TwoUse: Integrating UML Models and OWL Ontologies

Fernando Silva Parreiras
Steffen Staab
Andreas Winter

Nr. 16/2007

Arbeitsberichte aus dem Fachbereich Informatik

Arbeitsberichte des Fachbereichs Informatik

ISSN (Print): 1864-0346
ISSN (Online): 1864-0850

Kontaktdaten der Verfasser

Fernando Silva Parreiras, Steffen Staab, Andreas Winter
Institut für Informatik
Fachbereich Informatik
Universität Koblenz-Landau
Universitätsstraße 1
D-56070 Koblenz
EMail: parreiras@uni-koblenz.de; staab@uni-koblenz.de; winter@uni-mainz.de;

TwoUse: Integrating UML Models and OWL Ontologies

Fernando Silva Parreiras[1] *, Steffen Staab[1], and Andreas Winter[2]

[1] Institute for Computer Science, University of Koblenz-Landau
Universitaetsstrasse 1, Koblenz 56070, Germany
{parreiras, staab}@uni-koblenz.de
[2] Institute for Computer Science, Johannes-Gutenberg-University Mainz
Staudingerweg 9, Mainz 55128, Germany
winter@uni-mainz.de

Abstract. UML models and OWL ontologies constitute modeling approaches with different strength and weaknesses that make them appropriate for use of specifying *different aspects* of software systems. In particular, OWL ontologies are well suited to specify classes using an expressive logical language with highly flexible, dynamic and polymorphic class membership, while UML diagrams are much more suitable for specifying not only static models including classes and associations, but also dynamic behavior. Though MOF based metamodels and UML profiles for OWL have been proposed in the past, an integrated use of both modeling approaches in a coherent framework has been lacking so far. We present such a framework, *TwoUse*, for developing integrated models, comprising the benefits of UML models and OWL ontologies.

1 Introduction

The Unified Modeling Language (UML) is a visual design notation [1] for building software systems accompanied by the Object Constraint Language (OCL) [2]. UML is a general-purpose modeling language, capable of capturing information about different views of systems, like static structure and dynamic behavior.

Ontologies provide shared domain conceptualizations representing knowledge by a vocabulary and, typically, logical definitions ([3], [4]). The Web Ontology Language (OWL) [5] provides a class definition language for ontologies. More specifically, OWL allows for the definition of classes by required and implied logical constraints on the properties of their members.

UML and OWL comprise some constituents which are similar in many respects, like: classes, associations, properties, packages, types, generalization and instances [6]. However, both approaches have their advantages and disadvantages. E. g. UML provides means to express dynamic behavior, whereas OWL does not. On the other hand, OWL is capable of inferring generalization and specialization between classes as well as class membership of objects based on

* Financially supported by CAPES Brazil.

the constraints imposed on the properties of class definitions, whereas UML class diagrams do not allow for dynamic specialization/generalization of classes and class memberships or any other kind of inference *per se*.

Contemporary software development should make use of the benefits of both approaches to overcome their restrictions. This requires either to bridge software models from both approaches or to integrate them into a composed model. This paper presents the *TwoUse-approach* (Transforming and Weaving Ontologies and UML in Software Engineering) on integrating UML/OCL-based modeling and OWL-based modeling.

This paper is organized as follows. Section 2 describes a simple use case for developing an e-commerce system using separately UML-based modeling and OWL-based modeling. Comparing the approaches leads to requirements to be fulfilled by an integrated development approach. Section 3 presents the *TwoUse approach*, by showing its composed software model including its MOF based metamodel, library and a simple UML Profile. A validation of the *TwoUse-approach* by evaluating the criteria for integrated UML/OWL modeling and contrasting it to related approaches is given in Section 4. Finally, Section 5 presents the conclusions and points towards future works.

2 TwoFold-Use Case

Combining OWL and UML requires analysis of what can be done today on each side and how to find the way to the other side. Here we present some bridges to be built based on an example of an e-commerce company established inside the European Union, partially taken from [7].

Let us assume in this scenario that a development team will develop an e-commerce application with the following characteristics:

- Purchase orders associate a customer with a list of products.
- Customers have a home country.
- Some countries form a free-trade zone, like the European Union.
- Orders from customers who live in a country inside the European Union are duty free.
- A tax of 60% will be charged to non duty free orders.

UML-based software development

A version of the e-commerce domain modeled using an UML class diagram is presented in Fig. 1.

