

Model Driven Engineering with Ontology Technologies

Steffen Staab, Tobias Walter, Gerd Gröner, and Fernando Silva Parreiras

Institute for Web Science and Technology, University of Koblenz-Landau
Universitätsstrasse 1, Koblenz 56070, Germany
{staab, walter, groener, parreiras}@uni-koblenz.de

Abstract. Ontologies constitute formal models of some aspect of the world that may be used for drawing interesting logical conclusions even for large models. Software models capture relevant characteristics of a software artifact to be developed, yet, most often these software models have no formal semantics, or the underlying (often graphical) software language varies from case to case in a way that makes it hard if not impossible to fix its semantics. In this contribution, we survey the use of ontology technologies for software modeling in order to carry over advantages from ontology technologies to the software modeling domain. It will turn out that ontology-based metamodels constitute a core means for exploiting expressive ontology reasoning in the software modeling domain while remaining flexible enough to accommodate varying needs of software modelers.

1 Introduction

Today *Model Driven Development* (MDD) plays a key role in describing and building software systems. A variety of software modeling languages may be used to develop one large software system. Each language focuses on different views and problems of the system [1]. *Model Driven Engineering* (MDE) is related to the design and specification of modelling languages, and it is based on the four-layer modelling architecture [2]. In such a modelling architecture, the M0-layer represents the real world objects. Models are defined at the M1-layer, a simplification and abstraction of the M0-layer. Models at the M1-layer are defined using concepts which are described by metamodels at the M2-layer. Each metamodel at the M2-layer determines how expressive its models can be. Analogously, metamodels are defined by using concepts described as metametamodels at the M3-layer.

Although the four-layer modelling architecture provides the basis for formally defining software modelling languages, we have identified some open challenges. Semantics of modelling languages often is not defined explicitly but hidden in modelling tools. To fix a specific formal semantics for metamodels, it should be defined precisely in the metamodel specification. The syntactic correctness of models is often analyzed implicitly using procedural checks of the modelling

tools. To make well-formedness constraints more explicit, they should be defined precisely in the metamodel specification.

OWL 2, the web ontology language, is a W3C recommendation with a very comprehensive set of constructs for concept definitions [3] and allow for specifying formal models of domains. Ontologies are conceptual models, that can be described by OWL. Based on its underlying formal semantics different services are provided, which vary between satisfiability checking at the model layer, checking the consistency of instances with regard to the model, or classifying instances (finding their possible types) with regard to instance and type descriptions. Since ontology languages are described by metamodels and allow for describing structural and behavioural models, they provide the capability to combine them with software modelling languages.

In this chapter, we tackle challenges of defining both semantics and syntactic constraints, restricting the use of the abstract syntax of a modelling language, for software languages. We show how ontologies can support the definition of software modelling language semantics and provide the definition of syntactic constraints. Since OWL 2 has not been designed to act as a metamodel for defining modelling languages, we propose to build such languages in an integrated manner by bridging pure language metamodels and an OWL metamodel in order to benefit from both approaches.

This chapter is structured as follows: In Section 2, we give a short introduction of Model Driven Engineering (MDE). In Section 3, we consider ontology languages and technologies. Here we present the ontology language OWL 2. Furthermore, we consider standard ontology reasoning services and services for explanation and model repair. In Section 4 we present architectures for bridging software modelling languages and ontology technologies. Section 5 and 6 are dealing with ontology reasoning for modelling languages, where Section 5 considers structural modelling languages and Section 6 considers behavioral modelling languages. Section 7 presents the TwoUse Toolkit which implements most of the approaches presented in this paper. Section 8 gives some related work and Section 9 concludes the paper.

2 Model-Driven Engineering

The approach of model-driven software engineering (MDE) [1] suggests first to develop models describing a system in an abstract way, which is transformed in several steps into real, executable systems (e.g. source code).

Figure 1 illustrates a generic process of model driven engineering. Here a software designer starts with creating a model n , which conforms to a modelling language n . Model n describes the system by a very abstract representation.

By transforming the model n to model $n - 1$, which conforms to another modelling language $n - 1$, the software designer increases the platform specificity and simultaneously lowers the level of abstraction. At the end of the MDE process, source code may be generated from model 1, where the source code (e.g. Java code) conforms to some EBNF grammar.

In general MDE consists of the following two main artifacts: *Modelling languages*, which are used to describe a set of models and *model transformations*, which are used to translate models represented in one language into models represented in another language. Both, modelling languages and model transformations are introduced in the following two sections.

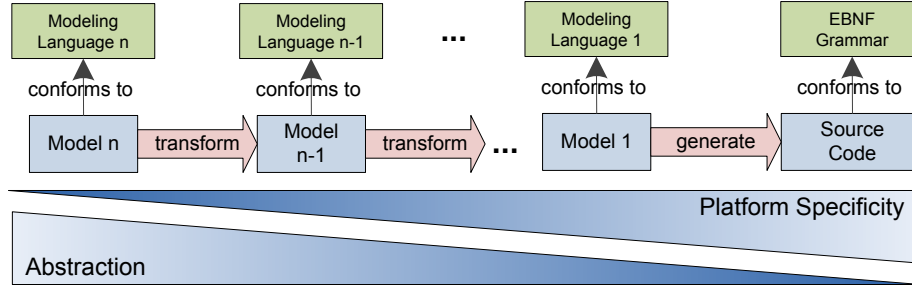


Fig. 1. Overview of MDE

2.1 Modelling Languages

The meaning of a model must be well-defined such that multiple developers can understand and work with it. If the meaning is not clear and unique there would be no possibility to define automated transformations from one abstract model to a more specific model. Furthermore, an agreed meaning of a model can be based on a common language defining precise syntax and semantics used for describing a model. In MDE models are described by modelling languages, where modelling languages themselves are described by so called metamodeling languages. A modelling language consists of an *abstract syntax*, at least one *concrete syntax* and *semantics*.

The abstract syntax of a modelling language is described by a metamodel and is designed by a *language designer*. A metamodel is a model that defines the concepts and reference for expressing a model. Semantics of the language may be defined by a natural language specification or may be captured (partially) by logics. A concrete syntax which could be of a textual or visual kind is used by a *language user* to create software models. Since metamodels are also models metamodeling languages are needed, to describe modelling languages. Here the abstract syntax is described by a metamodel.

In the scope of graph-based modelling to create software models [4] a meta-modelling language (e.g. grUML [5]) must allow for defining graph schemas, which provide types for vertices and edges, and structures them in hierarchies. In this case, each graph is an instance of its corresponding *graph schema*.

The Meta-Object Facility (MOF) is OMG's standard for defining metamodels. It provides a language for defining the abstract syntax of modelling lan-

guages. MOF is in general a minimal set of concepts which can be used for defining other modelling languages. The version 2.0 of MOF provides two metameta-models, namely *Essential MOF* (EMOF) and *Complete MOF* (CMOF). EMOF prefers simplicity of implementation before expressiveness. CMOF instead is more expressive, but more complicated to implement [6]. EMOF mainly consists of the Basic package of the Unified Modelling Language (UML) which is part of the UML infrastructure [7]. It allows for defining classes together with properties, which are used to describe data attributes of classes and which allow for referring to other classes.

Figure 2 depicts a simplified version of the ECore metametamodel. A meta-model described by Ecore consists of *Packages* which can be nested and contain a set of *TypedElements*. *Packages*, *TypedElements* and *Types* are *NamedElements*, thus they have a name. *Type* has the two subclasses *Class* and *Datatype*. A *Class* can optionally be abstract and can be specialized. *Classes* contain properties which are represented by the class *Property*. A *Property* is a *MultiplicityElement* and *TypedElement*. This means, for each *Property* a lower and upper cardinality is defined and each *Property* has at least one *Type*. *DataTypes* can be either *PrimitiveTypes* or *Enumerations*, where *Enumerations* contain a set of literals (not depicted in Figure 2).

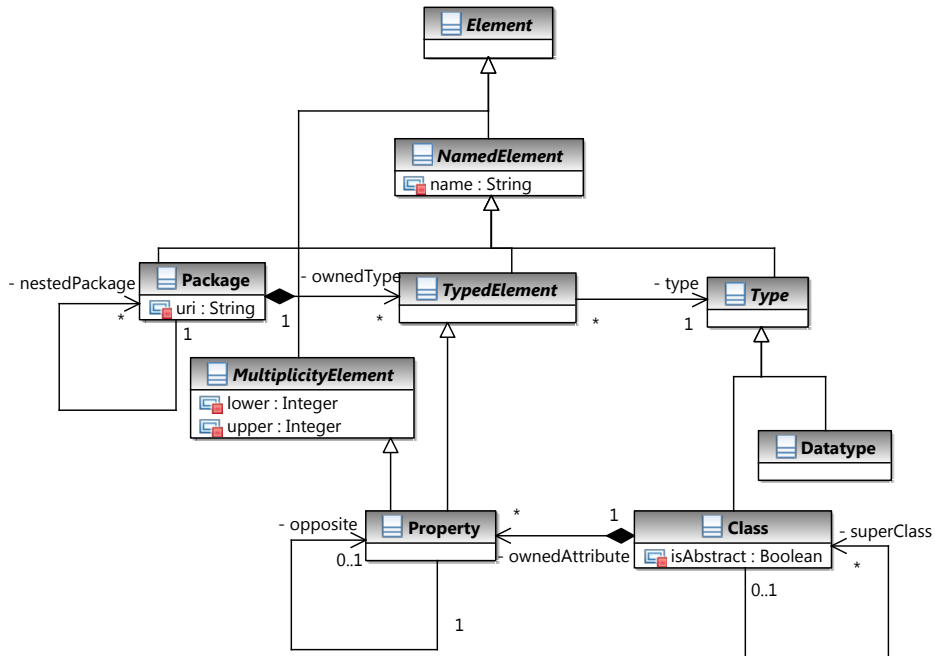


Fig. 2. Essential Meta Object Facility (EMOF)

Another metametamodel is provided by the Ecore metamodeling language, which is used in the Eclipse Modelling Framework [8]. It is an implementation of EMOF and will be considered in the rest of this paper. Ecore provides four basic constructs: (1) `EClass` - used for representing a modeled class. It has a name, zero or more attributes, and zero or more references. (2) `EAttribute` - used for representing a modeled attribute. Attributes have a name and a type. (3) `EReference` - used for representing an association between classes. (4) `EDataType` - used for representing attribute types.

As already mentioned in the introduction, models, metamodels and metametamodels are arranged in a hierarchy of 4 layers. Figure 3 depicts such a hierarchy. Here the Ecore metametamodel is chosen to define a metamodel for a process language, which is built by the language designer. He uses the metametamodel by creating instances of the concepts it provides. The language user takes into account the metamodel and creates instances which build a concrete process model. A process model itself, for example, can describe the behavior of a system running in the real world.

