Defining and Interpreting OCL Constraints on Arbitrary Models and Model Instances

Claas Wilke, Michael Thiele, and Christian Wende
{claas.wilke,michael.thiele,c.wende}@inf.tu-dresden.de

Technische Universität Dresden
Department of Computer Science
Institute for Software and Multimedia Technology
Software Technology Group

Abstract. In recent years the Object Constraint Language (OCL) has become a popular constraint language that is used on top of multiple modeling languages such as UML, EMF Ecore and various domain specific languages. OCL is only loosely coupled to these modeling languages, it is rather common to reuse the same OCL parsers for multiple different modeling languages. Nevertheless, dynamic OCL semantics as required for OCL constraint interpretation is typically implemented in a specific programming language and thus limited to specific programming or modeling languages. In this paper, we propose an architecture for an OCL tool that encapsulates meta-models, models and model instantiations behind well-defined interfaces. We present an OCL interpreter that can interpret OCL constraints on various different model instantiations and provides reuse of the same dynamic OCL semantics implementation for all these instantiation types. The architecture is evaluated by presenting three case studies defining and interpreting OCL constraints on multiple models and model instantiations.

Key words: OCL, MDSD, Modeling, Constraint Interpretation.

1 Introduction

Model-Driven Software Development (MDSD) aims to abstract from concrete software implementations and uses modeling languages to describe software systems. Model languages include the Unified Modeling Language (UML) [OMG09], Ecore of the Eclipse Modeling Framework (EMF) [URL10b] and domain-specific modeling languages (DSLs). The Object Constraint Language (OCL) [OMG10] was originally designed as an extension of the UML to express structural and functional constraints on UML models in a textual syntax. In recent years, OCL has become a popular constraint language for other modeling languages like EMF Ecore and various DSLs as well. In general, OCL can be defined and used as a constraint language on top of every modeling language, that contains basic modeling concepts like types and navigable properties.

When integrating OCL with such a modeling language, the concrete and abstract syntax of OCL is not altered. As OCL is statically typed, the abstract

syntax only needs to have access to the types of the model it is defined on. Thus, a generic implementation of an OCL tool that supports OCL constraints on different types of models can be designed and implemented if there is an abstraction of the underlying model type system.

One approach to evaluate the dynamic semantics of OCL constraints on instances of constrained models is *Constraint Interpretation*. Although OCL is defined abstractly and independent of specific implementations, OCL interpreters are normally limited to specific programming or modeling languages. Thus, a reimplementation of OCL's dynamic semantics is required for every type of model instantiation. Each reimplementation leads to code duplication, limits reuse and makes it more difficult to maintain or extend the OCL semantics for all types of model instances synchronously.

In this paper, we propose an architecture for an OCL tool that encapsulates meta-models, models and model instantiations behind well-defined interfaces. Thus, it is possible to reuse the complete OCL infrastructure including the parser, standard library and interpreter for every combination of different model and instantiation types. An implementation of the approach is used in our tool *DresdenOCL* [URL10c], released in its third version under the name of *Dresden OCL2 for Eclipse*. *DresdenOCL* is able to parse OCL constraints on different types of models such as UML, EMF Ecore, Java and XML Schema and interpret them on different model instantiations such as Java objects, XML files or UML diagrams.

TODO The paper is structured as follows...

2 The Generic Architecture of Dresden OCL2 for Eclipse

To differentiate between meta-models, models and runtime objects, the OMG introduced the *MOF Four Layer Metadata Architecture*, that locates the meta-model, model, and the model's instances at the layers *M2*, *M1* and *M0*. Each language at layer *Mn* is described in terms of a language resided at layer *Mn-1*. Meta-meta-models that are located at the layer *M3* can describe themselves reflexively.

As all models can be located in this layer architecture, OCL can be located there as well. OCL constraints are a model and can be described by their abstract syntax, i.e., their meta-model. Constraints are evaluated for runtime objects at the model instance layer. Thus, three layers must be known by the OCL tool to define and evaluate the constraints. This notion leads to the *Generic Three Layer Meta-Data Architecture* [DW09] that allows to define an OCL tool independently of specific layers. As can be seen in Figure 1 the OCL model (all constraints) enriches another model. In order to navigate through the model or to call operations on model elements, OCL needs to know the structure of the model it is defined on. Therefore, the abstract syntax of OCL extends the model's meta-model.