UML class diagrams alone are usually not expressive enough to describe detailed behavior of operations. For example, an operation called `getCharges()` could be declared as member of the class `PurchaseOrder` to calculate the taxes applied to an order, verifying whether it is duty free. Therefore it is to expect that e, g. a textual language, such as OCL, will have to be used to fill some gaps. OCL can be used [2]: as a query language; to specify invariants on classes and

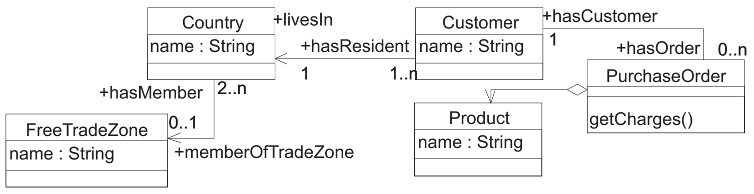

Fig. 1. UML Class Diagram of the e-commerce domain

types in the class model; to describe pre- and post conditions on operations and methods; and to specify derivation rules for attributes for any expression over a UML model. For instance, the operation `getCharges()` could be expressed using OCL like:

```
context PurchaseOrder::getCharges() : Real
body:
if self->select(hasCustomer.livesIn.memberOfTradeZone.name='EU')
           ->notEmpty()
then 0.00 else 0.60 endif
```

Nevertheless, this way of specifying the operation `getCharges()` has some shortcomings. The semantics of a duty free order is embedded in the operation specification and can be difficult to be found in bigger domains. Furthermore, if a class called `DutyFreeOrder` needs to be defined as a subclass of `PurchaseOrder` because of its particular behavior, the concept of a duty free order will be stated twice: as a class definition and as an expression of the operation `PurchaseOrder.getCharges()`.

OWL-based software development

To avoid such redundancy, and thus to improve maintainability, it would be preferable to define duty free order only once. To do so, one requires a logical class definition language that is more expressive than UML, e.g. the Web Ontology Language, OWL [5].

Describing logical classes, relations between them and characteristics of properties is possible using one of the OWL sublanguages: OWL Lite, OWL DL, and OWL Full. OWL DL supports users who want maximum expressiveness without losing computational completeness and decidability of reasoning systems. In this paper we refer to OWL DL as simply OWL.

When modeling part of the e-commerce domain using an UML profile for OWL, the ontology should look like the one depicted in Fig. 2.

OWL classes provide an abstraction mechanism for grouping resources with similar characteristics [5]. An OWL class can be described by six different ways: (1) a class identifier; (2) an exhaustive enumeration of individuals; (3) a property restriction; (4) the intersection of class descriptions; (5) the union of class descriptions; (6) the complement of a class description.

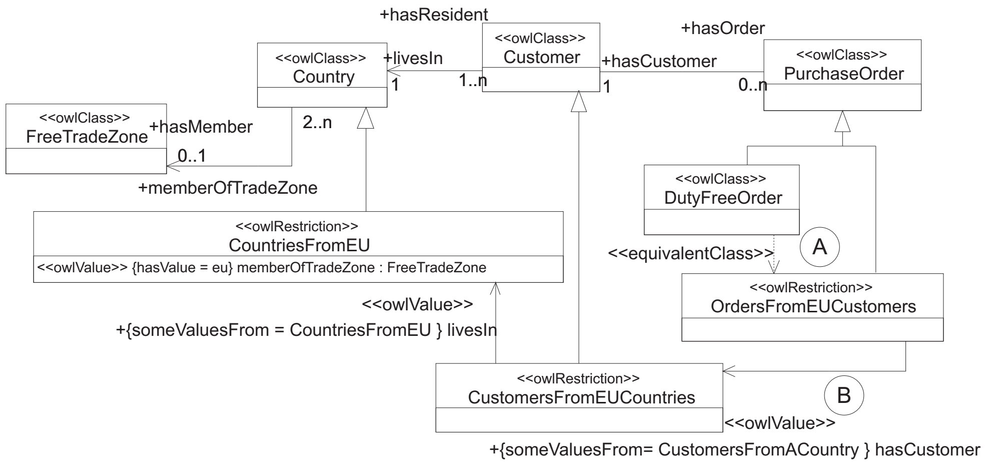

Fig. 2. Ontology of the e-commerce domain using an UML profile for OWL

A class of duty orders could be declared as the complement of the class of duty free orders(6). A class of economic resources could be declared as an union of product and payment method (5). The class `CountriesFromEU` (Fig. 2) could be declared as all countries that have in the property `memberOfTradeZone` the object value `eu` (1)(3).

