International Journal of Computer Applications (0975 - 8887)
Volume 171 - No.3, August 2017

Validation of UML Artifacts in Model Driven Engineering using Description Logics based Ontology Reasoners

Ali Hanzala Khan
National University of Sciences and Technology
NUST-PNEC Karsaz
Karachi Pakistan

Naeem Abbas
National University of Sciences and Technology
NUST-PNEC Karsaz
Karachi Pakistan

ABSTRACT

This article presents an automatic approach to validate UML artifacts created during Model Driven Engineering. This validation approach may be used at both model and metamodel layer of Model Driven Architecture. This approach first automatically translates the UML artifacts into logical equivalent OWL 2 axioms and then use OWL 2 reasoners to validate the translations. Furthermore, the viability of the approach is demonstrated by validating 303 models and metamodels available in an online repository and the results show that half of the models and metamodels found erroneous.

General Terms

UML, Automatic Validation

Keywords

Reasoning, Metamodels, Models, MDE, Ontology

1. INTRODUCTION

Model Driven Engineering (MDE) [1] advocates the use of models to represent the most relevant design decisions in a software development project. Each software model is described using a particular modeling language, such as the Unified Modeling Language (UML) [2] or a domain-specific language. The definition of a modeling language is given using a so-called meta modeling language or a language to define modeling language.

Creating a new metamodel is not a simple task since it requires a good knowledge of the problem domain and how to use modeling languages to improve the development of new systems. Also, a metamodel can contain errors. A metamodel contains constraints on how concepts in a model can be related to each other, such as multiplicity, domain and range, composition and subset constraints. These constraints may lead to contradictions.

Similarly, creating models can be a complicated task. Models are intended to conform to the metamodels after which they are specified, but is is not always easy to ensure that they conform to all of the constraints that the metamodel imposes. Without tool-aided validation, the models can easily end up containing both obvious and less trivial errors with regard to their metamodel specification. Furthermore, it will be a help in the development process to have software tools that can aid in on one hand validating the internal consistency of metamodels and on the other hand validating models against their metamodels. The idea presented here is to represent metamodels as ontologies based on Description Logics (DL) [3]. Later, the existing DL reasoners can be used to validate the metamodels.

The advantage of this approach is that once encoding of the metamodels is obtained it can be used by the reasoners to solve both the validation problems. The reasoners' capabilities to reason about individuals belonging to a system of classes enable us to check models against their metamodel specification. The reasoners' capabilities to draw conclusions about the system of classes itself can be used to check the consistency of metamodels.

2. METHODOLOGY

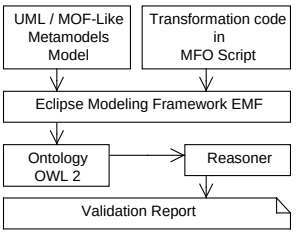

Fig. 1. Conceptual Model for Automatic Validation of Metamodel and Models

The study of metamodeling languages has led to a number of practical tools such as model repositories, diagram editors, model transformation tools and code generation tools that simplify enormously the creation of new practical tool chains that use UML or domain-specific modeling languages in software development projects. There exist metamodel repositories such as the Atlantic Metamodeling Zoo [4] that contain hundreds of metamodels for many different problem domains.

Two well-known metamodeling languages are MOF or the UML Infrastructure [5] that are used to define the UML as known by its practitioners. They can also be used to define new modeling languages. Other popular metamodeling languages include the Ecore, used in the Eclipse Modeling Framework (EMF) [6, 7] or KM3 [8].

These metalanguages are targeted to define domain-specific languages that are specific to a certain application domain.
There are several important differences between these metamodeling languages as described by Alanen et al. [9]. However, it can be acknowledged that share a common set of fundamental concepts: the classification of model elements into classes, the association of classes using properties and the specialization of classes and properties. In this article all such languages are called UML-like metamodeling languages, and it is these languages are targeted.

2.1 Description Logic and UML

This article also shows the logical equivalent representations of the UML construct using an ontology language based on Description Logics. DL are a family of logic languages that are especially suitable to model knowledge in a domain in terms of concepts and roles. The main characteristic of DLs is their reasoning capabilities. DLs are less expressive than first-order logic but they are decidable in the majority of cases and there exist efficient reasoning engines that can tackle classification and satisfiability problems.
By creating a mapping between a metamodeling language and the DL, this work obtains important benefits:

—Firstly, providing a formal and unambiguous definition of the metamodeling concepts that is independent of a specific model repository. This ensures interoperability of metamodeling tools.