The architecture of Dresden OCL2 for Eclipse was developed in respect to the generic three layer meta-data architecture. In order to reuse the developed OCL

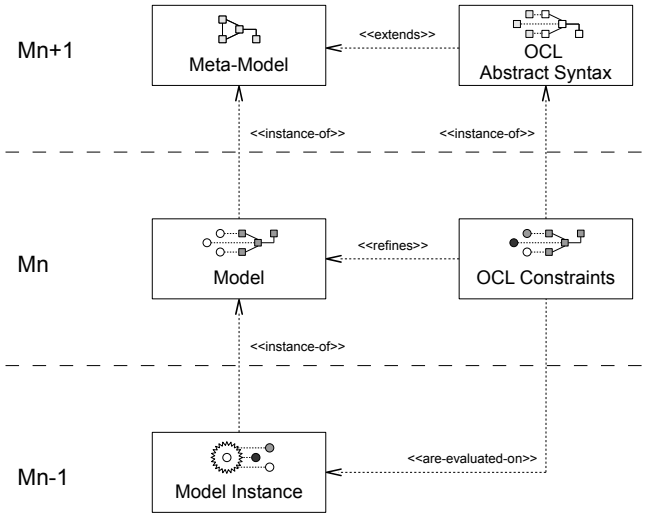

Fig. 1. The *Generic Three Layer Metadata Architecture*. Each OCL constraint requires a meta-model defining a model on which OCL constraints are specified. The constraints are evaluated on a model instance that implements the constrained model. In this context, a model instance can be a set of runtime objects or a model modeled in a constrained meta-model!

tools, DresdenOCL does not directly accesses models or model instance objects. Instead, these are hidden behind a common set of interfaces that delegate to their adaptee. The adaption of models and model instance objects is presented in the following.

2.1 Model Adaptation

Handling different types of models behind common interfaces is the basis to define a generic OCL meta-model and abstract syntax that can be used to define and parse OCL constraints on different meta-models. Thus, Dresden OCL2 for Eclipse is based on a *Pivot Model* designed by Matthias Bräuer [BD08] that abstracts the concepts of various meta-models such as the UML meta-model and the EMF Ecore meta-model. It describes the basic concepts that must be provided by a meta-model to define OCL constraints such as types, properties, operations and parameters.[1] The OCL tools only invoke the interfaces of the pivot model concepts when they want to reason on types defined in the adapted models (e.g. the OCL2 parser can invoke the operation `getType()` to reason the `Type` of an `Operation` or `Property`. Thus, the OCL2 Parser does not need to

[1] All these concepts exist, but a meta-model can be adapted to a subset of these concepts as well. E.g, the XML meta-model XML-Schema cannot express operations. Nevertheless, it can be adapted to the pivot model.

