlot`- and `hasCard`-role (1). In (2) the concepts `CiscoSlot` and `CiscoCard` are defined as a subclass of *top* ( $\top$ ). The *top*-concept is the common super type of all defined concepts in the knowledge base and captures all individuals in the domain. In (3) the `HotSwappableCard` is defined as a subclass of `CiscoCard`. In (4) the `Supervisor` concept is defined as an enumeration of the individuals `supervisor720` and `supervisor360`.

The axioms (5) to (10) define the description logics ABox. Here the concrete knowledge is asserted defining individuals of concepts and linking them using the roles defined in the TBox. In (5) the individual `cisco7603` is defined but has no direct type. In (6) the individual `cisco7604` is defined, which has an anonymous type. It defines,

that the individual must be connected by the *hasSlot* and *hasCard*-role with some individual of type *Supervisor*. Furthermore, all necessary individuals for slots and cards are defined (7, 8). Using role assertions all the individuals are linked and represent a concrete configuration of a *Cisco7603* device (9, 10).

$$\begin{aligned}
Cisco &\equiv \exists hasSlot.(\exists hasCard.Supervisor) & (1) \\
CiscoSlot, CiscoCard &\sqsubseteq \top & (2) \\
HotSwappableOSM &\sqsubseteq CiscoCard & (3) \\
Supervisor &\equiv \{supervisor720, supervisor360\} & (4) \\
cisco7603 &\in \top & (5) \\
cisco7604 &\in \exists hasSlot.(\exists hasCard.Supervisor) & (6) \\
slot1 &\in Slot & (7) \\
supervisor720, supervisor360 &\in Card & (8) \\
(cisco7603, slot1) &\in hasSlot & (9) \\
(slot1, supervisor360) &\in hasCard & (10)
\end{aligned}$$

**Fig. 4.** Description logics knowledge base representing the domain model from figure 3

### 3.3 Open and Closed World Assumption

While the underlying semantics of MOF-based class modeling adopts the closed world assumption (CWA), description logics adopt the open world assumption (OWA) by default. Traditional design of domain models is based on the closed-world assumption where the elements in the model are known and unchanging. The open world assumption assumes incomplete information as default and allows for validating incomplete domain models which are still in the design phase. However, research in the field of combining description logics and logic programming [17] provides solutions to support description logics-based reasoning with the closed world assumption as well [18]. Thus we are able to switch between reasoning with OWA and CWA.

Since description logics are useful in domain engineering for joint reasoning at the type layer and instance layer, for handling incomplete domain models and in language engineering for validating domain models with regard to its metamodel, we propose to develop language metamodels (cf. figure 2) and domain models (cf. figure 3) with embedded description logics-based constraints in an integrated manner. Our intention is to allow domain and language designers to create domain models and metamodels with the language they are familiar with as much as they can and selectively annotate elements with simple description logics-based constraints.

## 4 Metamodeling with Linguistic and Ontological Instantiation

In this section, we will present the approach and architecture which provide linguistic and ontological metamodeling. In section 4.1, first we present the overall approach and the architecture. In section 4.2 we present an excerpt of the integrated metamodel, and give an idea how it relates to a concrete syntax. In section 4.3 we present the different kinds of services which are provided by the environment.

### 4.1 Overall Approach

Figure 5 presents a multi-layered architecture depicting the *OntoDSL*-environment usable for language engineering and extended with new functionalities for domain engineering.

Core of the environment is the *Ontology-based MetaModeling Language (OntoM2L)* at the M3 layer, whose abstract syntax is described by an integrated metamodel. It consists of an (extended) *KM3 metamodel* [11] integrated with an *OWL2 metamodel* [12], which implements a description logic. An excerpt of the metamodel is depicted in figure 6. Linguistic instances of the integrated metamodel lie at the M2 layer. Here the environment provides the facility for language engineering and allows for building DSL metamodels. These metamodels can contain the definition of domain metatypes and metainstances. The DSL defined at the M2 layer is used to describe the core of a domain and can be used by a domain designer and language user to build domain models at the M1 layer. Because the metamodel allows for creating domain types (using the M2 concept *metatype*) and domain instances (using the M2 concept *metainstance*), domain designers and language users are able to model two ontological layers O2 and O1. Layer O2 consists of domain types and layer O1 consists of domain instances. Both ontological layers are connected by the explicit ontological *hasType*-relation between domain types and instances.