—Secondly, the ability to use existing reasoning tools to analyze and validate metamodels and detect problems.

—Thirdly, the same representation is used for both to encode the constraints that apply to metamodels and for the constraints that metamodels impose on models.

2.2 Related Work

The use of DL and ontology languages in the context of metamodeling has been proposed in the past. Van Der Straeten has studied the use of DL to formalize fragments of UML and detect inconsistency between models [10]. Parreiras et al. has discussed the benefits of integrating metamodeling and ontology languages [11] and have proposed the OntoDSL language to define new domain specific languages [12]. Gašević et al has discussed the use of UML diagrams to construct ontologies [13]. Wang et al has suggested a partial mapping of MOF to OWL for consistency checking [14]. OMG also proposes the ODM which defines a UML to OWL mapping of classes and associations [15]. It is nonetheless believed that a detailed discussion on how to capture the semantics of UML-like metamodeling languages in the context of DLs and the use of DLs reasoners to validate metamodels are missing in the literature.
This proposal is based on the OWL 2 Web ontology language [16] and the SWRL Semantic Web Rule Language [17], whose semantics are rooted in DL. These are two standard recommendations from the W3C to improve machine interoperability of web content (the semantic web). There exists tools such as Pellet [18] that provide reasoning services for OWL 2 ontologies, and support DL-safe SWRL rules [19].
In order to preserve readability, this article writes OWL 2 using the functional syntax, while SWRL rules will be described using the human-readable syntax used by Horrocks et al. in their SWRL proposal [17].

3. REPRESENTING UML-LIKE METAMODELS IN OWL 2

This section provides a mapping of the basic UML and MOF metamodeling concepts of classes, various relationships among those conceptual classes into OWL 2.

3.1 Classes and Modeling Elements

The most fundamental concept in a UML and MOF metamodel is class. A class is the basic building block of a metamodels. A UML class in a metamodel represents an abstraction of the elements appearing in a model. Accordingly, each element in a model is an instance of a class in a metamodel. A class can be a specialization of another class. Each instance of the specialized class is implicitly an instance of its generalization and shares its features. A class can also represents an abstraction of entities that can appear in a model such as a state or a transition in a Statechart.
For each metaclass C in a UML metamodel, it is asserted that there exists a homonymous class in OWL 2 with the axiom Declaration(Class(C)). As an example, the mapping of UML Classes shown in Figure 2 into OWL 2 is as follows:

```
Declaration( Class(State) ) Declaration( Class(Transition) )
Declaration( Class(CompositeState) )
```

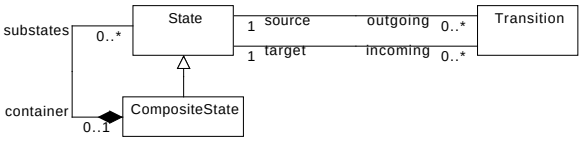

Fig. 2. Class Definition and Class specialization

A UML representation of specialization is also shown in Figure 2. In OWL 2 the SubClassOf axiom is used to represent class specialization. For each pair of metaclasses C1 and C2 where C2 is a subclass of C1, the axiom SubClassOf(C2 C1) is declare in the corresponding OWL 2 ontology. In this example class CompositeState as a subclass of State, which is consequently expressed as:
SubClassOf(CompositeState State)

3.2 Class Membership

Each element in a model, such as a transition in a given statechart, is an instance of a class in a metamodel. The UML and MOF semantics for instantiation follow closely the object-oriented paradigm. In this context, when declaring a model element m is of type C, it is asserted that:

—m is a direct instance of the class C
—m is an (indirect) instance of all the superclasses of C
—m is not an instance of any other class

However, declaring that an element m is of type C in OWL 2 asserts that:

—m is at least an indirect instance of the class C
—m is an (indirect) instance of all the superclasses of C

—m is not an instance of any classes disjoint to C

According to the UML semantics of class membership, when it is asserted that a model element is an instance of a class, it is also asserted that it is not an instance of its subclasses. This is not true in OWL 2. It is therefore needed to declare that there is a subset of model elements belonging to a class that do not belong to its subclasses. This set is called a direct instances of a class.
For each metaclass C a OWL 2 class C_DInstance is defined that is a subclass of C: SubClassOf(C_DInstance C). In the example shown in Figure 2,

SubClassOf(State˙DInstance State) SubClassOf(CompositeState˙DInstance CompositeState)
SubClassOf(Transition˙DInstance Transition)