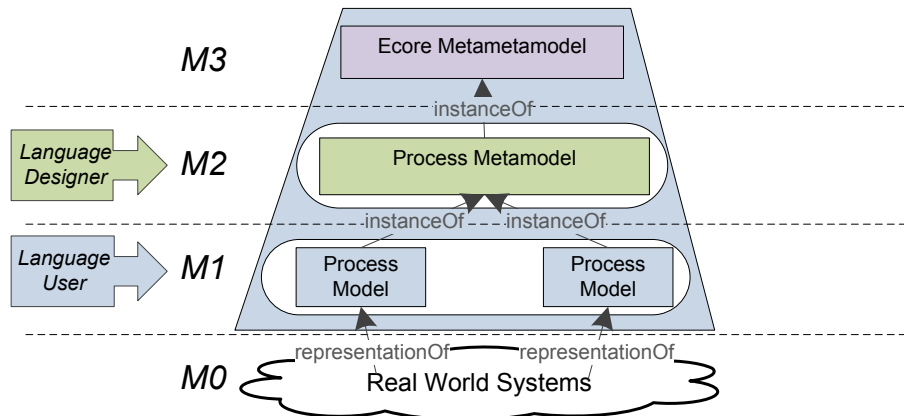


Fig. 3. A metamodel hierarchy

Figure 5 depicts a sample process model in concrete syntax which is instance of the metamodel depicted in Figure 4. Here we used a textual syntax to define classes like `ActivityNode` or `ActivityEdge` and references like `incoming` or `outgoing` to define links between instances of corresponding classes in the metamodel. In particular we considered the KM3 (Kernel MetaMetaModel) metamodeling language [37] to design the metamodel in figure 4. KM3 is an implementation of EMOF and provides a textual concrete syntax for coding M2 metamodels.

Action nodes (e.g., `Receive Order`, `Fill Order`) are used to model concrete actions within an activity. Object nodes (e.g. `Invoice`) can be used in a variety of ways, depending on where values or objects are flowing from and to.

Control nodes (e.g. the initial node before **Receive Order**, the decision node after **Receive Order**, and the fork node and join node around **Ship Order**, merge node before **Close Order**, and activity final after **Close Order**) are used to coordinate the flows between other nodes.

Process models in our example can contain two types of edges, where edges have exactly one source and one target node. One edge is used for object flows and another edge for control flows. An object flow edge models the flow of values to or from object nodes. A control flow is an edge that starts an action or control node after the previous one is finished.

```

1  abstract class ActivityNode {
    reference incoming [0-*] : ActivityEdge oppositeOf target;
    reference outgoing [0-*] : ActivityEdge oppositeOf source;
  }
  class ObjectNode extends ActivityNode { }
6  class Action extends ActivityNode {
    attribute name : String;
  }

11 abstract class ControlNode extends ActivityNode { }
  class Initial extends ControlNode { }
  class Final extends ControlNode { }
  class Fork extends ControlNode { }
  class Join extends ControlNode { }
  class Merge extends ControlNode { }
16  class Decision extends ControlNode { }

  abstract class ActivityEdge {
    reference source [1-1] : ActivityNode;
    reference target [1-1] : ActivityNode;
21 }
  class ObjectFlow extends ActivityEdge { }
  class ControlFlow extends ActivityEdge { }

```

Fig. 4. Process metamodel at M2 layer

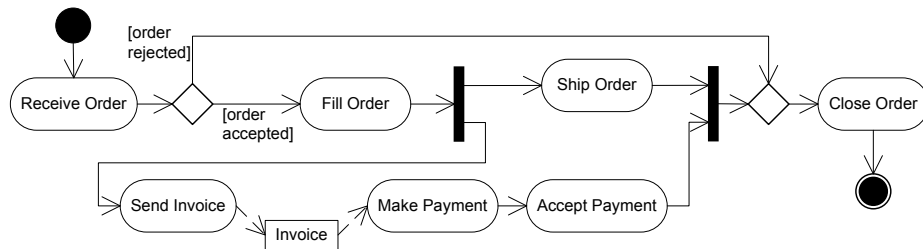


Fig. 5. Process model at M1 layer

2.2 Model Transformations

In the following, we present the idea of model transformations. Figure 6 gives an overview of all relevant artifacts that are involved in defining and executing a model transformation.

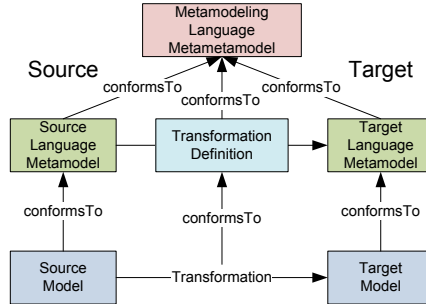


Fig. 6. Model transformations

A model transformation automatically generates a target model from a source model, according to a transformation definition. Here the source model and the target model conform to the metamodel of the corresponding source and target language. Further both metamodels conform to a metamodel of the metamodeling language.

A transformation definition is a set of transformation rules that together describe how a model conforming to the source metamodel can be transformed into a model conforming to the target metamodel. A transformation rule is a description of how one or more constructs in the source metamodel can be transformed into one or more constructs in the target metamodel.

Some prominent model transformation languages are ATL [9], QVT [10], MOLA [11] and FUJABA [12].

2.3 Challenges

In the following, we want to list two MDE challenges which partially come from [13] and are exemplified in the following. Currently, a language designer takes into account a usual metamodeling language which only allows for describing structural aspects of models by concepts like classes or references. Language users consider these languages to create models and require services (e.g. guidance, debugging) for more productive modelling. To provide services to language users, a language designer must specify modelling languages in a more formal way by describing well-formedness constraints in metamodels and (partially) formal semantics of modelling languages.

Metamodel well-formedness constraints and checks Model correctness is often analyzed implicitly in procedural checks of the modelling tools. To make well-formedness constraints more explicit, they should be defined precisely in the metamodel specification.

Although the Ecore language allows for defining cardinality restrictions for references it is impossible to define constraints covering all flows of a given process model. For example, a language designer wants to ensure, that every flow in an M1 process model goes from the initial node to some final node. To achieve such constraints, language designers have to define formal syntactic constraints of the modelling language which models have to fulfill. Such constraints should be declared directly within the metamodel of the language. Based on metamodel constraints, language designers may want to provide services to the language user.

Language semantics The semantics of modelling languages is often not defined explicitly but hidden in modelling tools. To fix a specific formal semantics for languages, it should be defined precisely either in the metamodel specification or by transformations which transform software models into logic representations.

In the next section we present a couple of reasoning services. Later from Section 4 to Section 6 we will show how to use the reasoning services to tackle the challenges mentioned above.

3 Ontology Languages and Technologies

In this section, we start with the metamodel of the ontology language OWL 2. Furthermore, we are dealing with ontology-based reasoning services which are later used in designing and using modelling languages. We start with a list of standard reasoning services (Section 3.2). Additionally, we consider in this section querying services which are based on SPARQL (Section 3.3), explanation services to explain inferences in ontologies (Section 3.4), repairing services which give advices of how to correct ontologies (Section 3.5), and linking services which show how to combine different ontologies (Section 3.6).

3.1 Ontology Language OWL

Improvements on the OWL language led the W3C OWL Working Group to publish working drafts of a new version of OWL: OWL 2.

OWL 2 is fully compatible with OWL-DL and extends the latter with limited complex role inclusion axioms, reflexivity and irreflexivity, role disjointness and qualified cardinality restrictions. Moreover, OWL 2 is axiom-based. The three important kinds of axioms are class axioms, object property axioms and data property axioms. Class axioms like `SubClassOf`- or `EquivalentClasses`-axioms are combined with class expressions and describe a subclass relation between two classes and a set of classes with equivalent concepts, respectively. Two of the

object property axioms are the `ObjectPropertyDomain`- and `ObjectPropertyRange`-axioms which restrict the object property only to be connected with given class expressions defined in the domain and range. The data property axioms for the data property domain are analogous to the one of the object property. The axioms for data property range define the data range a given data property contains. Classes and properties themselves are entities and are declared by a corresponding `Declaration`-axiom.

The complete abstract syntax is presented in [3]. Its semantics is presented in [14]

3.2 Standard Ontology Reasoning Services

In the following paragraphs, we consider standard reasoning services that are provided by reasoners (e.g., Pellet [15]).

Consistency Checking The reasoning service consistency checking checks if a given ontology \mathcal{O} is consistent, i.e. if there exists a model (a model-theoretic instance) for \mathcal{O} . If ontology \mathcal{O} is consistent, then return *true*, otherwise *false*.

Satisfiability Checking The satisfiability checking service finds all unsatisfiable concepts in a given ontology \mathcal{O} . A concept in an ontology \mathcal{O} is unsatisfiable if it represents an empty set of all models in the ontology, i.e. it cannot be instantiated. If the ontology \mathcal{O} has no unsatisfiable concept the service returns the empty set.

Classification The classification service returns for a given ontology \mathcal{O} and an individual i a set of concepts which contain/describe the individual. The individual conforms to all concepts in the result of the classification service.

Subsumption The subsumption checking service checks whether the interpretation of A , the set of individuals described by A , is a subset of the interpretation of B for a given ontology \mathcal{O} . If the interpretation of A is a subset of the interpretation of B , then it returns *true*. Otherwise it returns *false*.

3.3 Ontology Querying Service

SPARQL is the W3C standard query language for RDF graphs, which is a triple-based language [16]. Writing SPARQL queries for OWL can be time-consuming for those who work with OWL ontologies, since OWL is not triple-based and requires reification of axioms when using a triple-based language. Therefore, we proposed SPARQLAS, a language that allows for specifying expressions that rely on inferences over OWL class descriptions [17]. It is a seamless modification of the SPARQL syntax for querying OWL ontologies. SPARQLAS enables using

variables (prefixed by ?) wherever an entity (Class, Datatype, ObjectProperty, DataProperty, NamedIndividual) or a literal is allowed.

SPARQLAS queries are translated into SPARQL queries and can be executed by any SPARQL engine that supports graph pattern matching for the OWL 2 entailment regime [18]. SPARQLAS queries operate on both the schema level (M2 metamodel layer) and on the instance level (M1 model layer). For example, Listing 1.1 shows a SPARQLAS query about activities that are followed by a decision node. In this example, we ask about the individuals ?x whose type is an anonymous class where the property target has as value some control flow with some property target having some Decision.

Listing 1.1. Use Cases that includes some other use case

```
1 Namespace: uml = <http://www.example.org/BPM#>
   Select ?x
   Where:
     ?x type (ActivityNode and outgoing some (ControlFlow and
       target some Decision))
```

3.4 Explanation Service

In traditional ontology development environments (e.g., Protégé [19]), users are typically able to model ontologies and use reasoners (e.g., Pellet [15]) to compute unsatisfiable classes, subsumption hierarchies and types for individuals.

In the following, we introduce the terminology for justifications and present some ideas on how to compute justifications.

Computing Justifications Explanation services for MDE that are based on ontology technologies are mainly based on computing so-called justifications.

However, since in the context of MDE ontology technologies are used in software modelling it has become evident that there is a significant demand for software modelling environments which provide more sophisticated explanation services. In particular, the generation of explanations, or justifications, for inferences computed by a reasoner is now recognized as highly desirable functionality for both ontology development and software modelling. A language user or designer developing (meta-) model recognizes an entailment and wants to get an explanation for the entailment in order to get the reason why the entailment holds (understanding entailments). If the entailment leads to some inconsistency or unsatisfiable classes, the user wants to get some debugging relevant facts and the information how to repair the ontology.

In the following, we present some ideas on how to compute justifications.

Terminology In the following, the terminology that is related to the field of justification and debugging is depicted.