The OWL2 part of *OntoM2L* can be used to define axioms and restrictions in the metamodel and domain model. To reason on the additional semantics, especially the one of the explicit *hasType*- and *instanceOf*-relations, the domain model at the M1 layer with its types and instances is transformed to a description logics TBox and ABox, represented by the DE Ontology. Its TBox describes the terminology of the domain and represents the domain types together with its constraints, while the ABox contains concrete assertions about domain instances. In the case of language engineering the metamodel together with its metatypes and metainstances is transformed into the TBox contained by the LE Ontology. Each linguistic instances of the metamodel are transformed into the ABox. The two knowledge bases, which are implemented by an OWL2 ontology, are used by an inference engine, which provides additional services. These services for validating and explaining the metamodel can be used by the different users of the environment.

### 4.2 Implementation

In the following, we present some technical details of the environment and give an idea, how it is implemented.

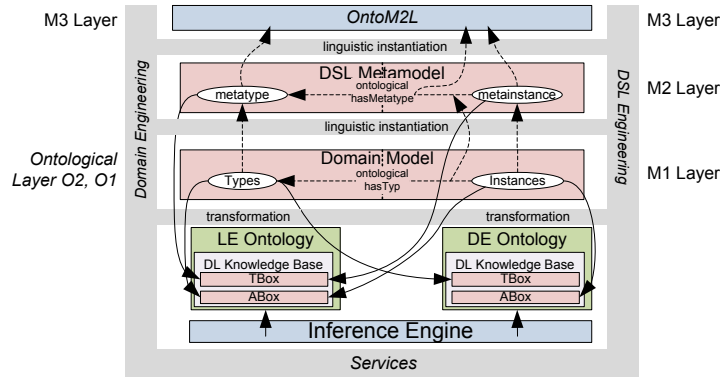


Fig. 5. Architectural overview of the environment

**Abstract syntax.** Figure 6 depicts an excerpt of the integrated metametamodel, which is part of OntoM2L and consists of two main parts: the *KM3+instance metametamodel* and the *OWL2 metamodel*.

The *KM3+instance metametamodel* provides all the concepts for modeling (meta-)types and (meta-)instances and adopts the *OWL2 metamodel*. For example, the class **KM3Class** is a specialization of **OWLClass**, thus it is possible to restrict classes by several class axioms provided by the *OWL2 metamodel*. **KM3Instance** is a specialization of **OWLIndividual** and **Instantiation** is a specialization of **ClassAssertion**. In *OWL* class assertions are used to define the class expressions as type of an individual. The *KM3+instance metametamodel* differentiates between elements of M2 layer and M1 layer. M2 elements, for example, are **Metatype** and **Metainstance**. Both can be connected by a **MetaHasType** relation. M1 elements, for example, are **Type** and **Instance** which optionally can be connected by a **HasType** relation.

The metametamodel allows for defining the linguistic instanceOf-relationship between M2 and M1 elements using the **InstanceOf**-class. Each linguistic instance must have exactly one metatype or metainstance. We must mention that several constraints for a restricted use of the metametamodel are not depicted in figure 6. They allow for defining, that **Type** only can be linguistic instance of **Metatype**, **Instance** only can be linguistic instance of **Metainstance** and **HasType** only can be linguistic instance of **MetaHasType**.

All classes in the M3 metametamodel which are specialization of **M2Element** are also specialization of **KM3Class** which is specialization of **OWLClass**. Hence their instances, which are represented at the M2 layer, are transformed to a **TBox** in the description logics knowledgebase (cf. figure 5). In the case of reasoning services for language engineering all instances of **M1Element**, which lie at the M1 layer, are transformed into a description logics **ABox**. Hence **M1Element** is specialization of **KM3Instance** in the metametamodel, because the **ABox** consists of instance definitions.

In the case of services for domain engineering we differ between elements for types and instances at the M1 layer. All instances of **Type** are transformed into a description logics **TBox**, hence **Type** is a specialization of **KM3Class** in the metametamodel. All



### 4.3 Services

In this section, we want to expose the services of the MDD environment for domain and language engineering. All services base on standard reasoning services and are provided to designers and users without any effort. This means that users and designers do not have to be familiar with using and reasoning of description logics knowledge bases.

**Services for language engineering.** Based on the knowledge base LE Ontology representing the language metamodel and the domain model, the environment provides several services to both language user and language designer. Language users mainly rely on services for validating their domain models and suggesting model elements to be used. Suggestion services can be realized by dynamic classification. It allows for determining the classes which one instance belongs to, based on all descriptions in the domain model and metamodel. The correctness of the domain-specific language under development is important for languages designers. Thus, they want 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. If language users want to verify whether all restrictions and constraints imposed by the metamodel hold, they can use a reasoning service to check the consistency of the domain model. An important feature of the environment is, if the model or metamodel are inconsistent or contain unsatisfiable concepts, the users and designers get additional explanations which lead to debug relevant facts and help in correcting the models [19].