Any model element m asserted to be of type C in a model is asserted to be of type C_DInstance in the transformation, in order to preserve the UML semantics of class membership.
Furthermore, a class equivalent to the union of its direct instances and any direct subclasses is explicitly declared. The example yields:

EquivalentClasses(State ObjectUnionOf(State˙DInstance CompositeState))
EquivalentClasses(CompositeState˙DInstance CompositeState)
EquivalentClasses(Transition˙DInstance Transition)

Moreover, the OWL 2 class representing the direct instances of C (C_DInstance) must be disjoint with the OWL 2 classes that represent direct subclasses of C. In this example, DisjointClasses(State_DInstance CompositeState)
In OWL 2, unless classes are explicitly stated to be disjoint or individuals to be unique instances, they may be not treated as such. This would allow us to assert that an individual belongs to class State and Transition simultaneously. However, the UML interpretation does not allow this. It is therefore to necessary to declare a class disjoint to any other classes that it is not a a superclass to, a subclass of, or sharing subclasses with.
In practice, in order to avoid parsing complicated graphs, the direct instances of a class disjoint to any classes that are not transitively a superclass to the direct instance are declared. As it was previously declared a class equivalent to the union of its subclasses and direct instances, it is therefore possible for a reasoner to infer which classes are disjoint to each other.
A pair of classes that have been declared disjoint to each other will naturally also be disjoint to any subclasses. Therefore, the only classes that need to be declared disjoint to the direct instance class are top level classes and any direct subclasses of a (transitive) superclass.
For the example shown in figure 3 following axioms are generated accordingly:

DisjointClasses(A˙DInstance ObjectUnionOf(C D E))
DisjointClasses(C˙DInstance ObjectUnionOf(D E))
DisjointClasses(D˙DInstance ObjectUnionOf(C E))
DisjointClasses(E˙DInstance ObjectUnionOf(C D))
DisjointClasses(F˙DInstance C)

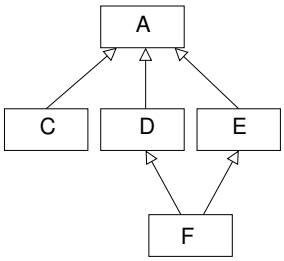

Fig. 3. An example of a metamodel with multiple inheritance.

3.3 Properties

Another fundamental concept in UML and MOF metamodels besides the class is the property. A property represents a basic relationship between classes. For example, the class representing a statechart transition may have two properties to represent the source and target state of each transition.
This section discusses how to represent UML set properties in OWL 2. In this section the properties related to an unordered set of objects are referred as set properties. These properties are used when the order of definition of elements is not relevant. For example the definition of classes in a package. The next section will discuss about the ordered set properties and bag properties. In addition, it will also discuss properties representing a composition.

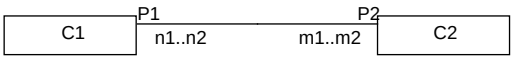

Fig. 4. A UML binary association with two properties

In a UML class diagram representing a UML metamodel, properties are presented as associations between classes. A binary UML association contains two properties that represent each end of the association. In the example shown in Figure 4 the properties are P1 and P2.
Each UML property is represented as an OWL ObjectProperty and prefixed the name of the property with the full name of the class in order to distinguish between properties with the same name belonging to different classes.
In this example, the UML properties P1 and P2 are declared as the OWL ObjectProperty C1_P2 and C2_P1 respectively:

Declaration(ObjectProperty(C1˙P2)) Declaration(ObjectProperty(C2˙P1))

Additional features of the properties such as type and multiplicity constraints are represented as domain, range and cardinality axioms in OWL 2.

ObjectPropertyDomain(C1˙P2 C1) ObjectPropertyRange(C1˙P2 C2)
ObjectPropertyDomain(C2˙P1 C2) ObjectPropertyRange(C2˙P1 C1)

The multiplicity of a UML property is mapped into OWL 2 by defining the owner class of the property as a subclass of a set of classes which own the same property of the same cardinalities.