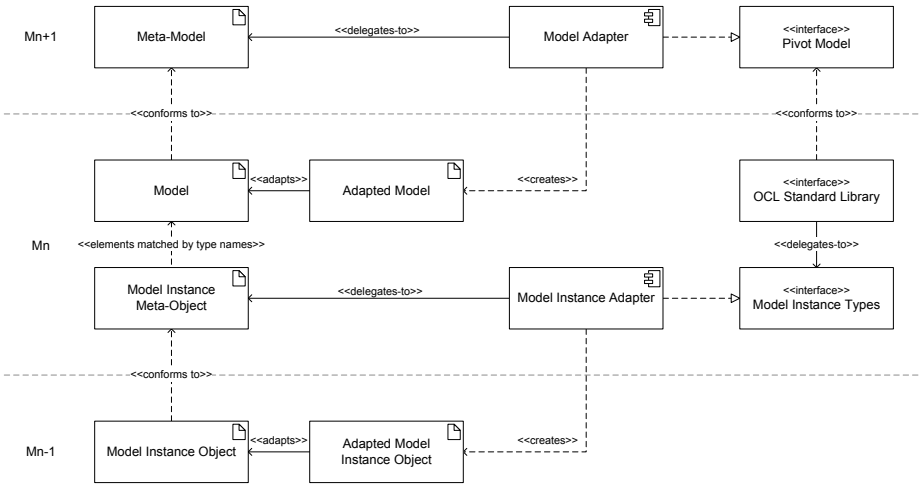

Fig. 2. The adaptation architecture of Dresden OCL2 for Eclipse. At the Mn+1 layer, meta-models are adapted to the pivot model. The adapter is responsible to wrap modeled models of the adapted meta-model as wrapped models as well. At the Mn layer, model implementation types can be adapted. Their adapter is responsible to wrap instances (or runtime objects) to wrapped model objects. The OCL standard library implements the logic to evaluate operation defined on OCL types. Other request such operation invocations or property request on wrapped objects are delegated via the interfaces of the model implementation type model.

know the adapted meta-model and can be connected with different meta-models very easily.

For every meta-model that shall be connected with Dresden OCL2 for Eclipse, a *Meta-Model Adapter* has to be implemented (see Figure 2, Mn+1 layer). The adapter contains implementations for all adapted pivot model elements (e.g., the UML meta-model element `UMLClass` is adapted to the pivot model element `Type`). Furthermore, the meta-model adapter contains a factory that creates adapters for currently accessed model elements (see Figure 2, Mn+1 and Mn layer). The model element adapters are only created when they are required and existing adapters are cached. Thus, we avoid unecessary and expensive adaptation, especially when working on large models of which only parts are constrained using OCL.

2.2 Model Implementation Adaptation

Similar to the adaptation of different meta-models and models (that can be considered as the specification or structure of a modeled system), it is also possible to adapt different kinds of implementations or semantics behind common interfaces to use the same OCL2 interpreter for the interpretation of OCL constraints

on different types of model implementations. Furthermore, a specific model type can have different semantics in different implementations. E.g., a UML class diagram can be implemented in Java, C#.[2] Thus, it is sensible to loosely couple the models and their semantics.

To fulfill these requirements, we designed a common *Model Implementation Type Model* that can be adapted to different kinds of model implementations. The implementation type model can be considered as similar to the pivot model, but has some differences as well. Similar to the pivot model, the implementation type model defines different interfaces for different elements of an implementation such as instances of primitive types, enumeration literals, collections and instances of normal types defined in the model. All these interfaces inherit a common interface *IModelInstanceElement*. The most important difference between the pivot model's and implementation type model's interfaces is that `IModelInstanceElements` have to provide a reflection mechanism whereas pivot model elements only allow to reason on relationship between elements of the same meta-level. During interpretation, the OCL2 interpreter must be able to reason on the Type of a *IModelInstanceElement*, to retrieve property values and to invoke operations. The kind of reflection mechanism provided can be considered as similar to the Java reflections that allow to reason on types, to cast objects and to access properties and operations.

Each kind of model implementation that shall be connected with Dresden OCL2 for Eclipse has to be adapted via a *Model Implementation Adapter* (see Figure 2, Mn layer). Similar to the meta-model adapter, the model implementation adapter has to provide adapters for the different concepts of the implementation type model and has to provide a factory that creates *Model Object Adapters* for the runtime objects of the adapted implementation (see Figure 2, Mn-1 layer). Similar to the meta-model factory, a model implementation factory only creates adapters for objects that are requested by the OCL2 interpreter during interpretation. Already adapted objects are cached to improve the performance and to avoid phenomenas like *Object Schizophrenia*.

2.3 Challenges

During the implementation of our generic architecture that allows connection of different types of models with different types of model implementations and vice versa we had to focus on some problems and difficulties that are shortly presented in this section.

As mentioned above, models and model implementations are only loosely coupled to support different implementations for the same type of model. Thus, we had to realize a matching algorithm that matches types of an implementation to the types of a given model. Currently, this type matching is realized by simply matching the names of the types (including the names of their enclosing name spaces where possible). Often, this solution is error prone and further

[2] Although the implementation of the same model in different programming languages is rather unusual, it is often required to connect models of the same meta-model with different types of implementations.

information is hidden behind our adapters that could be used to improve the algorithm. E.g., when coupling an instance of an EMF Ecore model, Eclipse provides further mechanisms to reason on the types of EObjects. Thus, we plan to improve our matching algorithm that is currently scattered over multiple classes of each implementation type adapter and plan to introduce a set of type matching strategies that can be implemented based on the *Chain of Responsibility* pattern[GHJV95]. The chain could start by trying to match the types using a implementation specific matcher and end by trying to simply match the type's names as currently done.