OWL permits a subclass to be asserted or to be inferred from the definition of a class in terms of other classes. Referring to Fig. 2, the class `DutyFreeOrder` defines as equivalent (A) the subclass `OrdersFromEUCustomers` of the domain of the property `hasCustomer`. These individuals are precisely those for which the range of `hasCustomer` is in the class `CustomersFromEUCountries` (B). Given that we know an individual to be an instance of `DutyFreeOrder`, we can infer that it has the property `hasCustomer` which the value is an instance of `CustomersFromEUCountries`. Conversely, if we have an individual which has the property `hasCustomer` and the value of `hasCustomer` associated with that individual is a `CustomersFromEUCountries`, we can infer that the individual is an instance of `DutyFreeOrder`.

OWL ontologies can be operated by a reasoner which usually provide services like consistency checking (i), concept satisfiability (ii), concept classification (iii) and instance classification (iv). For instance, we could verify whether it is possible to have a duty free order at all (ii), or whether, according to the class descriptions, every order is a duty free one (iii). We can ask a reasoner whether an instance of the `Order` class is also an instance of `DutyFreeOrder` (iv). The shortcoming is that we cannot specify the reasoner calls using ontologies, i.e., behavior feature. We have to rely on object class operations that have in their body these calls.

The reader may note that the capability to model UML classes and OWL classes in a single, integrated diagram is required. However, the OWL classes designed with UML cannot be referred to in UML methods, e.g. they cannot be exploited through OCL expressions. The reason is that the two types of classes have different metamodels and different semantics. The usage of an OWL reasoning service in the 'pure' UML part of such a diagram is not yet feasible. For example, an instance of the `PurchaseOrder` class could be classified dynamically by an OWL reasoner to belong to `DutyFreeOrder`. This information should then be used by the application logic in order to compute overall charges for this order.

In Fig. 2 it becomes clear that the software developer should be free to adopt any UML tool that supports UML2 extension mechanisms. In order to leverage the skills of the developers in UML and to avoid having to use new and various tools, one should be able to specify OWL ontologies and UML classes in an integrated diagram.

The weakness of UML profiles is that they are more complex to be managed than metamodels in a model driven approach and do not provide strict separations in a metamodeling framework. The realization of an integrated application with reasoning logic support requires capabilities for model driven engineering in order to enforce model constraints, perform model checking and to transform the given models into platform specific models, such as intermediating models, the normative OWL exchange syntax RDF/XML, and, specifically, Java code.

Requirements for an integrated UML/OWL software development

After analyzing the scenario above, we have identified requirements that should be fulfilled in an integrated framework:

1. *Ontologies Design*. Enough expressiveness to model a complete ontology.
2. *UML Tools Compatibility*. The liberty of choosing from a range of UML tools that support UML2 extension mechanisms, in order to use an UML profile for ontology design.
3. *OCL Suppor*t. Ability to write constraints on objects and specify query operations.
4. *Code Generation Support*. The capability to have the code generated. A range of implementations provides APIs to work with ontologies at the code level.
5. *UML and Ontology Modeling Integrated*. Ability to have sufficient semantics to model an UML diagram and an OWL ontology at the same time.
6. *Model Driven Engineering Support*. A model driven approach, allowing constraint enforcement, model checking and supporting model to model transformations, independent of any profile.
7. *Reasoner-query Operations Specification*. Possibility to specify operations that query inference engines over ontologies to infer knowledge about classes and instances.

These requirements will be revisted in section 4.

3 The TwoUse Approach

In order to fulfill the requirements specified in the last section, we present the TwoUse approach. TwoUse is based on four core ideas:

1. It provides an integrated *MOF based metamodel* as a common backbone for UML (including OCL) and OWL modeling;
2. It uses an *UML profile* as its integrated syntactic basis, supporting UML2 extension mechanisms and mappings from the profile onto TwoUse models;
3. It provides a canonical *set of transformation rules* in order to deal with integration at the semantic level.
4. It extends the *basic library* provided by the OCL specification [2], which we call OCL-DL.

To give an intuition of the target integration, let us consider our running example. Instead of defining the operation `getCharges()` in the class `PurchaseOrder` using complex OCL constraints, a more transparent and maintainable solution will use the expressiveness of the OWL language. Querying an *OWL reasoning service*, an OCL-like query may just ask whether a given instance of `PurchaseOrder` fulfills all the logical requirements of the OWL subclass `DutyFreeOrder`. The body of the `getCharges()` operation will then be specified very simply by:

```
context PurchaseOrder::getCharges() : Real
body: if self.isInstanceOf(DutyFreeOrder)
      then 0.00 else 0.60 endif
```

The advantage of this integrated formulation of `getCharges()` lies in separating two sources of specification complexity. First, the classification of complex classes remains in an OWL model. The classification is easily re-useable for specifying other operations and it may be maintained using diagram visualizations as well as decidable, yet rigorous reasoning models. Second, the specification of the business logic itself remains in an OCL specification which becomes smaller and, hence, better understandable and easier to maintain.