**Services for domain engineering.** The services for domain designers rely on the extracted description logics knowledge base DE Ontology. With regard to the example in figure 3 they want to check, if all instances are consistent with regard to the domain types. Furthermore, they want to check if it is possible to create instances of a given type, in other words if types are satisfiable. Since at the beginning of describing the domain often instances exist in the model without any domain type, domain designers automatically want to classify them to get its possible types.

## 5 Discussion of the Approach

In this section, we establish the viability of our approach by a proof of concept discussion. We analyze the approach with respect to the challenges of section 1.1.

To address the modeling of ontological and linguistic instantiation relationship (challenge 1) we built a metametamodel, which allows for defining metatypes and metainstances within a language metamodel at the M2 layer. This metamodel allows for creating types and instances in one domain model. Furthermore, the metametamodel allows for explicitly designing a linguistic-instanceOf relationship, which relates elements of two different modeling layers, and an ontological hasType-relationship which allows for relating domain types with corresponding domain instances at the M1 layer.

To consider the combination of language engineering and domain engineering, we created a metametamodel that joins both concerns (challenge 2). Language designers using the metametamodel can design DSL metamodels at the M2 layer which is related

to language engineering. Domain designers and language users are able to create domain models containing both, domain types and instances. Domain models lie at the M1 layer and must conform to DSL metamodels via the linguistic *instanceOf*-relationship.

To have a language that allows for defining constraints (challenge 3) we considered the extended KM3 metametamodel and integrated it with the existing OWL2 meta-model at the M3 layer. Designers are able to define several constraints for types and instances and in addition constraints and axiom that cross type and instance level.

The defined metamodels are transformed into a pure description logics knowledge base. Thus we use model-theoretic semantics, which is taken into account by the environment for providing different services. These services are used by designers and users to validate models. If the model is not valid, they get several explanations and debugging relevant facts (challenge 3).

## 6 Related Work

In the following, we want to compare our approach with related work. In the first part of this section, we will depict related work on foundations of model-driven development environments. In the second part, we give some related work which is dealing with ontological metamodeling. The third part of this section discusses related approaches enriching the expressiveness of modeling languages.

Already in 2003, Atkinson et. al defined requirements of model-driven development infrastructures. Besides requirements for defining abstract syntax, concrete syntax and semantics within the infrastructure, they suggest to consider the dimensions of language engineering and domain engineering [3]. As proposed in [3] we provide the facility to built types and instances at the same model layer and thus allow for dynamically extending the set of domain types available for modeling.

In [20] a metamodeling language is presented which allows for building ontological theories as a base for modeling languages from the philosophical point of view. The M3 metamodel consists of elements for individuals and universals (types) and in addition provides a textual concrete syntax. In addition to this approach, we already provide formal semantics in particular for the *hasType*-relation, at least if the developed models are transformed into a description logics TBox and ABox. In [21] an ontological meta-model extension for generative architectures (OMEGA) is described as an extension to the MOF 1.4 metamodel that allows for ontological metamodeling. The core addition to the original MOF model is the introduction of concepts for *MetaElement* and *Instance*, which form the basis for all instantiations. In fact, the *hasType*-relation between *Instance* and *MetaElement* is implemented by a simple UML association which does not provide any semantics to further tools.

There are many model-based development environments for DSLs available in the market like, for example, *MetaEdit+* [22] or *ATLAS Model Management Architecture (AMMA)* [23]. These environments are aligned with the OMG four-layer metamodel architecture. Some of them provide support for specifying queries and constraints, e.g. with OCL-like languages. Here checking constraints and executing queries takes place on one single layer. Instead, our description logics-based approach allows for defining constraints that cover model and instance layer and provides querying and reasoning

simultaneously on both of them. Several approaches describe transformations of MOF-based models to knowledge representation languages where reasoning and querying are adopted. For example, [24] presents transformations from MOF-based models to Alloy, [25] presents an approach to describe the semantics of MOF-based models with F-Logic. Instead of these approaches, where the expressiveness available for designers is limited to MOF (plus OCL), we provide integrated modeling. Thus the designer benefits from the expressiveness of OWL additionally to the one of MOF.