A class is unsatisfiable (with regard to one ontology) if it cannot possibly have any instances in any model of the ontology. In description logics notation,

$C \sqsubseteq \perp$ means that C is not satisfiable. If an ontology \mathcal{O} contains at least one unsatisfiable class it is called *incoherent*. An ontology \mathcal{O} is inconsistent if and only if it does not have any model. In description logics, this is the case if an ontology \mathcal{O} entails $\top \sqsubseteq \perp$. $\mathcal{O} \models \eta$ holds if all models of \mathcal{O} also satisfy η . Justifications are explanations of entailments in ontologies. Let \mathcal{O} be an ontology with entailment $\mathcal{O} \models \eta$. Then \mathcal{J} is a justification for η if $\mathcal{J} \subseteq \mathcal{O}$ with $\mathcal{J} \models \eta$ and for any $\mathcal{J}' \subset \mathcal{J}$ the following holds: $\mathcal{J}' \not\models \eta$.

In the following, we present the ideas of a simple black box method for computing a single justification and two further methods for computing more than one possible justification. For details about the algorithms we refer to literature, e.g., [20, 21].

Simple Black Box Method The approach of a simple black-box technique to compute justifications was presented in [21]. Given a concept C which is unsatisfiable with regard to an ontology \mathcal{O} . In a first step of the computation, axioms of \mathcal{O} are added to a newly created ontology \mathcal{O}' until C gets unsatisfiable with regard to \mathcal{O}' . In a second step extraneous axioms in \mathcal{O}' will be deleted to get a single minimal justification. The deletion of axioms stops when concept C gets satisfiable.

Glass Box Method In [20] a glass box technique for computing one single justification is presented. The technique is used for presenting the root cause of a contradiction and to determine the minimal set of axioms in the ontology which lead to a semantic clash. In glass box techniques, the internals of a description logics tableaux reasoner are modified to extract and reveal the cause for inconsistency of a concept definition. An advantage of such approaches is that by tightly integrating the debugging with the reasoning procedure, precise results can be obtained. On the other hand, the reasoner needs to maintain extra data structures to track the source and its dependencies and this introduces additional memory and computation consumption. We refer to [20] where the complete algorithms and methodologies are explained in detail.

Computing all justifications If an initial justification is given (for example, computed by some black box technique), other techniques are used to compute the remaining ones. In [21] a variation of the classical Hitting Set Tree (HST) algorithm [22] is presented. This technique is also reasoner independent (black-box). The idea is that given an algorithm to find a single justification for concept unsatisfiability (like the above black-box method), to find in a first step one justification and set it as the root node of the so called Hitting Set Tree (HST). In the next steps, each of the axioms in the justification is removed individually, thereby creating new branches of the HST, and find new justifications along these branches on the fly in the modified ontology. This process needs to be exhaustively done in order to compute all justifications. The algorithm repeats this process until the concept turns satisfiable.

Example Listing 1.2 depicts an ontology represented in the textual functional-style syntax. The ontology consists of concepts of the domain of physical devices. The general physical structure of a `Device` consists of a `Configuration` which has a number of `Slots` into which `Cards` can be plugged in. (In Section 5 we will consider the domain of physical devices in the context of MDE where domain-specific languages are used to model devices and its possible configurations.)

The most important class `Device` has as superclass an anonymous OWL concept, which defines that every device is connected via the property `hasConfig` with `Configuration7603`. Furthermore, class `Device` is equivalent with an anonymous concept which requires, that each device is connected via property `hasConfig` with some `Configuration7604`.

The property `hasConfig` connects `Device` with the intersection of the classes `Configuration7603` and `Configuration7604`.

Further classes in the metamodel are `Cisco`, a subclass of `Device` and class `Configuration`, which has two subclasses, namely `Configuration7603` and `Configuration7604` which are pairwise disjoint.

Listing 1.2. Example of Metamodel enriched by ontology expressions

```

1 EquivalentClasses(Device ObjectSomeValuesFrom(hasConfig Configuration7603))
  SubClassOf(Device ObjectSomeValuesFrom(hasConfig Configuration7604))

  SubClassOf(Cisco Device)
5
  SubClassOf(Configuration7604 Configuration)
  DisjointClasses(Configuration7604 Configuration7603)

  SubClassOf(Configuration owl:Thing)
10
  SubClassOf(Configuration7603 Configuration)
  DisjointClasses(Configuration7603 Configuration7604)

  ObjectPropertyDomain(hasConfig Device)
15 ObjectPropertyRange(hasConfig ObjectIntersectionOf(Configuration7604 Configuration7603))

```

It is obvious that the ontology contains two unsatisfiable classes, namely `Device` and `Cisco`. The objectives for the explanation service are to provide a set of justifications, consisting of axioms. To get an explanation, why the class `Device` is not satisfiable, the metamodel is transformed into a description logics TBox (as explained in Section 5).

In Listing 1.3 we present two justifications for class `Device` and class `Cisco`, respectively, which are for simpler reading and understanding rendered using the Manchester Syntax style [23]. The output was generated by the TwoUse Toolkit(cf. Section 7).

Listing 1.3. Justifications for metamodel unsatisfiability

```

1 Unsatisfiability of Device:
  Explanations (2):
  1) hasConfig range Configuration7603
     and Configuration7604
  5 Configuration7603 disjointWith Configuration7604
     Device subClassOf hasConfig some Configuration7604

  2) hasConfig range Configuration7603
     and Configuration7604

```

10 Configuration7603 **disjointWith** Configuration7604
Device **equivalentTo** hasConfig **some** Configuration7603

Unsatisfiability of Cisco:

Explanations (2):

- 15 1) hasConfig **range** Configuration7603
 and Configuration7604
 Configuration7603 **disjointWith** Configuration7604
 Cisco **subClassOf** Device
 Device **subClassOf** hasConfig **some** Configuration7604
- 20 2) hasConfig **range** Configuration7603
 and Configuration7604
 Configuration7603 **disjointWith** Configuration7604
 Cisco **subClassOf** Device
- 25 Device **equivalentTo** hasConfig **some** Configuration7603
-

3.5 Ontology Repair Service

In the above section, we have highlighted some ontology explanation services that can be used to highlight the core erroneous axioms and concepts in a defect ontology. The ontology results from the language metamodel and model, which are transformed into the ontology. The next step is to resolve the errors by processing the justifications to give at least advices how to repair the ontology, respectively metamodel and model.

In the following, we discuss an idea for computing advices for repairing/changing models. In general, approaches for ontology repairing are presented by Kalyanpur et al., for example, in [24, 25].

Although there might be (semi-)automatic strategies for repairing ontologies and models, each step underlies the knowledge and proof of domain experts.

Based on a set of justifications different axiom rating strategies can be adopted. A simple one is to compute the frequency of an axiom. Here the number of times the axiom appears in each justification of the various unsatisfiable concepts in an ontology is counted. If an axiom appears in all justifications for n different unsatisfiable concepts removing the axiom from the ontology ensures that n concepts become satisfiable. Thus, the higher the frequency, the lower (better) the rank assigned to the axiom.

With regard to our example with the two unsatisfiable classes Device and Cisco, the axioms Configuration7603 **disjointWith** Configuration7604 and **hasConfig range** Configuration7603 and Configuration7604 have the highest frequencies. Thus, removing one of the axioms would solve the unsatisfiability problem of class Device and Cisco.

3.6 Ontology Linking Service

In a model-driven paradigm, resources that are expressed using different modelling languages must be reconciled before being used. To apply (semi-) automatically linking between models and metamodels, both are transformed into an ontology, where models are transformed into the description logics ABox and metamodels to the description logics TBox. In this section, we only illustrate

some of the multiple ontology matching techniques. For a deeper understanding on this topic, please refer to [26].

Ontology matching is the discipline responsible for studying techniques for reconciling multiple resources on the web. It consists of two steps: (1) match and determine alignments and (2) the generation of a processor for merging and transforming the ontologies. Matching identifies the correspondences. A correspondence for two ontologies A and B is a quintuple including an id, an entity of ontology A, an entity of ontology B, a relation (equivalence, more general, disjointness) and a confidence measure. A set of correspondences is an alignment. Correspondences can be done at the schema-level (metamodel) and at the instance-level (model).

Matchings can be based on different criteria: name of entities, structure (relations between entities, cardinality), background knowledge like existing ontologies or WordNet. Furthermore, matchings are established according to the different structures that are compared.

Internal structure comparison: this includes property, key, datatype, domain and multiplicities comparison.

Relational structure comparison: the taxonomic structure between the ontologies is compared.

Extensional techniques: extensional information is used in this method, e.g., formal concept analysis.

Automatic matching techniques can be seen as support but should be assisted by domain experts, because of false positive matches.

4 Bridging Software Languages and Ontology Technologies

In the following, we present two general approaches of bridging software models and ontologies. The two approaches mainly differ in the layer of the model hierarchy where they are defined and the layer where the bridge is used and applied on software models.

4.1 Language Bridge

Figure 7 depicts the general architecture of a language bridge, combining software languages and ontology technologies. The bridge itself is defined at the M3 layer, where a metamodel like Ecore is considered and bridged with the OWL metamodel. Here we differ between two kinds of bridges: *M3 Integration Bridge* and *M3 Transformation Bridge*.

M3 Integration Bridge The design of an M3 integration bridge consists mainly of identifying concepts in the Ecore metamodel and the OWL metamodel which are combined.

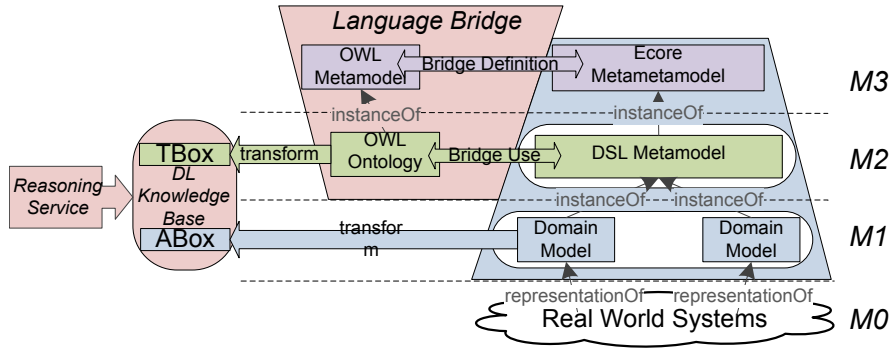


Fig. 7. Language bridge

Here existing metamodel integration approaches (e.g. presented in [27] and [28]) to combine the different metamodels are used. The result is a new meta-modelling language, which allows for designing language metamodels at the M2 layer with integrated constraints.

An integrated metamodeling language provides all classes of the Ecore metamodel and OWL metamodel. It merges, for example, OWL Class with Ecore EClass, OWL ObjectProperty with Ecore References or OWL DataProperty with Ecore Attribute. Thus, a strong connection between the two languages is built. Since a language designer creates a class, he is in the scope of both OWL class and Ecore class. Hence a language designer can use the designed class within OWL class axioms and simultaneously use features of the Ecore meta-modelling language, like the definition of simple references between two classes.

The integration bridge itself is used at the M2 layer by a language designer. He is now able to define language metamodels with integrated OWL annotations to restrict the use of concepts he modeled and to extend the expressiveness of the language.

To provide reasoning services to language users and language designers, the integrated metamodel is transformed into a Description Logics TBox. The models created by the language users are transformed into a corresponding Description Logics ABox. Based on the knowledge base consisting of a TBox and ABox we can provide standard reasoning services and provide specific modelling to both language user and designer.