3.1 Metamodel

To accomplish the composition of TwoUse, we have adopted the OWL metamodel specified by OMG ODM Ontology Definition Metamodel [6]. TwoUse imports two others: UML [1] and OCL [2]. The UML metamodel allow us to specify a class with its behavioral and structural features. The OCL metamodel makes it possible to specify expressions and pre-built operations to the classes.

The TwoUse metamodel comprises basically two metaclasses: The metaclass `TUClass` and the metaclass `TUPackage`. The metaclass `TUClass` specifies a TwoUse class. A TwoUse class is a class with all features of both `UML::Class` and `OWL::OWLClass`. The metaclass `TUPackage` defines a TwoUse package, which contains TwoUse classes. Thus, a TwoUse package contains features of both `UML::Package` and `OWL::Ontology`. Figure 3 depicts the metamodel.

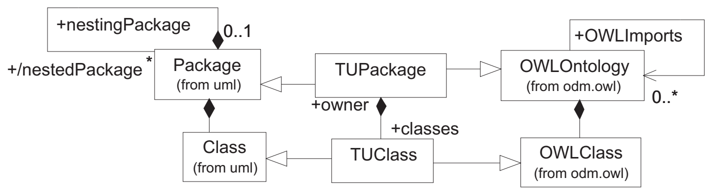

Fig. 3. TwoUse Metamodel (M2), conforming with OMG MOF(M3)

3.2 OCL-DL: An OCL Library Extension

To enable the modelers to have access to pre-built operations from all TwoUse classes, which will permit operations to call the OWL reasoner, we propose an extension to the OCL basic library and call it OCL-DL.

OCL prescribes a predefined type called `OclAny`, which acts as a supertype for all the types except for the OCL pre-defined `collection` types. Hence, features of `OclAny` are available on each object in all OCL expressions and all classes in a UML model inherit all operations defined on `OclAny`. We highlight two of such operations:

– `oclIsTypeOf(typespec: OclType): Boolean`. Evaluates to `true` if the given object is of the type identified by `typespec`.
– `oclIsKindOf(typespec: OclType): Boolean`. Evaluates to `true` if the object is of the type identified by `typespec` or a subtype of it.

Any of the operations above evaluates to false if the type identified by `typespec` is a subtype of the type of the given object. The reason is that we cannot determine the semantics of a subtype within object orientation. Thus, in the context of the `PurchaseOrder` class, `oclIsKindOf(DutyFreeOrder)` would return false, because `DutyFreeOrder` is a subclass of `PurchaseOrder` and it is impossible to determine if the given object meets the additonal constraints for `DutyFreeOrders`.

That is the basis for one of the improvements provided by OCL-DL. We propose new operations which rely on *reasoning engine services* to extend the boundaries of OCL towards OWL. By way of illustration, one could use the OCL-DL operation `owlIsInstanceOf(DutyFreeOrder)` which makes use of a reasoner and returns true if, the properties of the object satisfy the sufficient conditions to be a member of class `DutyFreeOrder`. We propose the following OCL-DL pre-built operations:

– `owlIsInstanceOf(typespec: OclType): Boolean`. Evaluates to `true` if the object satisfy all the logical requirements of the OWL class `typespec`.
– `owlAllTypes(): Set(OclType)`. Returns all types classified by a reasoner, whose the object satisfies the logical requirements.
– `owlAllInstances(): Set(T)`. Returns all instances that satisfy the logical requirements of the class of the given object. The type of T is equal to the type of the object.

In order to add the operations above, we extend the OCL library adding a new M1 instance of the metaclass `OclType` called `OwlAny`. While all classifiers except collections conform to `OclAny`, only the classifier TUClass conforms to `OwlAny`. Thus, `OwlAny` acts as a supertype for all TwoUse classes, which inherit all operations of `OwlAny`.

Figure 4 describes the OCL-DL in the context of four metamodels: OCL, UML, OWL and TwoUse. At level M2, white boxes represent metaclasses imported from UML metamodel and light grey boxes represent metaclasses from OCL metamodel. The back box represent the TwoUse metaclass imported from TwoUse metamodel, which specializes the UML class and the OWL class, the dark grey box. At level M1, we show the class `PurchaseOrder` which is a M1 instance of the metaclass `TUClass` and, because of that, is a subtype of the OCL-DL class `OwlAny`. `PurchaseOrder` is indirectly a M1 instance of the UML metaclass `Class` too and so is indirectly a subtype of the OCL Class `OclAny`. The class `Product` is only an UML Class and is also a subtype of `OclAny`. The class `Country` is a M1 instance of the metaclass `OWLClass` and does not have any operations.