SubClassOf(C1 ObjectMinCardinality(m1 C1˙P2))
SubClassOf(C1 ObjectMaxCardinality(m2 C1˙P2))
SubClassOf(C2 ObjectMinCardinality(n1 C2˙P1))
SubClassOf(C2 ObjectMaxCardinality(n2 C2˙P1))

In this example, both P1 and P2 are two ends of the same association and are opposite properties. In UML two opposite properties form a bidirectional and navigable association. It is represented in OWL 2 by stating that P1 is the inverse of P2 using the following axiom: InverseObjectProperty(C1_P2 C2_P1)

3.4 Composition

Composition is an important concept used to denote hierarchy and ownership in a model; it lets us organize a model as a collection of smaller parts. Composition has two requirements: it should not be possible to create a cyclic composition, and an element can only be referred to by one composition slot. This section defines a mapping of UML composition into OWL 2.
In OWL 2 there is no axiom defining composition. So in order to represent the composition constraints in OWL 2, two properties are defined: contains — denoting the aggregation of elements in the model, and owns — denoting the direct owner of an element. Figure 5 shows the basic metamodel illustrating these properties, and a model conforming to the metamodel. Composition can be used to create a hierarchy of any kind of objects, meaning the domain and range of these properties will be the default owl:Thing—the superclass of all individuals in an ontology.

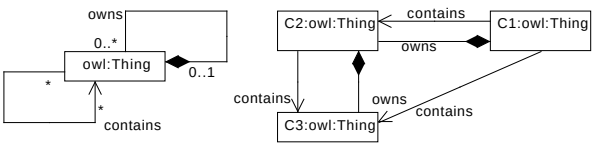

Fig. 5. Left: the composition metamodel. Right: a model conforming to the metamodel

The contains property is used to capture the acyclic requirement, by defining it as an irreflexive object property:
IrreflexiveObjectProperty(contains)
Furthermore, aggregation is transitive. However in order to guarantee decidability, it is not possible to combine the IrreflexiveObjectProperty and TransitiveObjectProperty axioms. The transitivity is expressed using the following SWRL rule:
$contains(?x, ?y) \wedge contains(?y, ?z) \implies contains(?x, ?z)$
Given that contains tracks the aggregation of model elements, each object that owns another object must also contain it. It is expressed by defining owns as a subproperty of contains. SubObjectPropertyOf(owns contains). The subproperty axiom is further discussed in Section 3.5.
The composition slot requirement states that an object can only have one owner. In other words, the inverse property of owns has a maximum cardinality of one. This restriction is expressed by using the inverse functional property axiom: InverseFunctionalObjectProperty(owns)
The owns and contains properties are defined as global properties; they apply to all classes in a metamodel. However, this work also wishes to be able to apply range and cardinality restrictions on composite relationships. If a metamodel defines a class CompositeState as being in composite relationship with a class State it should only be in a composite relationship with that class. However, these restrictions cannot be applied directly on the owns property while still using it to enforce composition restrictions over the entire metamodel. This problem is solved by making a separate subproperty of owns for each composite relationship in the metamodel, on which the restrictions can be applied like as any other property.
If the class CompositeState is in a composite relationship labeled substates with the State, it is declared as the following:

SubObjectPropertyOf(substates owns)
ObjectPropertyDomain(substates CompositeState)
ObjectPropertyRange(substates State)

As with any other property, cardinality is imposed on individual classes in the metamodel by subclassing restrictions like ObjectMinCardinality(n substates State). Furthermore, in order to be able to impose cardinality restrictions on the owning end of a composite association, an inverse object property to the properties representing a specific association in defined. It is translated as:

InverseObjectProperties(substates substates˙owned)

3.5 Property Subsetting

This section provides a mapping of UML property subsetting into OWL 2. Property subsetting was introduced in the MOF 2.0 and the UML 2.0 infrastructure, and allows for the specialization of an existing property, with new characteristics and a new basic type while retaining its existing features. Each instance of the specialized property is also an instance of the original property, and therefore elements that are a part of its slot should be a part of the original property slot. A UML example of property subsetting is shown in Figure 6.

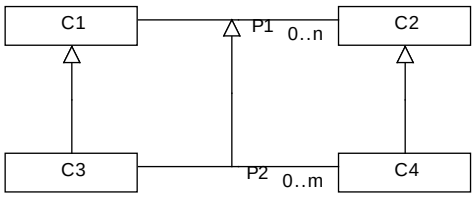

Fig. 6. UML association subsetting P2 ⊆ P1

The OWL 2 axiom SubObjectPropertyOf is used to subset a property: SubObjectPropertyOf(P2 P1). Furthermore, it modifies the specialized property, within the constraints imposed by the original property.
It is translated into OWL 2 as:

DisjointClasses(C1 C2) SubClassOf(C3 C1) SubClassOf(C4 C2)
Declaration(ObjectProperty(C1˙P1))
ObjectPropertyDomain(C1˙P1 C1) ObjectPropertyRange(C1˙P1 C2)
SubClassOf(C1 ObjectMaxCardinality(n C1˙P1))
SubObjectPropertyOf(C3˙P2 C1˙P1) ObjectPropertyDomain(C3˙P2 C3)
ObjectPropertyRange(C3˙P2 C4) SubClassOf(C3 ObjectMaxCardinality(m C3˙P2))

4. REPRESENTING UML-LIKE MODELS IN OWL 2

A Model consists of classes and associations, if a model is designed for some specific domain then that model must be an instance of metamodel which defines the domain language. Since a model only consists of class instances and association between those instances, ensuring that a model is part of the domain defined by a metamodel is primarily a case of ensuring that the reasoner has enough information to enforce cardinality constraints and class membership, and that no associations or individuals other than what is explicitly provided by the model and metamodel is inferred. In other words, providing the reasoner with enough information to evaluate the model according to a closed world assumption.

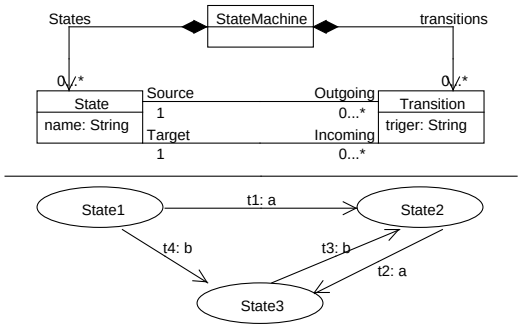

Fig. 7. Top: Metamodel of State Machine, Bottom: An State Machine Model.

4.1 Modeling Classes as Instances

While modeling for DSL, a class of a model is depicted as an instance of conceptual class of Metamodel which is representing the DSL. An instance of class in OWL 2 is represented by an axiom ClassAssertion(ClassName InstanceName), Figure 7 depicting the state machine metamodel and its model, the translation of the classes of state machine model in OWL 2, as follows:

ClassAssertion(State State1) ClassAssertion(State State2)
ClassAssertion(State State3) ClassAssertion(Transition t1:a)
ClassAssertion(Transition t2:a) ClassAssertion(Transition t3:b)
ClassAssertion(Transition t4:b)

4.2 Modeling Associations as Properties

For representing the association between concepts of Metamodel the OWL 2 property axiom is used, similarly for representing the the associations between concepts of model the OWL 2 property is instantiated by using axiom ObjectPropertyAssertion(PropertyName RangeClass DomainClass), the association among the classes of model represented in Figure 7 is translated by using OWL 2 axioms is as follows.

ObjectPropertyAssertion(Source State1 t1:a) ObjectPropertyAssertion(Source State2 t2:a)
ObjectPropertyAssertion(Source State3 t3:b) ObjectPropertyAssertion(Source State4 t4:b)

ObjectPropertyAssertion(Outgoing t1:a State1) ObjectPropertyAssertion(Outgoing t2:a State2)
ObjectPropertyAssertion(Outgoing t3:b State3) ObjectPropertyAssertion(Outgoing t4:b State4)

ObjectPropertyAssertion(Target State2 t1:a) ObjectPropertyAssertion(Target State3 t2:a)
ObjectPropertyAssertion(Target State2 t3:b) ObjectPropertyAssertion(Target State3 t4:b)

ObjectPropertyAssertion(Incoming t1:a State2) ObjectPropertyAssertion(Incoming t2:a State3)
ObjectPropertyAssertion(Incoming t3:a State4) ObjectPropertyAssertion(Incoming t4:a State3)

4.3 Applying Closed World Restrictions

UML and MOF-like modeling adopts the closed world assumption whereas Ontology Development Modeling adopts the open world assumption, due to this when ever the closed world UML or MOF-like models or metamodels are translated in ontology, for the sake of correctness of reasoner error report and validation outcome there is a need to create the closed world environment inside the open world of Ontology. For creating closed world environment while translating the model in OWL 2, first the equivalent instance is required for each class with the respective classifying class by using OWL 2 axioms,

EquivalentClasses(ClassifierClassName˙Metamodel ObjectOneOf(InstantiatedClassName˙Model))

and secondly all classifying classes must be made equivalent with owl:Thing by using following OWL 2 axiom,

