Extending Variability for OCL Interpretation

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, OCL advanced from a language used to constrain UML models to a constraint language that is applied to various object-oriented modelling languages. This includes Domain Specific Languages (DSLs) and meta-modelling languages like the Meta Object Facility (MOF) or Ecore. Consequently, it is rather common to provide variability for OCL parsers to work with different modelling languages. A second variability dimension relates to the infrastructure that models are realised in. Current OCL interpreters do not support such variability as they are typically bound to a specific realisation technique like Java, Ecore, or another specific model repository. In this paper we propose a generic adaptation architecture for OCL that hides models and model instances behind well-defined interfaces. We present how the implementation of such an architecture for DresdenOCL enables reuse of the same OCL interpreter for various model realisation techniques. Finally, our approach is evaluated in three case studies defining and interpreting OCL constraints on multiple models and model instances.

Key words: OCL, OCL Infrastructure, OCL Tool, MDSD, Modelling, Constraint Interpretation, Technological Spaces, Variability, Adaptation.

1 Introduction

Model-Driven Software Development (MDSD) aims to abstract from concrete software implementations and uses models to describe software systems. To ensure consistency of models, the Object Constraint Language (OCL) [1] has been developed as an extension of the Unified Modelling Language (UML) [2][3]. In recent years, OCL advanced to a constraint language used for various object-oriented modelling languages [4] and meta-modelling languages like the Meta Object Facility (MOF) [5] or Ecore [6].

~~In addition to the languages that OCL is combined with, a second variability dimension is the infrastructure that constrained models are realised, instantiated, and validated in.~~ While OCL constraints are defined on models, they are evaluated on instances of these models. Besides instances of constrained models stored in various model repositories like MOFLON [7], Netbeans MDR [5], or EMF [6], they also can be realised using classic programming languages like Java [8], or C# [9], database systems like relational databases [10], or XML [11].

Although several OCL tools support *variability at the model level*~~, i.e. the modelling language used to specify constrained models~~ [12][13], *variability at the model instance level*~~, i.e. the implementation infrastructure for model instances,~~ is not provided. For OCL compilers as presented in [5], [8], [9], or [10] such variability is not possible, as the code generated from OCL constraints needs to be bound to the realisation technique used at model instance level. However, for OCL interpreters, we argue that a decoupling of the semantics evaluation from a concrete model instance realisation technique is possible. In this paper we propose a *Generic Adaptation Architecture* for OCL interpreters that hides models and model instances behind well-defined interfaces. This enables reuse of the complete OCL infrastructure including the OCL parser, standard library and interpreter for various kinds of models and model instances.

The remainder of this paper is structured as follows. In Sect. 2 we analyse diverse applications for OCL and motivate two variation points for OCL interpretation. In Sect. 3 we discuss the design and implementation of a *Generic Adaptation Architecture* for OCL interpreters to realise the motivated variation points in *DresdenOCL*. In Sect. 4 we document the feasibility and benefits of our adaptation architecture by applying it to three case studies that use OCL with different models and model instances. In Sect. 5 we elaborate on lessons learnt during implementation and application of our approach. Finally, we present related work in Sect. 6 and conclude our contributions in Sect. 7.

2 Foundations

In the introduction section, different use cases for the definition of OCL constraints on different models and meta-models have been depicted. As identified in [8], OCL evaluation always requires three meta levels of the *MOF Four Layer Metadata Architecture* [14]. OCL can be considered an extension of a modelling language. Thus, its abstract syntax must extend a meta-model that defines types, navigable properties, and operations (cf. Fig. 1, (a), Mn+1 layer). This extension allows the definition of OCL constraints on models of this meta-model (Mn layer). During the evaluation of these constraints, an OCL interpreter has to query model instance elements for their properties or invoke operations on them (Mn-1 layer).

Model instances can be realised in a different *Technological Space* [15] than their model. Then, there are two model representations at the Mn layer: the constrained model and a *Model Realisation* that can be obtained by a *Reflection* [16] mechanism of the model instance. Thus, there has to be a matching of the model realisation to the constrained model.