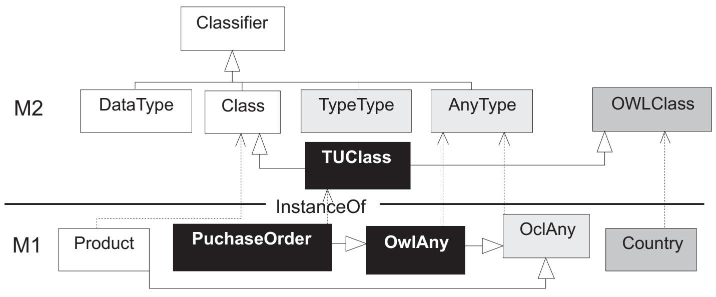

Fig. 4. OCL-DL Library Extension and sample model classes.

Since OWL classes do not support operations and cannot be referred in OCL expressions, we use a TwoUse Class to build the bridge. A TwoUse Class inherits from both OWL and UML Classes. Due to that, a M1 instance of the metaclass `TUClass` is an instance of the metaclass `OWLClass` too. This extension gives an extra power to OCL implementations adding query capabilities.

3.3 Profile

Based on the definitions of `TUClass` and `TUPackage`, we can reuse an existing UML profile for OWL. Here, we used the profile for OWL proposed by OMG to design the model. We call this UML class diagram with elements stereotyped by an UML Profile for OWL a *hybrid diagram*. The hybrid diagram comprises three different views, illustrated in Fig. 5: (1) the UML view with its OCL expressions, (2) the OWL view with its class definitions and (3) the TwoUse view, which integrates UML classes and OWL classes and, relying on OCL-DL, defines query operations that use reasoning services. The OWL View consists of eight classes, which are needed to realize the duty free order. The UML View has the same five classes from Fig. 1. Applying the rules, the TwoUse View should have seven classes and an OCL-DL query operation.

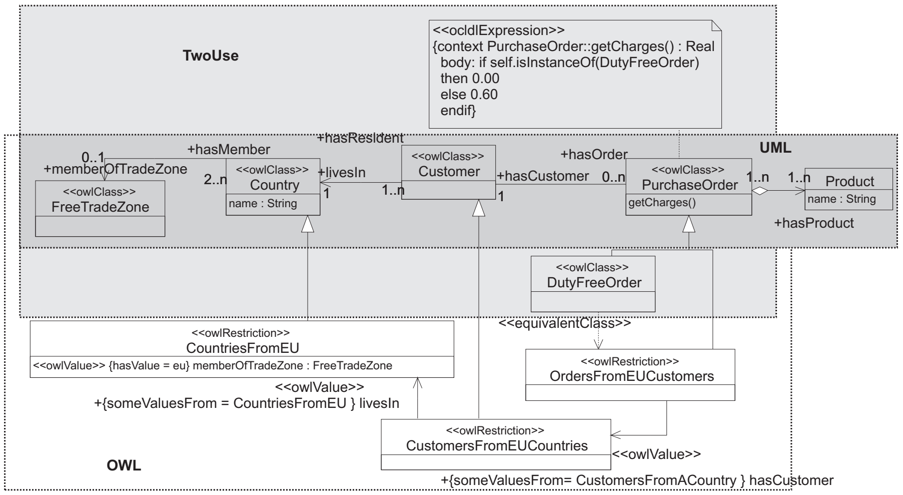

Fig. 5. UML Class Diagram profiled with UML Profile for OWL and TwoUse Profile

Although we reuse an UML profile for OWL to map onto TwoUse classes, OWL classes referred by any of the OCL-DL pre-built operations must be TwoUse too. The TwoUse class is the bridge that links OWL elements with OCL-DL expressions. And to be compatible with tools that support UML2 extension mechanisms that don't support OCL-DL, these expressions must be specified with the stereotype `<<ocldlExpression>>` with base class of UML metaclass `UML::Opaque Expression`. This stereotype has the property `referredOwlClass [*]` and the values are the referred classes.

With these considerations we have the expressiveness needed to design mapping from a profiled UML diagram onto TwoUse models, instances of the TwoUse metamodel.