Another problem when using adapters for model implementation types is the unwrapping mechanism of adapted elements when invoking operations on the implementation's elements. E.g., to invoke an operation of a Java implementation we require `Objects` as parameters instead of `IModelInstanceElements`. This unwrapping mechanism is easy where elements that have been adapted before are simply unwrapped again. Unfortunately, during interpretation of OCL constraints, new instances of primitive types or new collections can be created (e.g., when invoking the OCL operation `size()` that returns an `Integer` instance. Thus, the factory of a model implementation type has to provide operations to reconvert primitive types and collections to elements of the adapted model instances. In some cases this can become rather complicate because the adaptation between types of the implementation and the model implementation type interfaces has not to be bijective. For example Java `ints` and `java.lang.Integers` are both mapped to `IModelInstaceIntegers`. During unwrapping, the factory of the Java implementation type has to reflect whether the method to invoke requires an `int`, an `Integer` or another Java integer-like type's instance. All in one, the unwrapping mechanism of an adapted implementation can be considered as the most complicate and error-prone part of the complete implementation type adaptation. Nevertheless, this part is necessary to support interpretation of OCL constraints containing operation invocations on model defined types.

2.4 Improving the Adaptation Process

Adapting different types of meta-models and model implementations to Dresden OCL2 for Eclipse is easier than writing new OCL tools for each combination. Nevertheless, the adaptation process contains parts that are similar for each adaptation and can be error prone as well. When the adaptation of a model implementation type is wrong, the interpretation results may be wrong as well. To improve the adaptation process, we developed a code generator for the adaptation of meta-models to the pivot model. The code generator requires an annotated EMF Ecore model describing the adaptation of meta-model elements to the pivot model elements. The code generator generates the skeleton code for all required adapters and thus avoids manual implementation of these adapters. For model implementation types, such a code generator is currently missing.

Furthermore, we developed two generic JUnit test suites, that can be used to test the correct adaptation of a meta-model or model implementation type, respectively. The test suites require a specific test model modeled in the adapted

meta-model or a specific test implementation implemented in the adapted model implementation type. The test suites then check if all required methods to reason on types, operations, properties etc. are implemented appropriately in respect to the specified test model or test implementation. These generic test suite helped us to ensure that all existing adaptations behave in the same expected manner and to easily detect wrong adaptations of some elements.

3 Case Studies

In this section we shortly present three case studies we used to investigate the power of our generic interpretation approach. Please be aware of the fact that the case studies are examples for adaptation only. Multiple other adaptations are possible and tasks for future works.

3.1 The Royal and Loyal System Example

As a first case study, we modeled and implemented the *Royal and Loyal System Example* as defined in [WK03]. The case study was desigend by Warmer and Kleppe to explain the Object Constraint Language and thus, can be used to test the interpretation of almost all possible kinds of OCL expressions. The case study contains 13 classes (including inheritance and enumeration types). We modeled the Royal and Loyal System by using the UML2 meta-model of the Eclipse Model Development Tools (MDT) project [URL10a]. We took over 130 constraints from the book and implemented instances in Java to interpret these constraints on Java objects.

To load the Royal and Loyal System Example into Dresden OCL2 for Eclipse, the MDT UML2 meta-model had to be adapted by a meta-model adapter (see Figure 3). Here, the meta-model adaptation was realized at the layer M2. At M1, the class diagram describing the Royal and Loyal System Example was adapted as an `IModel`. Furthermore, the Java implementation was adapted as an implementation of this model. E.g., the Java class `LoyaltyAccount` was mapped as an implementation of the UML class `LoyaltyAccount`. Please be aware of the fact, that both classes are located at the M1 layer because the Java class is only another representation of the same class as described by the UML class! To load the Java classes as a model implementation, we had to implement a model implementation adapter for Java. This adapter was also responsible to adapt the Java objects to `IModelInstanceElements` at the M0 layer. The Royal and Loyal case study proved a sound implementation of the OCL Standard Library specification and also a correct adaptation of the MDT UML meta-model and Java model instance objects.