Instances of this *Generic Three Layer Architecture* are shown in Fig. 1 (b) and (c). The first example illustrates the use of a UML class diagram constrained with business rules for which model instance elements are Java objects. At the layer M1 a technical space bridge can be observed, as the Java classes correspond to the

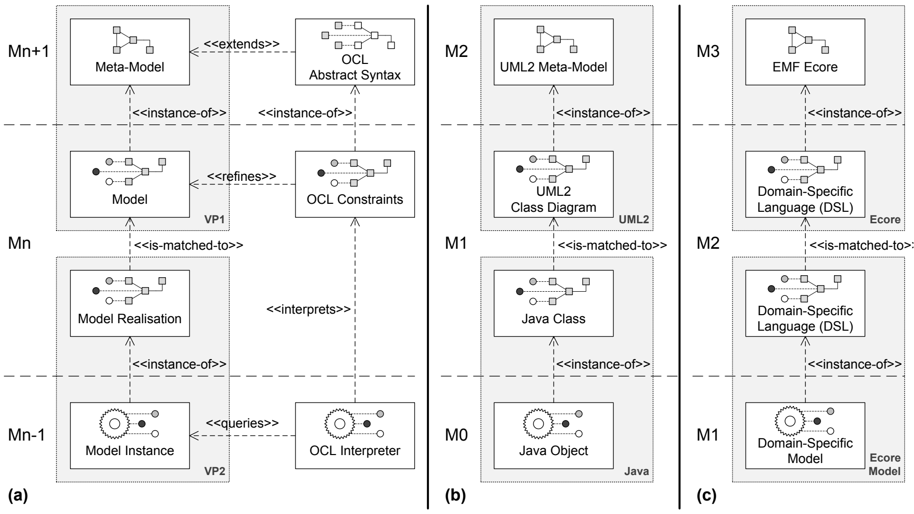

Fig. 1. The Generic Three Layer Architecture

UML model. The second example depicts the usage of OCL for Well-Formedness Rules (WFRs). Here, the model is a domain-specific language modelled in Ecore (M2) for which WFRs are evaluated on domain-specific models (M1). In this case, the model and its realisation are identical.

With the generic three layer architecture it is possible to combine various models and instances for constraint interpretation independently as depicted as a feature model in Fig. 2. The feature selection has to ensure that model instances are located exactly one meta-layer below their model. In the feature model two variation points regarding OCL interpretation can be identified: constrained models (VP1) as well as model instances (VP2) can vary.

In order to define constraints on different models, most OCL tools existing today, support VP1 [6, 13, 17, 18]. Yet, those tools do not address VP2 as their supported models require specific instances (typically located in the same technological space). As presented in this paper, this tight coupling is not necessary and a reimplementation of OCL interpreters for different technological spaces can be avoided. We propose an implementation of the generic three layer architecture to address this problem.

3 Implementation

In this section we discuss the implementation of a generic adaptation achitecture to realise the variation points identified in the previous section. First, we present *Model Adaptation* to address variation point VP1. Afterwards, we discuss how *Model Instance Adaptation* enables variation point VP2.

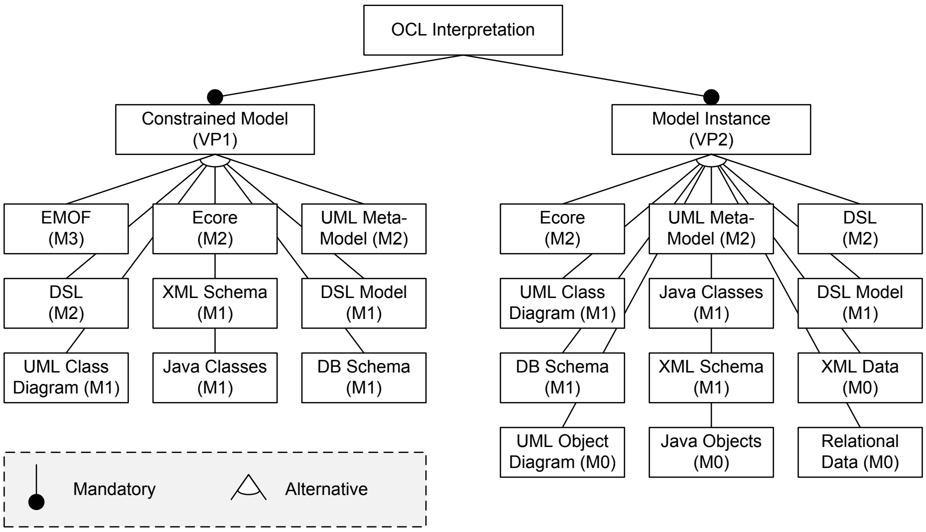

Fig. 2. Features of OCL Interpretation

3.1 Model Adaptation (VP1)

In order to define OCL constraints for various models, DresdenOCL provides a set of common interfaces abstracting structures that are required to navigate and query models. These interfaces – called *Model Types* (or *Pivot Model*) [12] – define the basic concepts such as `Types`, `Properties`, `Operations` and `Parameters` that bind OCL constraints to a concrete model (cf. Fig. 3). DresdenOCL uses these concepts to parse and statically analyse OCL constraints, e.g., the OCL2 parser can determine the `Type` of OCL expressions.