3.4 Mappings

This section presents guidelines to map from the hybrid diagram onto the TwoUse model, and illustrates how code can be generated from the models. We do not intend to expose all mapping rules but only the ones of classes that are more important for understanding our proposal.

Model to Model. The elements of the TwoUse view in the hybrid diagram map basically onto instances of two kinds of elements of the TwoUse metamodel: `TUClass` and `TUPackage`. The OCL-DL expressions map onto instances of the OCL-DL metamodel. The elements of the OWL view in the hybrid diagram map onto instances of the OWL metamodel. The elements of the UML view are copied, because they don't need to be changed.

The relationships among elements from the TwoUse view and elements from the others are preserved, as the TwoUse metamodel specializes both UML and OWL metamodel. No direct relationship is allowed between OWL and UML without TwoUse.

The rules to map classes in the hybrid diagram onto TwoUse classes are: (1) any class that has the stereotype `<<owlClass>>` and has any operation or any UML property declared; or (2) any class with stereotype `<<owlClass>>`, of which the name is a property value of `ReferredOwlClass` property of the stereotype `<<ocldlExpression>>`.

Any classes with the stereotype `<<owlClass>>` and only properties with the stereotype `<<datatypeProperty>>` or `<<objectProperty>>` that are not mapped onto TwoUse classes are mapped onto OWL classes. The other elements are mapped according to their stereotype.

Any class without any stereotype results in a regular UML class. The properties can be available on ontologies and be accessible from both sides or can remain only in UML. A TwoUse package is any package that has a TwoUse class. The rules used to map onto a TwoUse class are reused to verify whether a package has any TwoUse classes and map it onto a TwoUse package.

The UML Opaque Expresssions stereotyped with `<<ocldlExpression>>` are mapped onto OCL-DL. The OCL-DL and OCL expressions share the same structure and conform basically with the same metamodel. OCL-DL expressions can have in their body OCL ones.

In a model driven approach, mappings and transformations onto different levels of abstraction are necessary too. The next section describes mappings onto a lower level of abstraction.

Model to Code. The model-to-code mappings are highly dependent of the platform language. We present the java code that implements the multiple inheritance of the `TUClass` multiple inheritance via Java Interfaces.

The class `PurchaseOrder` extends the class `OWLIndividual`. Here we show how the operation `getCharges()` that calls the OCL-DL operation `owlIsInstanceOf` could be implemented:

```
public class PurchaseOrder extends OWLIndividual {
   ...
   public double getCharges(){
      OWLClass OWLDutyFreeOrder = model.getOWLClass("DutyFreeOrder");
      if (this.owlIsInstanceOf(OWLDutyFreeOrder)){ return 0; }
      else {return 0.60;}
```

```
    };
    public boolean owlIsInstanceOf(OWLNamedClass owlClass) {
        ...
        OWLReasoner reasoner = reasonerManager.getReasoner(model);
        OWLIndividual individual = this;
        return reasoner.isInstanceOf (owlClass, individual);
    }
    ...
}
```

3.5 Model Driven Engineering with TwoUse

With the intention of summarizing and providing a complete view of our proposal, Fig. 6 presents a Model Driven View of the TwoUse idea, using modeling spaces [8]. The four lanes, UML, TwoUse, RDF/XML and Java are grouped into three modeling spaces: MOF, RDF(S) and EBNF. For each lane we show three modeling levels according to the OMG's Four layered metamodel architecture: the metametamodel level (M3), the metamodel level (M2) and the model level (M1). The relationships inside each quadrant show dependency, the ones that cross vertical borderlines are transformations, the ones that cross the horizontal borderline mean instantiation. The boxes inside each quadrant represent packages.