M3 Transformation Bridge The M3 Transformation Bridge allows language designers and language users to achieve representations of software languages (Metamodel/Model) in OWL. It provides the transformation of software language constructs like classes and properties into corresponding OWL constructs.

As one might notice, Ecore and OWL have a lot of similar constructs like classes, attributes and references. To extend the expressiveness of Ecore with

OWL constructs, we need to establish mappings between the Ecore constructs onto OWL constructs. Table 4.1 presents a complete list of similar constructs.

Table 1. Ecore and OWL: comparable constructs

Ecore	OWL
package	ontology
class	class
instance and literals	individual and literals
reference, attribute	object property, data property
data types	data types
enumeration	enumeration
multiplicity	cardinality

Based on these mapping, we develop a generic transformation script to transform any Ecore Metamodel/Model into OWL TBox/ABox – *OWLizer*. Figure 8 depicts the conceptual schema of transforming Ecore into OWL.

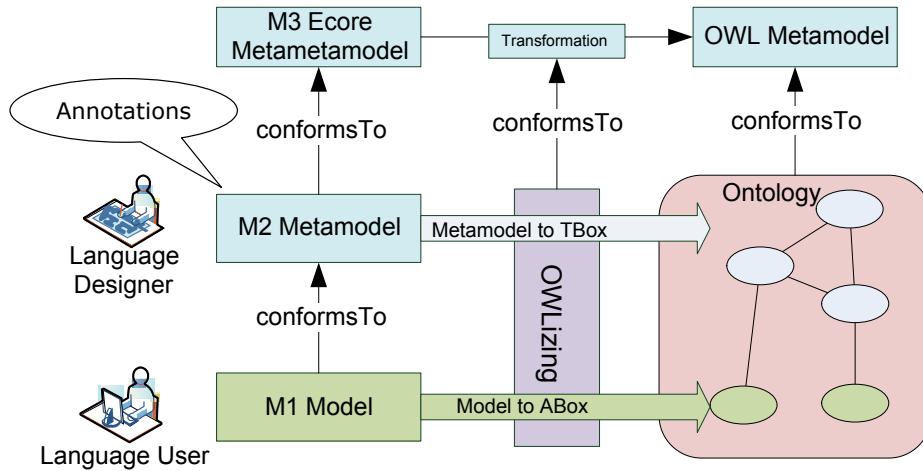


Fig. 8. OWLizer

Figure 8 shows three modelling levels according to the OMGs metamodel architecture: the metametamodel level (M3), the metamodel level (M2) and the model level (M1). Vertical arrows denote instantiation whereas the horizontal arrows are transformations, and boxes represent packages.

A model transformation takes the UML metamodel and the annotations as input and generates an OWL ontology where the concepts, enumerations, properties and data types (TBox) correspond to classes, enumerations, attributes/references and data types in the UML metamodel. Another transformation takes

the UML model created by the UML user and generates individuals in the same OWL ontology. The whole process is completely transparent for UML users.

As one may notice, this is a generic approach to be used with any Ecore-based language. For example, one might want to transform the UML Metamodel/Models as well as all the Java grammar/code into OWL (classes/individuals). This approach can be seen as a linked data driven software development environment [29].

4.2 Model Bridge

Model bridges connect software models and ontologies on the modelling layer M1. They are defined in the metamodelling layer M2 between different metamodels. Figure 9 visualises a model bridge. The bridge is defined between a process metamodel on the software modelling side and an OWL metamodel in the OWL modelling hierarchy. The process metamodel is an instance of an Ecore (EMOF) metametamodel.

A model bridge is defined as follows: (1) Constructs in the software modelling and in the ontology space are identified. These constructs, or language constructs, are used to define the corresponding models in the modelling layer M1. (2) Based on the identification of the constructs, the relationship between the constructs are analyzed and specified, i.e. the relationship of an Activity in a process metamodel like the BPMN metamodel to an OWL class. We distinguish between a *transformation* and *integration bridge*.

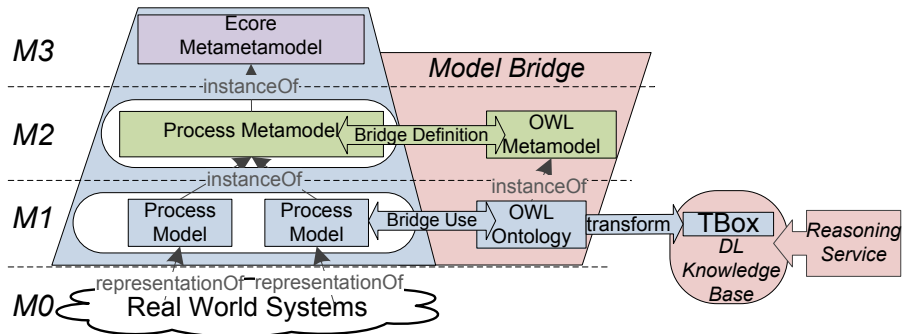


Fig. 9. Model bridge

M2 Integration Bridge *Integration bridges* merge information of the models from the software modelling and from the ontology space. This allows the building of integrated models (on modelling layer M1) using constructs of both modelling languages in a combined way, e.g. to integrate UML class diagrams and OWL.

As mentioned in Section 4.1, UML class-based modelling and OWL comprise some constituents that are similar in many respects like classes, associations, properties, packages, types, generalization and instances [30]. Since both approaches provide complementary benefits, contemporary software development should make use of both. The benefits of an integration are twofold. Firstly, it provides software developers with more modelling power. Secondly, it enables semantic software developers to use object-oriented concepts like inheritance, operation and polymorphism together with ontologies in a platform independent way.

Such an integration is not only intriguing because of the heterogeneity of the two modelling approaches, but it is now a strict requirement to allow for the development of software with many thousands of ontology classes and multiple dozens of complex software modules in the realms of medical informatics [31], multimedia [32] or engineering applications [33].

TwoUse (Transforming and Weaving Ontologies and UML in Software Engineering) addresses these types of systems [34]. It is an approach combining UML class-based models with OWL ontologies to leverage the unique and potentially complementary strengths of the two. TwoUse consists of an integration of the MOF-based metamodels for UML and OWL, the specification of dynamic behavior referring to OWL reasoning and the definition of a joint profile for denoting hybrid models as well as other concrete syntaxes.

Figure 10 presents a model-driven view of the TwoUse approach. TwoUse uses UML profiled class diagrams as concrete syntax for designing combined models. The UML class diagrams profiled for TwoUse are input for model transformations that generate TwoUse models conforming to the TwoUse metamodel. The TwoUse metamodel provides the abstract syntax for the TwoUse approach, since we have explored different concrete syntaxes. Further model transformations take TwoUse models and generate the OWL ontology and Java code.

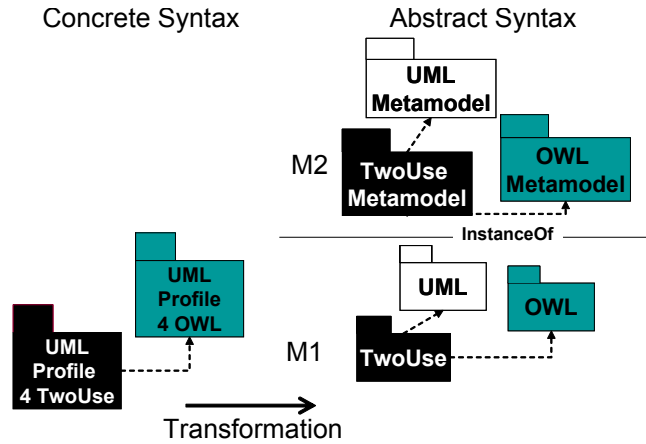


Fig. 10. Example of M2 Integration Bridge

TwoUse allows developers to raise the level of abstraction of business rules previously embedded in code. It enables UML modelling with semantic expressiveness of OWL DL. TwoUse achieves improvements on the maintainability, reusability and extensibility for ontology based system development.

M2 Transformation Bridge A *transformation bridge* describes a (physical) transformation between models in layer M1. The models are kept separately in both modelling spaces. The information is moved from one model to the model in the other modelling space according to the transformation bridge. With respect to the example depicted in Figure 9, a process model like a UML Activity Diagram is transformed to an OWL ontology. The transformation rules or patterns are defined by the bridge. Thus, having a process model as an ontology we can provide services for reasoning on the semantics of process models.

5 Ontology Reasoning for Structural Modelling

In this section, we show how bridge (domain-specific) modelling languages with ontology languages. In particular, we show how the abstract syntax of modelling languages can be restricted by the use of integrated ontology languages. The challenge is to formally define language metamodels with integrated ontology-based axioms and expressions that allow for reasoning on the metamodel itself and all conforming models.

In recent works (e.g. [35, 36]) we have exemplified, that ontology reasoning for structural modeling is mainly based on combining a metamodel (e.g. Ecore) with the metamodel of an ontology language. Having a metamodel bridge with the OWL metamodel a language designer is able, for example, to define formal well-formedness constraints based on OWL to restrict the use of concepts provided by the metamodel. The metamodel and conforming models are transformed to an ontology representation for reasoning on the structure of models that is prescribed by the language metamodel (and additional well-formedness constraints).

5.1 Challenges and Tasks in Structural Modelling

*Comarch*¹, one of the industrial partners in the *MOST project*² has provided the running example used in this section. It is an excerpt of a use case where telecommunication providers want to model configurations for telecommunication systems.

Let us elaborate the following example: The general physical structure of a Device consists of a Bay which has a number of Shelves. A Shelf contains Slots into which Cards can be plugged. Logically, a Shelf with its possible Slots and Cards is stored as a Configuration.

¹ <http://www.comarch.com/>

² <http://www.most-project.eu/>

Figure 11 depicts the development of a domain model for physical devices by a language user. Firstly (*step 1*), the language user starts with an instance of the general concept `Device`. A device requires at least one configuration. Thus the language user plugs in a `Configuration` element into the device.

In *step 2*, the language user adds exactly three slots to the device model. At this point, the language user wants to verify whether the configuration satisfies the domain restrictions, which is done, for example, by invoking a query against the current physical device model.

After adding three slots to the model of the physical device, the language user plugs in some cards to complete the end product (*step 3*). Knowing which cards and interfaces should be provided by the device, he may insert an `SPAInterfaceCard` for 1-Gbps broadband connections, a `SupervisorEngine720` card for different IP and security features and a controller for swapping cards at runtime (`HotSwapController`). At this point, the language user wants to use reasoning services.

The domain-specific language (DSL) defines the knowledge about which special types of cards are provided by a `Configuration`. Having the information that its instance is connected with three slots, the replacement of the `Configuration` type by the more specific type `Configuration7603` is recommended to the language user as result of activating the reasoning service. Moreover, the language user is informed how this suggestion takes place and about restrictions related to such a configuration.

Since it has been inferred that the device has the `Configuration7603`, in *step 4*, the available reasoning service for the `Device` element infers that the device is one of type `Cisco7603`. The necessary and sufficient condition to be a `Cisco7603` are checked by reasoning services.

Steps *2a* and *2b* show a second path in the scenario of modelling a physical device where debugging comes into play. After creating elements for a device, a configuration and slots, the language user plugs into one slot a `HotSwappableOSM` card and into the remaining slots two `SPAInterface` cards (*step 2a*). Here an inconsistency occurs. The language user needs an explanation why there is a problem with this configuration. Explanation services give the information that each configuration must have a slot in which a `SuperVisor720` card is plugged in. Having a correct configuration, the DSL user can continue with *steps 3 and 4* as described above.