## 7 Conclusion

We have shown how a combination of an extended KM3 metamodel and the OWL2 metamodel supports language and domain engineering. Description logics can support modeling and give constraints and semantics covering both, the instances and types defined in a model. We have presented an integrated approach where the modelers are able to use a simple, Java-like syntax but in addition can benefit from a language which provides much expressiveness and services for productive modeling. Furthermore, we presented an approach of joint domain- and language engineering. The result of language engineering is a new DSL, which defines the core of a domain and prescribes the design of domain models. Domain engineering, which results in a domain model, provides the facility to define new domain types during modeling, which is in general not possible using pure language engineering approaches. Currently rely on two ontological layers, since they can be covered by one OWL ontology, the work in the future may consist of generalizing the approach to allow modeling an arbitrary number of ontological layers.

*Acknowledgement.* We like to thank Krzysztof Miksa from Comarch for providing the use cases. This work is supported by EU STRP-216691 MOST.

## References

1. Kelly, S., Tolvanen, J.: Domain-specific modeling: enabling full code generation. Wiley-IEEE Computer Society Pr (2008)
2. Walter, T., Silva Parreiras, F., Staab, S.: OntoDSL: An Ontology-Based Framework for Domain-Specific Languages. In: Model Driven Engineering Languages and Systems, MoDELS 2009. Volume 5795 of LNCS., Springer (2009) 408–422
3. Atkinson, C., Kühne, T.: Model-driven development: A Metamodeling Foundation. IEEE Software **20**(5) (2003) 36–41
4. Mernik, M., Sloane, A.: When and how to develop domain-specific languages. Volume 37., ACM New York (2005) 316–344
5. OMG: Meta Object Facility (MOF) Core Specification. <http://www.omg.org/spec/MOF/2.0/> (January 2006)
6. Weiss, D., Lai, C.: Software product-line engineering. Addison-Wesley Reading, MA (1999)
7. Tairas, R., Mernik, M., Gray, J.: Using ontologies in the domain analysis of domain-specific languages. In: Models in Software Engineering. Volume 5421 of LNCS., Springer (2009) 332–342

8. Berardi, D., Calvanese, D., De Giacomo, G.: Reasoning on UML Class Diagrams. *Artificial Intelligence* **168**(1-2) (2005) 70–118
9. Baader, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.: *The description logic handbook*. Cambridge University Press New York, NY, USA (2007)
10. Guizzardi, G., Pires, L., Van Sinderen, M.: On the role of domain ontologies in the design of domain-specific visual modeling languages. In: *Proceedings of the 2nd Workshop on Domain-Specific Visual Languages, 17th ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2002)*. (2002)
11. Jouault, F., Bezivin, J.: KM3: a DSL for Metamodel Specification. In: *8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*. Volume 4037 of LNCS., Springer (2006) 171–185
12. 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)
13. OMG: *Object Constraint Language Specification, version 2.0*. Object Management Group. (June 2005)
14. Mernik, M., Sloane, A.: When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)* **37**(4) (2005) 316–344
15. Happel, H.J., Seedorf, S.: Applications of ontologies in software engineering. In: *Workshop on Sematic Web Enabled Software Engineering, SWESE 2006*. (2006) 5–9
16. de Almeida Falbo, R., Guizzardi, G., Duarte, K.: An ontological approach to domain engineering. In: *International Conference on Software Engineering and Knowledge Engineering, SEKE 2002*. Volume 27 of *International Conference Proceedings.*, ACM (2002) 351–358
17. Motik, B., Horrocks, I., Rosati, R., Sattler, U.: Can OWL and logic programming live together happily ever after? In: *International Semantic Web Conference, ISWC 2006*. Volume 4273 of LNCS., Springer (2006) 501–514
18. Parsia, B., Sirin, E.: Pellet: An OWL DL Reasoner. In: *International Workshop on Description Logics, DL 2004*. Volume 104 of *CEUR Workshop Proceedings*. (2004)
19. Parsia, B., Sirin, E., Kalyanpur, A.: Debugging OWL ontologies. In: *International Conference on World Wide Web, WWW 2005*, ACM (2005) 633–640
20. Laarman, A., Kurtev, I.: Ontological Metamodeling with Explicit Instantiation. In: *Conference on Software Languages Engineering, SLE 2009*. LNCS, Springer (2009)
21. Gitzel, R., Ott, I., Schader, M.: Ontological Extension to the MOF Metamodel as a Basis for Code Generation. Volume 50., Oxford University Press (2007) 93–115
22. Kelly, S., Lyytinen, K., Rossi, M.: *MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment*, Springer (1996) 1–21
23. Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-Based DSL Frameworks. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, ACM (2006) 22–26
24. Anastakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. In: *Model Driven Engineering Languages and Systems MoDELS 2007*. Volume 4735 of LNCS., Springer (2007) 436
25. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The missing link of MDA. In: *International Conference on Graph Transformation*. Volume 2505 of LNCS., Springer (2002) 90–105