3.2 SEPA Business Rules

The company Nomos Software provides a service to check busines rules on financial SEPA messages that are required for financial transactions of bank offices

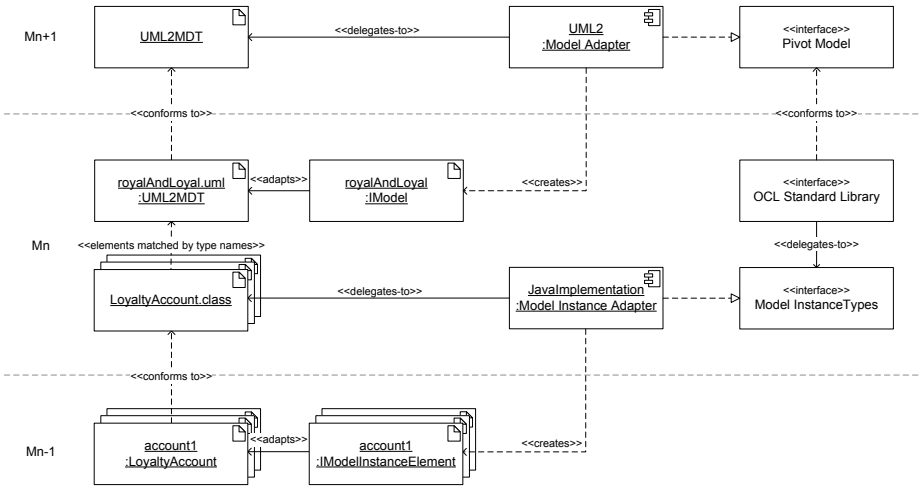

Fig. 3. In the Royal and Loyal case study, the MTD UML2 meta-model is adapted at M2. At M1, the Royal and Loyal UML class diagram is adapted as a model and its Java classes are adapted as a model implementation. At M0, each object of the Java classes is adapted as a model instance element.

as defined by the European Payment Council (EPC), ISO20022, and the Euro Banking Association (EBA) [IS006,EPC09]. The SEPA messages are described in XML documents that are instances of XML Schema Definitions (XSD). The service of Nomos Software provides the possibility to validate XML files against a set of business rules that are defined in OCL on the XSD the XML files are instances of.

We implemented an XSD meta-model and an XML model implementation adaptation for Dresden OCL2 for Eclipse and used the same XSD and XML files that are provided with an online demo of the Nomos service[3] to test the adaptation (see Figure 4). At the M2 layer, the XML Schema meta-model was adapted by a meta-model adapter. At the M1 layer, the `pain.008.001.01.xml` was adapted as an `IModel`. An adaptation for XML documents as model implementations was realized as well. Each type of a node in the XML document was mapped to a Type defined in the XML Schema at layer M1. At layer M0, each node was adapted as an `IModelInstanceElement`. We took the same OCL business rules that Nomos Software uses to test the OCL2 Interpreter of Dresden OCL2 for Eclipse. About 120 constraints have been interpreted for three different XML instances of the XSD and the results have been compared with the results of the Nomos service demo to check the right interpretation of the constraints. The case study showed that the abstract description of model implementation types enables Dresden OCL2 for Eclipse to interpret constraints on