5.2 Bridges for Structural Modelling Languages

As a bridge for structural modelling we propose an M3 integration bridge, where the bridge is defined at the M3 layer by an integration of a metamamodel and ontology language (cf. Figure 7 and 12).

In particular, we consider an M3 integration bridge where the metamodeling language KM3 [37] is combined with the OWL2 metamodel. An overview of bridge definition and use is depicted in Figure 12.

The bridge itself is used at the M2 layer. Here the language designer is able to define constraints and expressions within his metamodel to formally define

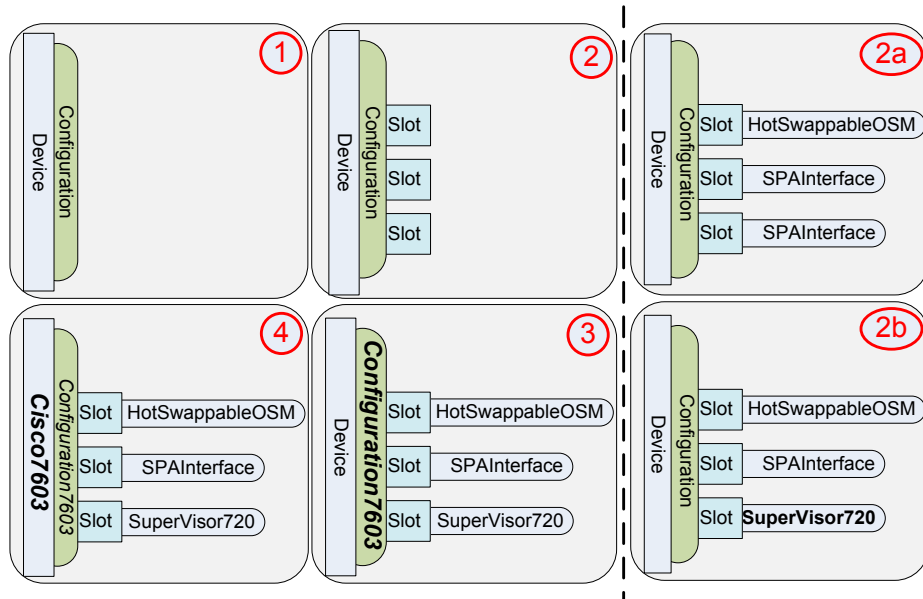


Fig. 11. Modelling a physical device in four steps (M1 layer)

the abstract syntax of a new modelling language. The modelling language itself is used at the M1 layer. The constraints and expressions restrict the creation of domain models which are instances of the metamodel.

5.3 Defining an M3 Integration Bridge

In the following, we introduce the idea of integrated metamodeling which gives an example of an M3 integration bridge. First we present some excerpts of an M3 metamodel, which defines the abstract syntax for metamodels with integrated ontology annotations.

Here, we integrate the existing KM3 [37] metamodel, a simplified subset of EMOF, with the OWL 2 metamodel at the M3 layer. Thus, we can provide a new metamodeling language which allows for integrated modeling of both KM3-based metamodel and OWL ontologies.

Furthermore, we present some parts of the concrete syntax that are used by the language designer to implement metamodels. Like in Section 2.1, we use a textual syntax that is easy to implement, as an extension of the concrete syntax of KM3, and might provide some productivity in coding metamodels. Thus we give the idea of how to define such a syntax using EBNF grammar rules.

Abstract Syntax. The integrated M3 metamodel enables language designers to describe M2 classes together with OWL annotations. The KM3

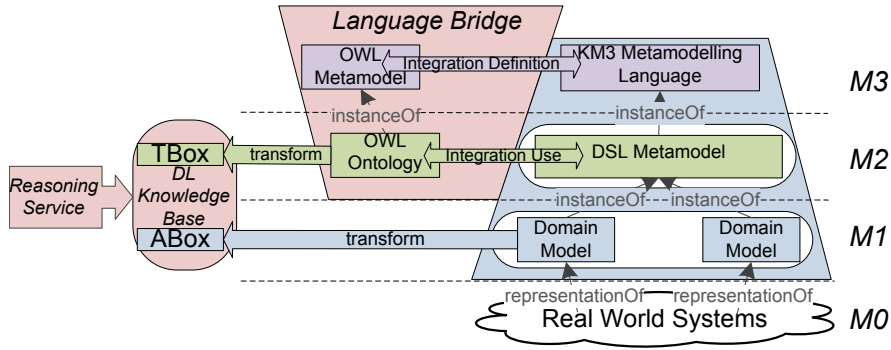


Fig. 12. M3 Integration Bridge on KM3 and OWL

metametamodel allows for specifying behavioral and structural features of classes. The OWL metamodel provides for the use of OWL primitives and corresponding sound and complete reasoning over these primitives.

Our integration approach considers classes from the OWL2 metamodel and a further (meta-) metamodel and combines corresponding classes by creating a so-called TwoUSE-class which is a specialization of both classes and thus inherits the properties of both.

Figure 13 depicts an excerpt of the integrated metametamodel. Here, the central concepts are the TUClass, the TUAttribute, and the TUREference.

The integration between the KM3 and the OWL 2 metamodels is done by applying the class adapter design pattern [38] into KM3 metaclasses that are similar to OWL 2 metaclasses (for similarities between OWL and class-based modelling languages see [39]; for integration methods of (meta-)modelling language and ontology languages see [27, 40]).

TUClass is a specialization of KM3 Class and OWLClass and inherits their behavior. Using the TwoUses classes TUAttribute and TUREference we connect OWL DataProperty with KM3 Attribute and OWL ObjectProperty with KM3 Reference respectively. Using the TwoUse classes for references and attributes of KM3 we ensure that all instances have the behavior of an OWL Object Property or OWL Data Property respectively.

Concrete Syntax. In the following, we present some parts of a concrete syntax which extends the one from KM3 [37]. In this section, we only want to give the idea of how to extend the KM3 syntax by OWL class axioms and object property axioms. Because the KM3 concrete syntax is based on a grammar, in the following we define rules in EBNF and give an example of using them to code metamodels.

OWL Class Axioms. The first line of Listing 1.4 shows rules for the definition of a class. Each class is optionally abstract, is defined by the keyword class,

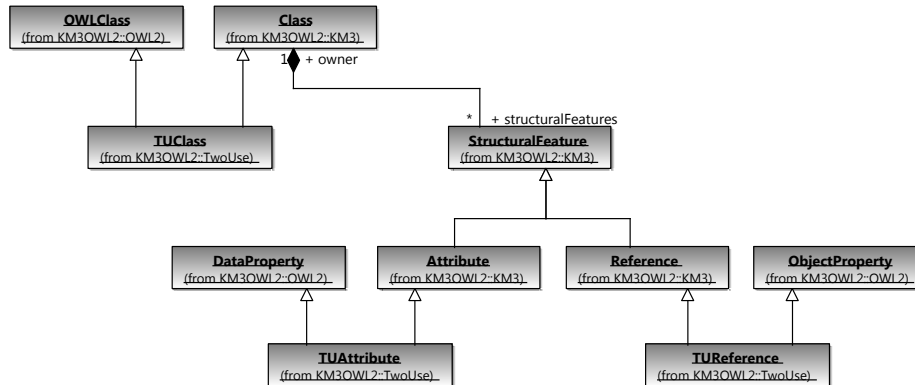


Fig. 13. Excerpt of KM3+OWL 2 metamodel (M3 layer)

has a name and a list of super types. Between the curly brackets a list of class features are defined such as attributes and references. In our special case the grammar rule is extended by the new non-terminal `classAxioms` which optionally produces a list of OWL 2 class axioms (see Listing 1.4, line 2). The non-terminal `ClassAxiom` is used and is defined by an OWL 2 textual concrete syntax.

Listing 1.4. Adopting OWL Class Axioms

```

1 class = ["abstract"] "class" name [supertypes] [classAxioms] "{" features "}";
  classAxioms = ClassAxiom { " " ClassAxiom };

```

Listing 1.5 presents an example of using OWL class axioms in combination with the declaration of classes in KM3. In line 1, the class `Configuration` is defined having a reference called `hasSlot` to a further class `Slot`. Line 4 depicts the head of the class `Configuration7603` which has the super type `Configuration` and thus inherits the feature `hasSlot`. After declaring the super types the definition of an OWL class axiom follows. It defines that the class `Configuration7603` is equivalent with the anonymous OWL class `restrictionOn hasSlot with exactly 3 Slot`, the instances which are connected with exactly 3 further instances of type `Slot` using the `hasSlot` reference.

Listing 1.5. Example of using OWL Class Axioms (M2 layer)

```

1 class Configuration extends restrictionOn hasSlot with min 1 Slot {
  reference hasSlot [1-*]: Slot;
}
4 class Configuration7603 extends Configuration equivalentTo restrictionOn hasSlot with exactly 3 Slot {
5 }

```

Object Property Axioms. Listing 1.6 presents the grammar rules that adopt OWL object property axioms on references. A general reference feature in KM3 is declared by the keyword `reference`, a name, a multiplicity, the optional indication if the reference acts as a container, and a type where the reference points

to. In addition to this declaration, we introduce the new non-terminal `objectPropertyAxioms` which is defined in line 2 of the Listing and produces a list of `ObjectPropertyAxiom`. Again this non-terminal is defined in our OWL 2 natural style syntax. Thus a list of OWL object property axioms can be append to the declaration of KM3 references.

Listing 1.6. Adopting OWL Object Property Axioms

```
1 reference = "reference" name multiplicity isContainer ":" typeref "oppositeOf"
name [objectPropertyAxioms] ",";
objectPropertyAxioms = ObjectPropertyAxiom { "," ObjectPropertyAxiom};
```

Listing 1.7 depicts an example of using OWL object property axioms together with the reference features of KM3. Here the two classes `Slot` and its subclass `Slot7609-2` are defined. `Slot` contains a reference called `hasCard`, `Slot7609-2` contains a reference called `hasInterfaceCard`. `hasCard` points to elements of type `Card`, `hasInterfaceCard` points to elements of type `CiscoInterface`, a subclass of `Card`. Using OWL object property axioms we state that `hasInterfaceCard` is a sub property of `hasCard`. Thus, if an instance of `CiscoInterface` is connected via the reference `hasInterface` with some interface we can infer that this instance is also connected via the reference `hasCard` with the interface instance. Thus, the restriction `MinCardinality(1 hasCard)` holds.

Listing 1.7. Example of using OWL Object Property Axioms (M2 layer)

```
1 class Slot equivalentTo restrictionOn hasCard with min 1 Card{
  reference hasCard [1-*]: Card;
}
5 class Slot7609-2 extends Slot{
  reference hasInterfaceCard [1-*]: CiscoInterface subpropertyOf hasCard;
}
```

5.4 Using an M3 Integration Bridge

In the following, we show how to adopt reasoning services on structural modelling languages that are used for modelling configurations. In particular, we consider languages for modelling configurations of physical devices as introduced in Section 5.1. In the following, we exemplify the bridging approach which tackles the design and use of the PDDSL (Physical Device DSL). Further bridging approaches between configuration languages and ontology technologies can be found in [41].

Designing PDDSL. Listing 1.8 depicts an example of an integrated PDDSL language. Here, we define constraints and restrictions within the metamodel definition.