EquivalentClasses(owl:Thing ObjectUnionOf(InstantiatedClass1 InstantiatedClass2 InstantiatedClassN))

By applying above rules on model of Figure 7, the OWL 2 closed world translation is as follows,

EquivalentClasses(State ObjectOneOf(State1))
EquivalentClasses(State ObjectOneOf(State2))
EquivalentClasses(State ObjectOneOf(State3))
EquivalentClasses(Transition ObjectOneOf(t1:a))

```
EquivalentClasses(Transition ObjectOneOf(t2:a))
EquivalentClasses(Transition ObjectOneOf(t3:b))
EquivalentClasses(Transition ObjectOneOf(t4:b))
EquivalentClasses(owl:Thing ObjectUnionOf(State Transition))
```

5. AUTOMATIC TRANSLATION TO ONTOLOGIES

The transformation from a metamodel expressed using UML to OWL 2 is implemented using the Model-to-Text transformation tool MOFScript [20] [21].
MOFScript consists of two parts: the MOFScript tool and the MOFScript language. The MOFScript tool is developed as an Eclipse [6, 7] plugin and contain an implementation of the MOFScript language and it provides ways of editing, compiling and executing MOFScript transformation code. MOFScript transformations are MOFScript language programs that define a set of rules that can translate metamodel elements and relations between them to expected output through print statements. The MOFScript transformation code is written based on one or more input metamodels, then compiled and executed on one or more loaded input files which contains models conforming to the input metamodels.
The input metamodel in this implementation is UML2 2.1.0, the input file is a .uml file which contains a UML metamodel in XML syntax and output file contain an OWL 2 ontology written in OWL 2 functional syntax.

6. CORRECTNESS AND EXPERIMENTAL RESULTS

To determine the correctness of presented approach in practice the test have been conducted by translating metamodels from Atlantic Zoo in OWL 2 Ontology and then validate it by using OWL 2 reasoners.

6.1 Determining Correctness of the Translation

To determine whether the translation is correct it would be helpful to have an unambiguous description of the semantics of the metamodels that are required to translate. This is not possible for two reasons. First, the semantics of UML are not unambiguously defined. Secondly, even if UML had clear semantics this article is not really focused in UML itself, rather it focuses in aspects of a part of UML that it shares with other metamodel formalisms, such as generalization and composition.
What does it mean to be "correct"? A first requirement seems to be that if one translate metamodels it deems to be correct, the resulting ontologies should be consistent and satisfiable. But this requirement is not enough. After all, a translation that takes every metamodel into an empty ontology will never generate an inconsistent ontology, vacuously satisfying the requirement. To demonstrate that this is not an empty concern the work of Wang ea. [14] is referred. Their translation is quite literal, mapping elements in UML to elements in OWL without regard for the differences in semantics. Some constructs that have no counterparts in OWL are translated in the form of comments, and some constructs are simply ignored. The result is a translation that admittedly does transform valid metamodels to consistent otologies, but is also rather useless.
To ensure that the notion of correctness of this article is useful it needs to be consider that which metamodels are translated as inconsistent ontologies. It can be done by placing a number of constraints on the metamodels. [22] Metamodels violating the constraints should result in ontologies that are inconsistent or unsatisfiable. The notion of correctness thus becomes a question of discriminating between correct and incorrect metamodels.
To make sure that the translation is correct in the sense that it can discriminate between the relevant classes of metamodels, a number of pairs of metamodels are constructed. The metamodels in each pair make use of the same features but one is violating a certain constraint and the other is not. It can thus be sure that the translation can make the relevant distinction. An example of such a pair is given in figure 8. The list of constraints that can be validated using presented approach is as follows:

—Multiplicity constraints of the form $n..m$ where $n > m$

—Multiple ownerships with minimum cardinality constraints of a class.

—Ownership with a minimum cardinality of a class whose superclass is already owned with a minimum cardinality constraint.

—Classes being subclasses of themselves.

—Subset properties that violate the domain or range of their super-properties.

—Classes being in a composition relationship with themselves with multiplicity strictly greater than zero.

—Composition relationships forming a cycle between different classes where all the multiplicities are strictly greater than zero.

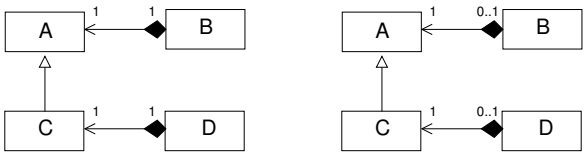

Fig. 8. Example of a contrasting metamodel pair. The metamodel on the left is violating the constraint that instances cannot have more than one owner. The metamodel on the right is valid.