[3] Available at http://www.nomos-software.com/demo.html

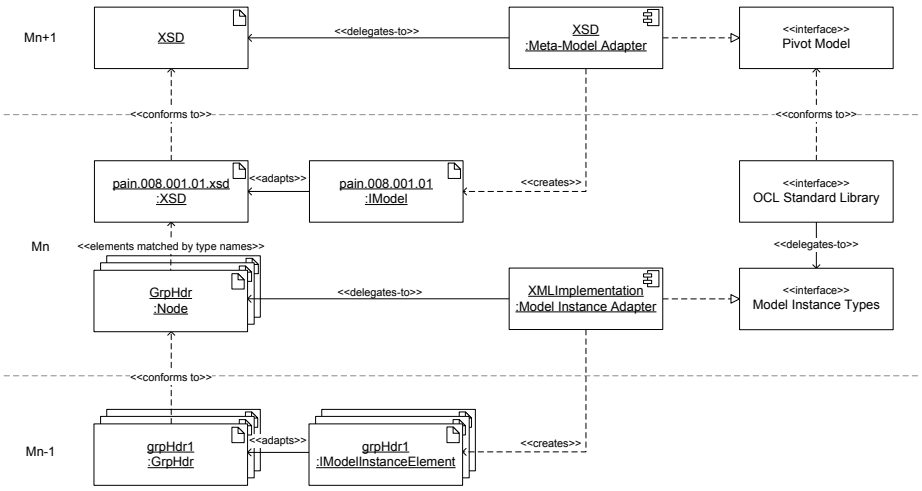

Fig. 4. In the SEPA case study, the XSD meta-model is adapted at M2. At M1, the Pain XML Schema is adapted as a model. The types of XML nodes are mapped to the types in the XSD file. At M0, each node of the XML instance is adapted as a model instance element.

XML nodes as well. The OCL2 Interpreter had not to be modified to interpret OCL constraints on XML nodes.

3.3 The OCL2.2 Standard Library

TODO

check for all operations that are given in the standard on the modelled SL and on the Interfaces (2 different instances for 1 model)

3.4 Future Work

For future case studies we plan to adapt meta-model and model implementations for web services (Meta-Model: WSDL, Model Implementation: TODO), static programming languages such as C# (Meta-Model: UML2, Model Implementation: C#), data bases (Meta-Model: SQL-DDL, Model Implementation: SQL).

TODO WFRs of OCL SL ...

4 Related Work

MDT OCL, USE, BluePrint OCL, AspectJ, ...

In [Har06,Har07] Hartmann presented an approach for XML-based template engines that showed a use case for OCL constraints on XML Schemas. OCL was used to express constraints that cannot be defined in XML Schema.

OCL on XML/DTD:

- http://lci.cs.ubbcluj.ro/ocle/
- Modeling XML applications with UML (D. Carlson) (Buch)

TODO

5 Conclusion

- Conclusion:
  - exchangability
  - execution layer can be reused as well
  - less errors, only one implementation, well tested

6 Acknoledgements

We want to thank Tricia Balfe of Nomos Software for providing their data for the XML case study and for continous feedback during adaptation of the case study.

TODO

References

[BD08] Matthias Bräuer and Birgit Demuth. Model-level integration of the ocl standard library using a pivot model with generics support. In David H. Akehurst, Martin Gogolla, and Steffen Zschaler, editors, *Ocl4All - Modelling Systems with OCL*, volume 9 of *Electronic Communications of the East (ECAST)*. Technische Universität Berlin, Germany, 2008.

[DW09] Birgit Demuth and Claas Wilke. Model and object verification by using dresden ocl. In *Proceedings of the Russian-German Workshop "Innovation Information Technologies: Theory and Practice", July 25-31, Ufa, Russia, 2009*, page 81. Ufa State Aviation Technical University, Ufa, Bashkortostan, Russia, 2009.

[EPC09] Sepa business-to-business direct debit scheme customer-to-bank implementation guidelines, October 2009.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Indianapolis, IN, USA, 2nd edition, 1995.

[Har06] Falk Hartmann. An architecture for an xml-template engine enabling safe authoring. In *Proceedings of the 17th International Conference on Database and Expert System Applications (DEXA06)*, pages 502–507. IEEE Computer Society, Washington, DC, USA, 2006.

[Har07] Falk Hartmann. Ensuring the instantiation results of xml templates. In *Proceedings of the IADIS International Conference WWW/Internet 2007*, pages 269–276, October 2007.

[IS006] Payments standards - initiation - unifi (iso 20022) message definition report, October 2006.

[OMG09] Omg unified modeling language^{TM}(omg uml), omg available specification, version 2.2, February 2009.

[OMG10] Object constraint language, version 2.2, February 2010.

[URL10a] Eclipse model development tools. Eclipse Project Website, April 2010.

[URL10b] Eclipse modeling framework (emf) project. Eclipse Project Website, April 2010.

[URL10c] Website of the dresden ocl toolkit. Sourceforge project Website, April 2010.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language - Getting Your Models Ready for MDA*. Pearson Education Inc., Boston, MA, USA, 2nd edition, 2003.