To design the metamodel of PDDSL we used the combined metamodeling language consisting of KM3+OWL. The metamodel represent the abstract syntax of the PDDSL as well as well-formedness constraints for the *M1-layer*. This additional constraints are useful to define the syntactic structure of the domain

model at the M1-layer as well as to indicate constraints that apply at the level of the modelling language itself (M2-layer).

Listing 1.8. Example of defining an integrated metamodel for PDDSL

```

1 class Device {
  reference hasConfiguration [1-*]: Configuration;
}

5 class Cisco7603 extends Device, equivalentWith restrictionOn hasConfiguration
with min 1 Configuration7603 {
}

  class Configuration extends IntersectionOf(restrictionOn hasSlot with min 1
10 Slot, restrictionOn hasSlot with some restrictionOn hasCard with some
  SuperVisor720){
  reference hasSlot : Slot;
}

15 class Configuration7603 extends Configuration, equivalentWith
IntersectionOf(restrictionOn hasSlot with exactly 3 Slot, restrictionOn hasSlot
with some restrictionOn hasCard with some UnionOf(HotSwappableOSM,
  SPAinterfaceProcessors) { }

20 class Slot {
  reference hasCard [1-*]: Card;
}

  class Card {
25 }

  class SuperVisor720 extends Card {
}

30 class SPAinterfaceProcessors extends Card {
}

  class HotSwappableOSM extends Card {
}

```

Using PDDSL. Having a PDDSL metamodel specified by the language designer, the language user is able to build different domain models. These domain models describe possible configurations of physical network devices. Possible domain models are depicted in Figure 11. During domain modelling, the language user experiences several benefits. In development environments these benefits would be implemented by reasoning services (cf. Section 3) which are automatically invoked. The language user needs no background information on how the reasoning services work and how they are connected with the knowledge base. The reasoning engine returns suggestions and explanations to the DSL user.

To allow for such services, the ABox and TBox of an ontology are extracted from the domain model (*M1 layer*) and the integrated metamodel (*M2 layer*), respectively. In the following, we will consider more precisely the services that are provided to user.

Reasoning Services for Modelling Configurations In the following, we present some concrete services that are used by language users.

Detecting Inconsistencies in Domain Models To detect inconsistencies in domain models the PDDSL metamodel is transformed into a description logics TBox. All classes in the PDDSL metamodel are transformed to a concept in a description logics TBox. All references in the metamodel are represented by roles (object properties) in the TBox. The domain model itself is transformed to a description logics TBox to check whether the domain models are consistent we use the *consistency checking reasoning service*. With regard to the example the service returns that the model in Figure 11 (2a), is inconsistent, because the configuration does not contain the mandatory supervisor card (as it is defined in the metamodel in Figure 1.8).

Finding and Explaining Errors in Configuration Instances Language users that use the PDDSL language (which describes sets of possible configurations) to create domain models (which describe concrete configurations) require debugging of domain models and explanation of errors and inconsistencies. More precisely, a user of PDDSL wants to identify illegal configurations, wants to get explanations why the configuration of a device is inconsistent and wants to get suggestions how to fix the configuration.

To validate configuration models against the PDDSL metamodel, the model is transformed into a description logics ABox. The TBox is built by the PDDSL metamodel which contains all metaconcepts and additional constraints. The TBox and ABox build the description logics knowledge base. Using the consistency checking reasoning service, we can check if the knowledge base is consistent and thus validate the configuration

If we want to detect invalid cards in a configuration, we need some explanations why the model is inconsistent with regard to its metamodel. Explanations are provided by explanation services introduced in Section 3.4.

Figure 11 (2a) depicts an example of a domain model with an invalid configuration. Using the consistency checking reasoning service and requesting some explanation, the reasoner delivers the answer shown in Listing 1.9:

Listing 1.9. Explanation for domain model inconsistency

```

1 CHECK CONSISTENCY

Consistent: No

5 Explanation:
  Configuration7603 subClassOf Configuration
  config76 type Configuration7603
  card.SPA1 type SPAInterface
  Configuration subClassOf hasSlot min 1 Slot
10      and hasSlot some hasCard some Supervisor720
  card.SPA2 type SPAInterface
  card.HS type HotSwappableOSM
  Supervisor720 subClassOf Card

```

The reason for the inconsistency here is the missing supervisor card, which must be part of every configuration. Since `config76` has as type `Configuration7603` which is a subclass of `Configuration` it must be connected via a slot with some card of type `Supervisor720`. This is not fulfilled because all cards (`card_SPA1`,

card_SPA2, card_HS) plugged into the configuration are either of type HotSwappableOSM or type SPAInterface.

Classification of Device Configurations Language users often start modelling with general concepts, e.g. with model elements of type `Device` or `Configuration`, depicted in Figure 11 (1). To classify a device or configuration, we have to again transform the instances together with its links into a description logics ABox. The TBox is built by the PDDSL metamodel.

Since description logics allow for simultaneously reasoning on the model (TBox) and instance layer (ABox) we are able to use the *classification reasoning service* (introduced in Section 3.2) to compute all possible types of the individual `config76` with regard to all other individuals and relations in the ABox. The result is the more specific type `Configuration7603` of the individual `config76`. Again the individual `device76` can be classified. The most specific type here is `Cisco7603`.

6 Ontology Reasoning for Behaviour Modelling Languages

In this section, we demonstrate the representation of behaviour models in OWL and applications of reasoning services in order to provide model management services like the retrieval of process models based on a process description. We apply our approach on process models, represented by UML Activity Diagrams.

In order to use ontology reasoning for process models, a first step is to build a model bridge from process models (software models) in a UML-like representation to an ontology (TBox). The architecture of a model bridge, or in particular of a process model bridge is depicted in Figure 14. The model bridge is defined in the metamodeling layer M2 and is used in layer M1 to transform or integrate model entities on layer M1. In this section, we consider a transformation bridge.

We present our process model bridge that defines a transformation from process models given as UML activity diagrams to on OWL ontology (TBox). This requires a thorough consideration of the entities that are represented in process models, their relations like control flow relations and how they are transformed to OWL ontologies. A challenge in this task is to capture the semantics of process models like activity ordering and flow conditions in the ontology.

6.1 Transformation of Process Models into Semantic Representations

Process models capture the dynamic behaviour of an application or system. In software modelling, they are represented by graphical models like BPMN Diagrams or UML Activity Diagrams. The metamodels of both are instances of Ecore meta-metamodels. The two metamodels provide flexible means for describing process models for various applications. However, due to their flexibility

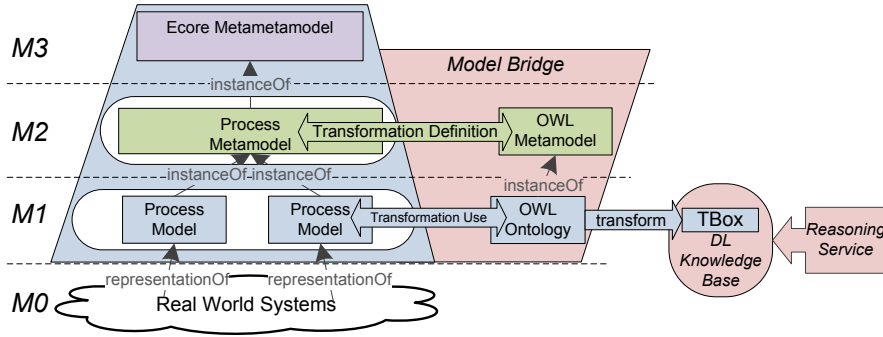


Fig. 14. Process Model Bridge

further modelling constraints and semantic descriptions are required for a clearer representation of the intended meaning.

We have identified the following additional modelling characteristics for process models in the software modelling space which are analysed in detail in [44]. (1) A semantic representation of control flow dependencies of activities within a process, i.e. execution ordering of activities in a control flow. Such constraints allow for the description of order dependencies e.g., an activity requires a certain activity as a predecessor or successor. (2) It is quite common in model-driven engineering to specialise or refine a model into a more fine-grained representation that is closer to the concrete implementation (cf. [45]). In process modelling, activities could be replaced by sub-activities for a more precise description of a process. Hence, modelling possibilities for sub-activities and also for control flow constraints of these sub-activities should be supported. (3) Quite often, one may formulate process properties that cover modality, i.e. to express a certain property like the occurrence of an activity within a control flow is optional or unavoidable in all possible process instances (traces).

The next subsection gives an overview of the model bridge from a UML activity diagram to an OWL ontology including a discussion of design decisions. The transformation bridge describes how a process model on modelling layer M1 is transformed from a UML representation to an ontology. We demonstrate how entities of an activity diagram like activities, gateways and conditions are modelled in an OWL ontology. Furthermore, we demonstrate the usage of logical representation of a process model in order to achieve the afore-mentioned modelling possibilities in software process modelling. We use the OWL representation of process models in combination with reasoning services to retrieve processes and check process constraints in the second part of this section.

Mapping Process Models to OWL A process model describes the set of all legal process runs or traces. Activities are represented by OWL classes and a process is modelled as a complex expression that captures all activities of the process. A process run is an instance of this complex class expression in

OWL. The process models are described in OWL DL, as syntax we use the DL notation. Basic modelling principles and transformation patterns from UML activity diagrams to OWL are given in Table 2.

Construct	UML Notation	DL Notation
1. Start		$Start_i$
2. End		End_i
3. Activity		$Receive\ Order$
4. Edge		TO_i
5. Process P		$P \equiv Start_i \sqcap \exists_{=1} TO_i.$ $(ReceiveOrder \sqcap \exists_{=1} TO_i.End_i)$
6. Flow		$ReceiveOrder \sqcap \exists_{=1} TO_i.FillOrder$
7. Decision		$ReceiveOrder \sqcap \exists_{=1} TO_i.$ $((RejectOrder \sqcup FillOrder)$ $\sqcap \exists_{=1} TO_i.CloseOrder)$
8. Condition		$ReceiveOrder \sqcap \exists_{=1} TO_i.$ $((FillOrder \sqcap \kappa_{OrderAccepted}) \sqcup$ $(Stalled \sqcap \neg\kappa_{OrderAccepted}))$
9. Fork and Join		$ReceiveOrder \sqcap \exists TO_i.$ $(ShipOrder \sqcap \exists_{=1} TO_i.CloseOrder)$ $\sqcap \exists TO_i.(SendInvoice \sqcap$ $\exists_{=1} TO_i.CloseOrder) \sqcap = 2 TO_i$
10. Loop		$Loop_j \sqcap \exists_{=1} TO_i.FillOrder,$ $Loop_j \equiv ReceiveOrder \sqcap \exists_{=1} TO_j.$ $(Loop_j \sqcup End_j)$

Table 2. Transformation to OWL.

Control flow relations between activities are represented by object properties in OWL, i.e. by the object property TO_i (Table 2, No. 4). In order to allow for process composition and refinement in combination with cardinality restrictions, the roles for each process (TO_i) are distinguished from each other. All roles TO_i are defined as sub-roles of TO in order to simplify the retrieval. The object property TO and therefore also the sub-properties TO_i are sub-properties of the transitive property TOT . The property TOT is used to express transitive connections of activities to their predecessor and successor activities with an arbitrary number of activities between them. This kind of hierarchical structuring of object properties is similar to the best practice modelling style for relation pattern as in the DOLCE plan extension [46].