6.2 Validating the Atlantic Metamodel Zoo

The Atlantic metamodel zoo is a library of 303 metamodels, maintained by the AtlanMod team. The AtlanMod team does not state the criteria for inclusion of a metamodel in the zoo, but most of the metamodels seem to have been entered by modeling and computer science researchers. The metamodels are available in different languages, including UML 2, OWL, KM3 etc.
All examples in the UML 2 zoo are translated into OWL 2 using the implemented translator and then used the OWL 2 reasoner Pellet to validate the generated ontologies. Pellet is an open source Java-based ontology reasoner developed by a Clark& Parsia LLC, which is an R&D firm, specializing in Semantic Web and advanced systems[23]. Given a source metamodel, the process consists of three steps:

(1) Run the source metamodel through the translator.

(2) Check the satisfiability of the generated ontology. If the ontology is inconsistent it will also be reported in this step.

(3) If the reasoner reported any problem with the ontology, then ask reasoner to generate an explanation in form of a list of violated axioms.

The time taken to translate the metamodels varies considerably. 93% of the metamodels took under 10 seconds but large metamodels can take minutes, with the largest taking 13.5 minutes to translate on a Pentium 4 PC. The time taken for the reasoner to detect constraint violations (step two above) is between 3 and 7 seconds for each metamodel in the zoo.
When the reasoner reports a problem it is necessary to manually inspect the generated explanation and usually also a diagram of the metamodel in question since it is in many cases not obvious from the explanation what the cause of the problem is.
Out of the 303 metamodels, 43% were found to have problems. A list of the ill-formed metamodels and the classes causing problems is reported in the appendixes of PhD Thesis [24]. The metamodels that were not found to have problems are not guaranteed to be well formed, they are just free from violations that presented method can detect. However, it is still consider that this shows that the problem of metamodel validation is indeed an issue of practical concern.

7. CONCLUSION

This article proposed an approach that validates UML and MOF-Like metamodels and its corresponding models by mapping from UML-like metamodeling and modeling concepts to OWL 2. The purpose of this mapping is to describe the semantics of metamodels using a language with a formal semantic definition and to allow the automatic validation of metamodels. Many metamodels found in the largest public repository contain errors and this approach can find them quickly and effortlessly
Several researchers including the author of this article [25] have proposed formalizations of UML, MOF and Ecore metamodeling languages, including Akehurst et al. [26], Alanen and Porres [22], Clark et al. [27], Jouault and Bézivin [8], Schätz [28] and Varró [29]. As discussed in the introduction, other researchers have discussed the significant theoretical aspects that may be seen as a foundation of this work. Whereas the work presented in this article extends the previous work [25] which is implemented and tested to demonstrate the viability.
Another approach to validating UML and MOF metamodels is through OCL [30] rules. A metamodel can define a well-formed rule that prevents an association's minimum cardinality from being greater than its maximum cardinality. However, OCL operates on the syntactic level. Consequently, validating all combinations of UML elements requires a very extensive ruleset. The advantage of using a reasoner to detect contradictions is that one does not need to express the well-formed rules explicitly.
One issue when mapping UML-like languages into OWL is that these languages use opposite assumptions. UML-like languages operate on a closed-world assumption, where complete knowledge is assumed to be provided in a model or metamodel, which means everything not provably true is considered false. Conversely, OWL uses the open world assumption, in which an ontology is not considered to provide complete knowledge. As it can be seen in for instance Section 3.2, this requires some non-obvious additional declarations in the mapping. It also means that if one wishes to validate models against the translated meta-models, it will need similar additional declarations.

Nonetheless, it is consider that the use of languages and tools envisioned for the semantic web as a foundation for software modeling languages and tools is a promising proposition. The existing issues of interoperability of metamodels and models that face current tools could be addressed by reusing results from the semantic web community. However, as can be seen in this article, the mapping between languages is not trivial if it intends to preserve the original semantics.

7.1 Future Work

In future, this approach may be evolved to analyze the consistency of full system model comprising of numerous models of more than one type.

7.2 Acknowledgement

The authors would like to thank Sören Höglund and Espen Suenson from Åbo Akademi University for their valuable help in the development of the initial version of transformation tool that increased the viability of this work.

8. REFERENCES

[1] Stuart Kent. Model Driven Engineering. In *Proc. of IFM International Formal Methods 2002*, volume 2335 of *LNCS*. Springer-Verlag, 2002.