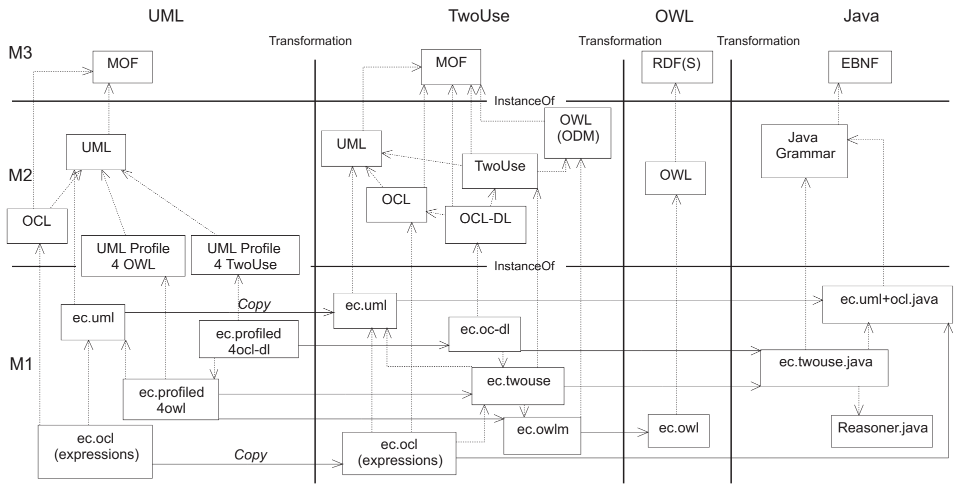

Fig. 6. layered metamodels and modeling spaces of the TwoUse Approach

The M3 quadrant of the *UML lane* in the MOF modeling space shows the MOF metametamodel, whereas M2 illustrates the organization of the UML metamodel with OCL and the two profiles concerning our approach: an UML profile for OWL and another for TwoUse, which contains only a stereotype for OCL-DL expressions and its properties. The UML profiles conceptually span both M2 and M1 levels [9]. The M1 quadrant of the same lane shows possible resulting packages from designing our e-commerce example using an UML tool. The hybrid diagram is composed of: pure UML elements (`ec.uml`) with their OCL expressions (`ec.ocl`); elements stereotyped by the UML profile for OWL (`ec.profiled4owl`); and the opaque expressions stereotyped by the UML profile for TwoUse, i.e., the OCL-DL expressions (`ec.profiled4ocl-dl`).

The M2 quadrant of the *TwoUse lane*, also in the MOF modeling space, presents the organization of the packages in the TwoUse metamodel. The TwoUse package imports the UML and OWL metamodels and specializes elements from both, as shown in Sect. 3.1. The OCL-DL package imports the TwoUse and the OCL packages and makes viable checking, parsing and code generation from expressions that manipulates OWL elements, as explained in Sect. 3.2.

The M1 quadrant of the same lane represents the packages of models generated from the hybrid diagram. The UML and OCL elements are copied, because they don't need to be transformed. The package `ec.profiled4owl` serves as basis for two transformations: (1) the generation of the OWL model (`ec.owlm`), instance of OWL metamodel and (2) the generation of TwoUse classes in the TwoUse package (`ec.twouse`), as explained on Sect. 3.4. The expressions stereotyped by the UML Profile for TwoUse are transformed into the model (`ec.ocl-dl`), instance to the metamodel OCL-DL.

The *OWL lane* in the RDFS modeling space represents the normative OWL exchange syntax according to RDFS metamodel. The transformation from the ontology model (`ec.owlm`), which is an instance of the MOF based OWL metamodel, into the OWL exchange syntax RDF/XML is doable using XLST, since all M1 models are stored using the XMI standard.

The *Java lane* in the EBNF modeling space aims at representing the platform specific model of choice and it is not our purpose detail the mappings. The M2 quadrant represents the Java grammar, as the M1 quadrant of the same lane represents java packages. Basically we generate two packages: one with the implementation of the UML elements and expressions (`ec.uml+ocl.java`), and one with the implementation of the TwoUse classes (ec.twouse.java), as described on Sect. 3.4. The later imports packages that implement the reasoner (`reasoner.java`) and depends on the prior two.

4 Evaluation and Related Works

After introducing and explaining our solution, we revisit the requirements to evaluate how our approach fulfills them in comparison to other approaches. Table 1 summarizes the requirements evaluation.

Ontologies design (1) is viable by any of the solutions, but using just UML, as proposed by ODM specification, lacks for expressiveness. The better solution should be using a UML Profile for OWL. Using pure UML would not be expressive enough to have, in the same diagram, OWL and UML (5). It would not be

Table 1. Qualitative evaluation of functional requirements

	Requirements	UML	OWL Profiled UML	**TwoUse**
1	Ontologies Design	+/-	+	+
2	UML Tools Compatibility	+	+	+
3	OCL Support	+	+/-	+
4	Code Generation Support	+/-	+	+
5	UML and Ontology Modeling Integrated	-	+	+
6	Model Driven Engineering Support	+/-	-	+
7	Reasoner-query Operations Specification	-	-	+

possible to differentiate UML classes from OWL ones. Pure UML and stereotyped UML classes allied with mappings could be enough to design UML models and ontologies simultaneously. At this point, TwoUse and an OWL profiled UML solution share the same strategy. Then they are compatible with UML Tools that support UML2 extension mechanisms (2). Different MOF based metamodels and UML Profile for OWL Ontologies are available [10] [11] [6], some of them with new adornments. Our approach gives the developer the freedom to choose which profile to use, provided that the profile has the sufficient semantics to support mappings onto TwoUse models.