A process is composed by activities and is described in OWL by axioms as shown in No. 5. An axiom defines a process as one complex DL expression capturing all activities that occur in the process. It starts with $Start_i$ followed by a sequence of composed activities. The last activity is the concept End_i .

The control flow (No. 6) is a class expression in OWL like $ReceiveOrder \sqcap \exists TO_i.FillOrder$ meaning the activity $ReceiveOrder$ is directly followed by the activity $FillOrder$. We use concept union for decisions (No. 7). The non-deterministic choice between the activity $RejectOrder$ and $FillOrder$ is given by the class expression $\exists TO_i.(RejectOrder \sqcup FillOrder)$. The activity $CloseOrder$ merges the flows to one outgoing sequence flow. Deterministic choices and exclusive decisions are represented by using different (disjoint) flow conditions.

Flow conditions (No. 8) are assigned to the control flow. The semantics of a flow condition is a restriction on all instances that satisfy the target activity of this flow. Hence, the target instances of the flow are instances of the activity $FillOrder$ and they have to satisfy the $OrderAccepted$ condition, i.e. they are also instances of the class $\mathcal{K}_{OrderAccepted}$.

A loop (No. 10) is a special kind of decision. An additional OWL class $Loop_j$ for the subprocess with the loop is introduced to describe multiple occurrences of the activities within the loop. Parallel executions are represented by intersections (No. 9). It is an explicit statement that an activity have multiple successors simultaneously.

6.2 Ontology Reasoning for Process Retrieval

In the previous subsection, we used the expressive power of OWL to provide a semantic representation of process models. Based on this representation, we describe queries in DL and use reasoning services to retrieve processes and process information.

Queries are general and incomplete process descriptions that specify the core functionality of a process of interest. The query result contains all processes of the knowledge base or process repository that satisfy the query specification.

The following example demonstrates a query for process that executes the activity $FillOrder$ before $MakePayment$ with an arbitrary number of activities between them:

$$Q \equiv \exists TOT.(FillOrder \sqcap \exists TOT.MakePayment)$$

To exploit the enriched semantic process representation as presented in the previous subsection, the process retrieval covers three non-disjoint patterns of process structuring and control flow information. (i) A query describes the relevant ordering conditions like which activity has to follow (directly or indirectly) another activity. The transitive object property TOT is used to indicate the possibly indirect connection of the activities. (ii) Besides ordering constraints, semantic query processing allows the retrieval of processes that contain specialised or refined activities, i.e. the process retrieval takes into account the terminological knowledge of the processes like hierarchical structuring of activities. For instance, the result of the demonstrated query also contains all processes that contains sub-activities of $FillOrder$ and $MakePayment$, satisfying the ordering condition. The corresponding class expressions in the OWL model are specialisations of the class expression given by the query expression. (iii) Finally, the

usage of the queries allows handling of modality for activity occurrences in a process, like a query that expresses whether the activity *MakePayment* has to occur in each process or might occur only in some process.

In general, there are various capabilities of DL reasoning or entailment regimes in order to find process models that satisfy or match the process description given by the DL query expression. Again, the query description is also a (general) description of a process. For a more comprehensive overview of different DL inference or entailment methodologies, we refer the reader to [47]. Here, we present process retrieval for two inference methodologies: concept subsumption and concept satisfiability.

Retrieval by Concept Subsumption In this paragraph, we use entailment of concept subsumption in order to retrieve process models. The process description given by the query is a general and abstract process model. The more complex DL expressions that represent processes in the knowledge base (*KB*) are more specific with respect to the representation in OWL, i.e. containing further activities, intersections of further parts of the process and (additional) conditions. Hence, all retrieved processes that satisfy the process description of the query are specialisations in OWL of the general process description given by the query, i.e. these processes are subsumed by the more abstract query process.

As an example, we consider a query that searches for all processes that execute the *MakePayment* activity and its direct successor is the activity *AcceptPayment*. This query (*Q*) is described by the following DL expression $Q \equiv \exists TOT.(MakePayment \sqcap \exists TO.AcceptPayment)$. The direct successor relation of *MakePayment* and *AcceptPayment* is expressed by the non-transitive property *TO*. Consider the following process that satisfy this query description which is a fragment of the process depicted in Figure 5. $OrderProcess \equiv Start_1 \sqcap \exists_{=1} TO_1.(FillOrder \sqcap \exists TO_1.(ShipOrder \sqcap \exists_{=1} TO_1.End_1) \sqcap \exists TO_1.(MakePayment \sqcap \exists_{=1} TO_1.(AcceptPayment \sqcap \exists_{=1} TO_1.End_1))) \sqcap = 2TO_1)$

The knowledge base entails that *OrderProcess* is subsumed by the query process *Q*: $KB \models OrderProcess \sqsubseteq Q$

Retrieval by Concept Satisfiability A weaker entailment regime is checking the satisfiability of concept intersection. The concept intersection is the intersection of the query process description *Q* and a process from the knowledge base (*KB*), like the *OrderProcess*. The result of such a query *Q* are all process models of the knowledge base for which the intersection with the query is a satisfiable concept.

Obviously, the result set is in general larger than in the previous, stronger entailment methodology. The above mentioned process *OrderProcess* would also satisfy this query *Q* if there are no further statements like the disjointness of activities in the knowledge base. The result of the query *Q* has to satisfy the following condition:

$KB \cup \{p : (OrderProcess \sqcap Q)\}$ is satisfiable. The p in the query is an instance, i.e. a process run.

The difference of this weaker inference notion becomes more evident if we are looking for a process that is not in the knowledge base, e.g., a different quantifier is used in order to restrict the successors of an activity to a certain given activity like the following query.

$$Q \equiv \exists TOT.(MakePayment \sqcap \forall TO.AcceptPayment)$$

Here, we are looking for all processes that have the activity *MakePayment* in the control flow and the successor of this activity is restricted to *AcceptPayment*. We assume there is no such restriction for a process in the knowledge base. Hence, the stronger inference would not retrieve any process model. However, this weaker entailment would still retrieve the example process *OrderProcess*, since the intersection $(OrderProcess \sqcap Q)$ is satisfiable in the knowledge base KB . Again, we assume that no further restriction like disjointness of activities or conditions of process flows are in the knowledge base.

6.3 Summary

In this section, we demonstrated how to use OWL for modelling of process models in order to provide reasoning services to validate process models and retrieve processes and process information. A prerequisite is a transformation bridge that maps on the M1 layer UML activity diagrams to an OWL ontology. The transformation bridge we provided is quite generic and can be adapted to other process modelling languages like the Business Process Modeling Notation (BPMN).

7 A Toolkit for MDE with Ontology Technologies

The TwoUse Toolkit is modelling environment filling the gap between MDE and ontology technologies. It is an implementation of current OMG and W3C standards for developing ontology-based software models and model-based OWL ontologies. It is a model-driven tool to bridge the gap between Semantic Web and Model Driven Software Development.

The TwoUse Toolkit has two User Profiles: model-driven software developers and OWL ontology engineers. The TwoUse Toolkit provides the following functionality to model-driven software developers:

- Describe classes in UML class diagrams using OWL class descriptions.
- Semantically search for classes, properties and instances in UML class diagrams.
- Model variability in software systems using OWL classes.
- Design business rules using the UML Profile for SWRL.
- Extent software design patterns with OWL class descriptions.

- Make sense of UML class diagrams using inference explanations.
- Write OWL queries using SPARQL, SAIQL or the OWL query language based on the OWL Functional Syntax using the query editor with syntax highlighting.
- Validate refinements on business process models.

To OWL ontology engineers, The TwoUse Toolkit provides the following functionalities:

- Graphically model OWL ontologies and OWL safe rules using OMG UML Profile for OWL and UML Profile for SWRL.
- Graphically model OWL ontologies and OWL Safe Rules using the OWL 2 Graphical Editor.
- Graphically model and store ontology design patterns as templates.
- Write OWL queries using SPARQL, SAIQL or the OWL query language based on the OWL Functional Syntax using the query editor with syntax highlighting.
- Specify and safe OWL ontologies using the OWL 2 functional syntax with syntax highlighting.
- specify OWL ontology APIs using the agogo editor.

We have implemented the TwoUse Toolkit in the Eclipse Platform using the Eclipse Modelling Framework [49] and is available for download on the project website³.

8 Related Work

Modelling complex systems usually requires different modelling languages. Hence, the need of integrating them is apparent. Because of semantic overlap of languages, where synergies can be realized by defining bridges, ontologies provide the chance of semantic integration of modelling languages [50].

In the following, we group related approaches into three categories: Firstly, we present approaches where structural languages are bridged. Secondly, bridges for behavioural languages are described and finally, related work on OWL modelling for building mappings between models (model versions) is discussed.

Among approaches of bridges between ontology languages and structural modelling language, one can use languages like F-Logic or Alloy to formally describe models. In [51], a transformation of UML+OCL to Alloy is proposed to exploit analysis capabilities of the Alloy Analyzer [52]. In [53], 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 (e.g. [54, 55]) have explored the usage of F-Logic to

³ <http://code.google.com/p/twouse/>

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 language designers to specify class descriptions *à la* OWL together with MOF/OCL, increasing expressiveness.

There are various approaches that build bridges between behavioural modelling languages and ontologies. Here, we only mention those that are related to process modelling in OWL. The OWL-S process model [56] describes processes in OWL. The process specification language [57, 58] allows formal process modelling in an ontology. Process models are represented in OWL in combination with petri nets in [59] and in Description Logics for workflows in [60]. Compared to our demonstrated model, these approaches either lack in an explicit representation of control flow dependencies in combination with terminological information of activities like a hierarchical structuring of activities, or retrieval of processes with respect to control flow information is only weakly supported.

9 Conclusion

In this chapter, we described how ontology technologies can be adopted on model-driven engineering. The main artifacts in MDE are modelling languages and transformations. We have shown, that both can be supported by ontology technologies.

In this chapter, after the introduction, we first started with foundations on model-driven engineering, covering modelling languages, transformations and a short characterisation of challenges in model-driven engineering. We continued the chapter with an overview of the Web Ontology Language, ontological modelling and reasoning services. Here, we took a deeper look at the reasoning services and how they could be used in MDE. We first started with standard reasoning services (like consistency and satisfiability checking, or classification) and querying services. All these services are adopted on an ontology which is a representation of the metamodel of a modelling language and conforming models created by the language user. Besides standard reasoning services, we considered explanation and repairing services, where explanation services can inform about debugging relevant facts and repairing services give advises how to repair inconsistent models. A last ontology-based service is the linking service which is used to combine different modelling languages and its models for reasoning on one single representation of a system since systems are described by many different software languages.

In the subsequent section, we showed how to integrate ontology languages with standard metamodeling languages (like, for example, KM3) for an integrated modelling of metamodels together with the abstract syntax component of a modelling language. The integrated metamodels consist on the one side of usual concepts for classes or references, but on the other side they can contain additional semantic expressions, constraints and axioms which are based on

OWL. Later they are extracted for adopting ontology technologies like different reasoning services on modelling languages. We described the integration of software and ontology languages by introducing language and model bridges.

The support of modelling languages by ontology technologies was separated into support for structural modelling languages and support for behavioural modelling languages. In the first case, we mainly considered the Ecore meta-modelling language and UML, as well as languages for describing configurations. Reasoning services here allow for reasoning on the structure of models, which are prescribed by the metamodel of a modelling language. Possible services are the consistency checking service or the classification service. In the case of behavioural modelling languages, we considered process modelling languages. Here the ontology language was bridged with the process modelling language to describe the semantics of the language and the behaviour of processes.