[2] OMG. UML 2.2 Superstructure Specification, February 2009. available at `http://www.omg.org/`.

[3] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA, 2003.

[4] The Atlantic Zoo. Available at http://www.eclipse.org/gmt/am3/zoos/atlanticZoo/.

[5] OMG. UML 2.2 Infrastructure Specification, February 2009. available at `http://www.omg.org/`.

[6] The Eclipse Modeling Framework website. `http://www.eclipse.org/emf`.

[7] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, August 2003.

[8] Frédéric Jouault and Jean Bézivin. KM3: a DSL for Metamodel Specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, Bologna, Italy, 2006.

[9] Marcus Alanen, Torbjörn Lundkvist, and Ivan Porres. Comparison of modeling frameworks for software engineering. *Nordic J. of Computing*, 12(4):321–342, 2005.

[10] R. Van Der Straeten. *Inconsistency Management in Model-driven Engineering. An Approach using Description Logics*. PhD thesis, Vrije Universiteit Brussel, Brussels, Belgium, September 2005.

[11] Fernando Silva Parreiras, Steffen Staab, and Andreas Winter. On marrying ontological and metamodeling technical spaces. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 439–448, New York, NY, USA, 2007. ACM.

[12] Tobias Walter, Fernando Silva Parreiras, and Steffen Staab. OntoDSL: An ontology-based framework for domain-specific languages. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 408–422. Springer, 2009.

[13] Dragan Gašević, Dragan Djurić, and Vladan Devedžić. Mda-based automatic owl ontology development. *Int. J. Softw. Tools Technol. Transf.*, 9(2):103–117, 2007.

[14] Shengjun Wang, Longfei Jin, and Chengzhi Jin. Ontology definition metamodel based consistency checking of uml models. pages 1 –5, may 2006.

[15] International Business Machines, Object Management Group, and Sandpiper Software. Ontology definition metamodel (ODM). OMG Document ad/2003-02-23. Available at `http://www.omg.org/`.

[16] Conrad Bock, Achille Fokoue, Peter Haase, Rinke Hoekstra, Ian Horrocks, Alan Ruttenberg, Uli Sattler, and Michael Smith. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax. W3 Recommendation Available at `http://www.w3.org/TR/2009/REC-owl2-syntax-20091027/`.

[17] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A semantic web rule language combining OWL and RuleML. Availible at `http://www.w3.org/Submission/SWRL/`, 2004.

[18] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *Web Semant.*, 5(2):51–53, 2007.

[19] Vladimir Kolovski, Bijan Parsia, and Evren Sirin. Extending the shoiq(d) tableaux with dl-safe rules: First results. In Bijan Parsia, Ulrike Sattler, and David Toman, editors, *Description Logics*, volume 189 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[20] MOFScript Homepage. Available at `http://www.eclipse.org/gmt/mofscript/`.

[21] MOFScript User Guide, 2009. Document Available at `http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide-0.8.pdf`.

[22] Marcus Alanen and Ivan Porres. A Metamodeling Language Supporting Subset and Union Properties. *Springer International Journal on Software and Systems Modeling*, 7(1):103–124, 2007. Available at `http://www.springerlink.com/content/8k67436222447147/`.

[23] Clark and Parsia. *Pellet: OWL 2 Reasoner for Java*. Homepage, available at `http://clarkparsia.com/pellet`.

[24] Ali Hanzala Khan. *Consistency of UML Based Designs using Ontology Reasoners*. PhD thesis, Abo Akademi University, 2013.

[25] Ali Hanzala Khan and Ivan Porres. Consistency of uml class, object and statechart diagrams using ontology reasoners. *J. Vis. Lang. Comput.*, 26(C):42–65, February 2015.

[26] David H. Akehurst and Stuart Kent and Octavian Patrascoiu. A relational approach to defining and implementing transformations between metamodels. *Software and System Modeling*, 2(4):215–239, 2003.

[27] Tony Clark, Andy Evans, and Stuart Kent. The Metamodelling Language Calculus: Foundation Semantics for UML. In *Proceedings of the Fundamental Aspects of Software Engineering (FASE)*, pages 17–31, 2001.

[28] Bernhard Schätz. Formalization and rule-based transformation of emf ecore-based models. pages 227–244, 2009.

[29] Dániel Varró and András Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, 2(3):187–210, October 2003.

[30] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.