OCL is integrated with UML by nature (3). Using an OWL profiled UML enables OCL expressions to be written only in context of pure UML classes. Since a TwoUse class is an specialization of an UML class and a OWL class, all OCL expression can used with TwoUse classes automatically. Thanks to OCL-DL, only TwoUse supports reasoner-query operations specification (7), which brings additional power to the development of systems that use ontologies. That is one of the main points of our approach.

All solutions support code generation (4), relying on model mappings. Generation of code from UML models is quite common, as there is a range of strategies to generate code from an ontology [12] [13] [14] [15]. But UML cannot be used to generate code for object oriented and ontology driven approaches at the same time (5) due to lack of sufficient expressivenesss.

Finally, a profiled solution is not transparent enough to fully support Model Driven Engineering (6), to have a metamodel in order to validate models, and to allow metamodel to metamodel transformations. Neither UML can combine structural and behavioral features description with predicate definitions of classes. These characteristics are entirely supported by TwoUse.

As an alternative strategy to support MDE, one might think of applying model weaving techniques to create bridges between models. The Generic Model Weaver [16] enable the extension of a generic metamodel to define the links. However, this strategy would need two or more models, as with TwoUse the developers in fact model just a hybrid one. Furthermore, it would be hard to fit OCL-DL into this approach. Model weaving techniques could be useful to integrate and to reuse models already available in order to generate TwoUse models.

5 Conclusion

Based on some requirements like specification of operations that query a reasoning engine and Model Driven Engineering support, this paper proposes an approach able to capture some structural and behavioral features and allows modelers to describe the semantics of the domain at the level of a OWL ontology. We propose bridges based on a metamodel, library extensions and mappings. TwoUse achieves improvements like reusability and maintainability even for development of non-logical systems.

References

1. OMG: Unified Modeling Language: Superstructure, version 2.1.1. Object Modeling Group. (2007)
2. OMG: Object Constraint Language Specification, version 2.0. Object Modeling Group. (2005)
3. Gruber, T.R.: A translation approach to portable ontology specifications. Knowledge Acquisition **5**(2) (1993) 199–220
4. Staab, S., Studer, R., eds.: Handbook on Ontologies. International Handbooks on Information Systems. Springer (2004)
5. Mcguinness, D.L., van Harmelen, F.: OWL web ontology language overview (2004)
6. OMG: Ontology Definition Metamodel. Object Modeling Group. (2005)
7. Knublauch, H., Oberle, D., Tetlow, P., Wallace, E.: A semantic web primer for object-oriented software developers. W3C Working Group Note 9 March 2006, W3C (2006)
8. Djurić, D., Gašević, D., Devedžić, V.: Adventures in modeling spaces: Close encounters of the semantic web and MDA kinds. In Kendall, E.F., Oberle, D., Pan, J.Z., Tetlow, P., eds.: Workshop on Semantic Web Enabled Software Engineering (SWESE 2005), Galway, Ireland (2005)
9. Atkinson, C., Kühne, T.: Profiles in a strict metamodeling framework. Sci. Comput. Program. **44**(1) (2002) 5–22
10. Brockmans, S., Volz, R., Eberhart, A., Löffler, P.: Visual modeling of OWL DL ontologies using UML. In: International Semantic Web Conference. (2004) 198–213
11. Djurić, D., Gašević, D., Devedžić, V., Damjanovic, V.: A UML profile for OWL ontologies. In: MDAFA. (2004) 204–219
12. Eberhart, A.: Automatic generation of java/sql based inference engines from RDF schema and RuleML. In: ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web, London, UK, Springer-Verlag (2002) 102–116
13. Völkel, M., Sure, Y.: RDFReactor – from ontologies to programmatic data access. In: Poster Proceedings of the Fourth International Semantic Web Conference. (2005)
14. Kalyanpur, A., Pastor, D.J., Battle, S., Padget, J.A.: Automatic mapping of OWL ontologies into java. In Maurer, F., Ruhe, G., eds.: SEKE. (2004) 98–103
15. Knublauch, H.: Ontology-driven software development in the context of the semantic web: An example scenario with Protege/OWL. In Frankel, D.S., Kendall, E.F., McGuinness, D.L., eds.: 1st International Workshop on the Model-Driven Semantic Web (MDSW2004). (2004)

16. Fabro, M.D.D., Bézivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: a generic model weaver. In: Proceedings of the 1ére Journé sur l'Ingnierie Dirigé par les Modles (IDM05). (2005)