Before concluding the chapter, we presented the main functionalities of the TwoUse toolkit which implements most of the approaches presented in this paper, followed by an overview of related work.

References

1. Mellor, S., Clark, A., Futagami, T.: Model-driven development. *IEEE software* **20**(5) (2003) 14–18
2. Atkinson, C., Kuhne, T.: Model-driven development: a metamodeling foundation. *IEEE software* **20**(5) (2003) 36–41
3. 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)
4. Ebert, J.: Metamodels Taken Seriously: The TGraph Approach. In Kontogiannis, K., Tjortjis, C., Winter, A., eds.: 12th European Conference on Software Maintenance and Reengineering, Piscataway, NJ, IEEE Computer Society (2008)
5. Bildhauer, D., Riediger, V., Schwarz, H., Strauss, S.: grUML-An UMLbased Modeling Language for TGraphs. to appear in *Arbeitsberichte Informatik, Universität Koblenz-Landau* (2008)
6. OMG: Meta Object Facility (MOF) Core Specification. <http://www.omg.org/docs/formal/06-01-01.pdf> (January 2006)
7. OMG: UML Infrastructure Specification, v2.1.2. *OMG Adopted Specification* (2007)
8. Budinsky, F., Brodsky, S., Merks, E.: *Eclipse modeling framework*. Pearson Education (2003)
9. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: *Satellite Events at the MoDELS*. Volume 3844., Springer (2006) 128
10. OMG: MOF QVT Final Adopted Specification. *Object Modeling Group*. (June 2005)
11. Kalnins, A., Barzdins, J., Celms, E.: Model transformation language MOLA. *Model Driven Architecture* 62–76
12. Wagner, R.: Developing Model Transformations with Fujaba. In: *Proc. of the 4th International Fujaba Days*. (2006) 06–275
13. Wende, C.: *Ontology Services for Model-Driven Software Development*. MOST Project Deliverable (November 2009) www.most-project.eu.

14. Motik, B., Patel-Schneider, P.F., Grau, B.C.: OWL 2 Web Ontology Language Direct Semantics. <http://www.w3.org/TR/owl2-direct-semantics> (October 2009)
15. Parsia, B., Sirin, E.: Pellet: An OWL DL Reasoner. In: Proc. of the 2004 International Workshop on Description Logics (DL2004). Volume 104 of CEUR Workshop Proceedings. (2004)
16. Prud'hommeaux, E., Seaborne, A.: SPARQL query language for RDF (working draft). Technical report, W3C (March 2007)
17. Schneider, M.: SPARQLAS: Writing SPARQL Queries in OWL Syntax. Bachelor thesis, University of Koblenz-Landau (2010) German.
18. Glimm, B., Parsia, B.: SPARQL 1.1 Entailment Regimes. Working draft 26 January 2010, W3C (2010) Available on <http://www.w3.org/TR/sparql11-entailment/>.
19. Knublauch, H., Fergerson, R., Noy, N., Musen, M.: The Protégé OWL plugin: An open development environment for semantic web applications. *Lecture notes in computer science* (2004) 229–243
20. Kalyanpur, A., Parsia, B., Sirin, E., Hendler, J.: Debugging unsatisfiable classes in OWL ontologies. *Web Semantics: Science, Services and Agents on the World Wide Web* **3**(4) (2005) 268–293
21. Kalyanpur, A., Parsia, B., Horridge, M., Sirin, E.: Finding all justifications of OWL DL entailments. *Lecture Notes in Computer Science* **4825** (2007) 267
22. Reiter, R.: A theory of diagnosis from first principles. *Artificial Intelligence* **32**(1) (1987) 57–95
23. Horridge, M., Drummond, N., Goodwin, J., Rector, A., Stevens, R., Wang, H.: The Manchester OWL Syntax. In: OWLED2006 Second Workshop on OWL Experiences and Directions, Athens, GA, USA (2006)
24. Kalyanpur, A.: Debugging and Repair of OWL Ontologies. PhD thesis, University of Maryland, College Park (2006)
25. Kalyanpur, A., Parsia, B., Sirin, E., Cuenca-Grau, B.: Repairing Unsatisfiable Concepts in OWL Ontologies. *The Semantic Web: Research and Applications* 170–184
26. Euzenat, J., Shvaiko, P.: *Ontology matching*. Springer-Verlag, Heidelberg (2007)
27. Walter, T., Ebert, J.: Combining DSLs and Ontologies using Metamodel Integration. In: *Domain-Specific Languages*. Volume 5658 of LNCS., Springer (2009) 148–169
28. Parreiras, F.S., Walter, T.: Report on the combined metamodel. Deliverable ICT216691/UoKL/WP1-D1.1/D/PU/a1, University of Koblenz-Landau (2008) MOST Project.
29. Iqbal, A., Ureche, O., Hausenblas, M., Tummarello, G.: Ld2sd: Linked data driven software development. In: *Proceedings of the 21st International Conference on Software Engineering & Knowledge Engineering (SEKE'2009)*, Boston, Massachusetts, USA, July 1-3, 2009, Knowledge Systems Institute Graduate School (2009) 240–245
30. OMG: *Ontology Definition Metamodel*. Object Modeling Group. (September 2008)
31. O'Connor, M.J., Shankar, R., Tu, S.W., Nyulas, C., Parrish, D., Musen, M.A., Das, A.K.: Using semantic web technologies for knowledge-driven querying of biomedical data. In: *AIME*. (2007) 267–276
32. Staab, S., Scherp, A., Arndt, R., Troncy, R., Gregorzek, M., Saathoff, C., Schenk, S., Hardman, L.: Semantic multimedia. In: *Reasoning Web, 4th International Summer School, Venice, Italy*. Volume 5224 of LNCS., Springer (2008) 125–170
33. Staab, S., Franz, T., Görlitz, O., Saathoff, C., Schenk, S., Sizov, S.: Lifecycle Knowledge Management: Getting the Semantics Across in X-Media. In: *Foundations of Intelligent Systems, ISMIS 2006, Bari, Italy, September 2006*. Volume 4203 of LNCS., Springer (2006) 1–10

34. Silva Parreiras, F., Staab, S.: Using ontologies with uml class-based modeling: The twouse approach. *Data Knowl. Eng.* To be published.
35. Walter, T., Silva Parreiras, F., Staab, S.: *OntoDSL: An Ontology-Based Framework for Domain-Specific Languages*. In: *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009*. Volume 5795., Springer (2009) 408–422
36. Walter, T., Silva Parreiras, F., Staab, S., Ebert, J.: Joint language and domain engineering. In: *Proc. of 6th European Conference on Modelling Foundations and Applications , ECMFA 2010, Paris*. Volume 6138 of LNCS., Springer (2010)
37. Jouault, F., Bezivin, J.: *KM3: a DSL for Metamodel Specification*. *Lecture Notes in Computer Science* **4037** (2006) 171–185
38. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional (1995)
39. Silva Parreiras, F., Staab, S., Schenk, S., Winter, A.: Model driven specification of ontology translations. In: *Conceptual Modeling - ER 2008*. LNCS, Springer (2008)
40. Walter, T., Ebert, J.: Combining ontology-enriched domain-specific languages. In: *Proceedings of the of the Second Workshop on Transforming and Weaving Ontologies in Model Driven Engineering (TWOMDE) at MoDELS*. (2009)
41. Miksa, K., Kasztelnik, M., Sabina, P., Walter, T.: Towards semantic modelling of network physical devices. In: *Models in Software Engineering*. Volume 6002 of LNCS., Springer (2010) 329–343
42. Sirin, E., Parsia, B., Grau, B., 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
43. Horridge, M., Parsia, B., Sattler, U.: Explaining Inconsistencies in OWL Ontologies. In: *Scalable Uncertainty Management Conference 2009*. Volume 5785 of LNAI., Springer (2009) 124–137
44. Groener, G., Staab, S.: Modeling and Query Pattern for Process Retrieval in OWL. In: *Proc. of 8th International Semantic Web Conference (ISWC)*. Volume 5823., LNCS (2009) 243–259
45. Ren, Y., Groener, G., Lemcke, J., Rahmani, T., Friesen, A., Zhao, Y., Pan, J.Z., Staab, S.: Validating Process Refinement with Ontologies. In: *International Workshop on Description Logics*. (2009)
46. Gangemi, A., Borgo, S., Catenacci, C., Lehmann, J.: Task Taxonomies for Knowledge Content D07. In: *Metokis Project public Deliverable*. (2004) 20–42
47. Grimm, S., Motik, B., Preist, C.: Variance in e-Business Service Discovery. In: *Proc. of the ISWC Workshop on Semantic Web Services*. (2004)
48. Fowler, M., Beck, K., Brant, J., Opdyke, W.: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley (1999)
49. Budinsky, F., Brodsky, S.A., Merks, E.: *Eclipse Modeling Framework*. Pearson Education (2003)
50. Gasevic, D., Djuric, D., Devedzic, V.: *Model Driven Architecture and Ontology Development*. Springer (2006)
51. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A challenging model transformation. *Lecture Notes in Computer Science* **4735** (2007) 436
52. Jackson, D.: *Software Abstractions: logic, language, and analysis*. The MIT Press (2006)
53. 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
54. Sure, Y., Angele, J., Staab, S.: *OntoEdit: Guiding ontology development by methodology and inferencing*. *Lecture notes in computer science* 1205–1222

55. Gerber, A., Lawley, M., Raymond, K., Steel, J., Wood, A.: Transformation: The missing link of MDA. *Lecture Notes in Computer Science* (2002) 90–105
56. Martin, D.: OWL-S: Semantic Markup for Web Services. <http://www.w3.org/Submission/OWL-S> (2004)
57. Grüninger, M., Menzel, C.: The Process Specification Language (PSL) Theory and Application. *AI Magazine* **24** (2003) 63–74
58. Grüninger, M.: 29. In: *Ontology of the Process Specification Language*. Springer (2009) 575–592
59. Koschmider, A., Oberweis, A.: *Ontology Based Business Process Description*. In: *EMOI-INTEROP*. (2005)
60. Goderis, A., Sattler, U., Goble, C.: Applying DLs to workflow reuse and repurposing. In: *Description Logic Workshop*. (2005)
61. Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In: *Proc. of MoDELS 2006*. Volume 4199 of LNCS. (2006) 528–542
62. Roser, S., Bauer, B.: Ontology-based model transformation. In: *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers*. Volume 3844 of LNCS. (2005) 355–356
63. Bauer, B., Roser, S.: Semantic-enabled software engineering and development. In Hochberger, C., Liskowsky, R., eds.: *GI (2)*. Volume 94 of LNI. (2006) 293–296
64. Brockmans, S., Haase, P., Stuckenschmidt, H.: Formalism-Independent Specification of Ontology Mappings - A Metamodeling Approach. In: *OTM 2006 Conferences*. Volume 4275 of LNCS. (2006) 901–908
65. Roser, S., Bauer, B.: An approach to automatically generated model transformations using ontology engineering space. In: *Proc. of Workshop on Semantic Web Enabled Software Engineering (SWESE)*. (2006)
66. Goknil, A., Topaloglu, Y.: Ontological Perspective in metamodeling for Model Transformations. In: *Proc. of 2005 Symposia on Metainformatics*. Volume 214., ACM (2005)