For every model that shall be connected with DresdenOCL, a *Model Adapter* component has to be implemented (cf. Fig. 4, Mn+1 layer). It contains individual adapters that map concepts of the model to corresponding artefacts of the Model Types. E.g., the UML meta-model concept `UMLClass` is adapted to the model type concept `Type`. Furthermore, the model adapter component has to create adapters on demand resulting in an *Adapted Model* (cf. Fig. 4, Mn layer). The adapters are only created for model elements that are required and existing adapters are cached. Thus, unnecessary and expensive adaptation is avoided, especially when working on large models of which only parts are constrained.

3.2 Model Instance Adaptation (VP2)

In our generic adaptation architecture we applied the same principles for model instances as those are also hidden behind a set of common interfaces. This enables the reuse of the same OCL interpreter for the dynamic evaluation of OCL constraints on model instances with various realisation techniques.

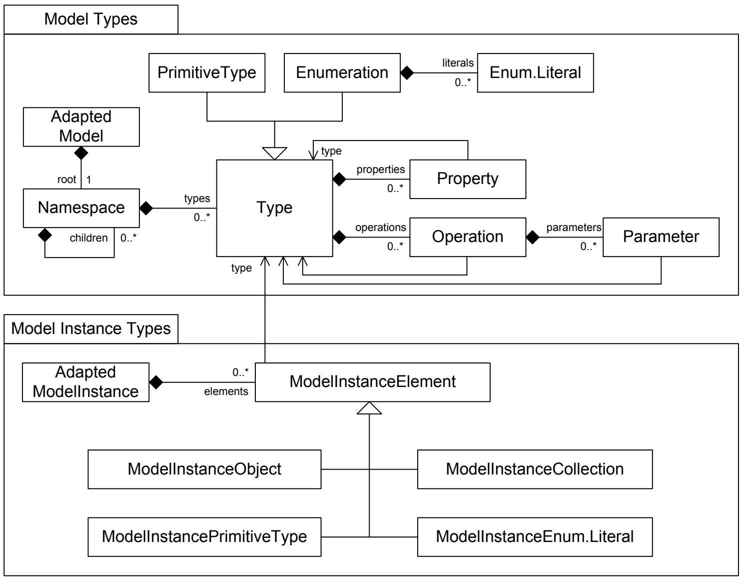

Fig. 3. Interfaces for model and model instance adaptation

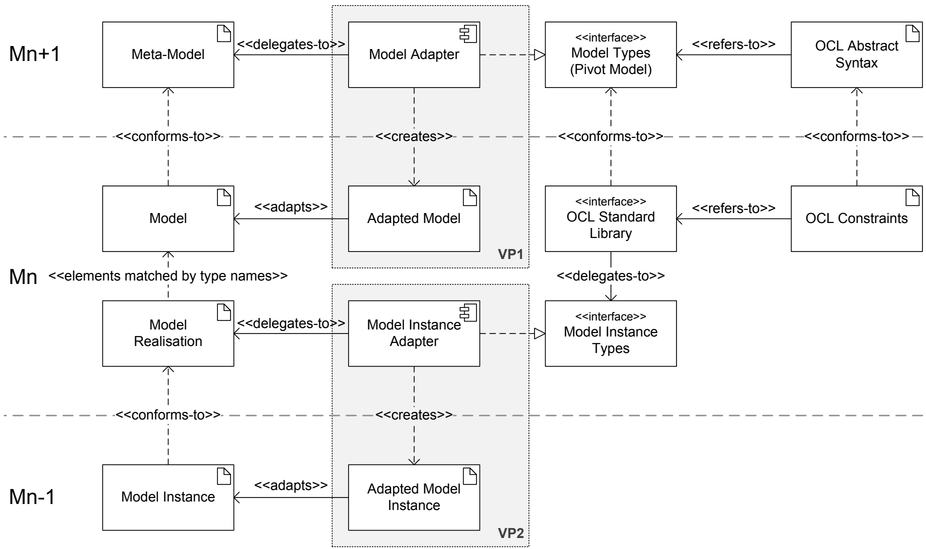

Fig. 4. The *Generic Adaptation Architecture* of DresdenOCL

To provide means for model instance adaptation, we introduced the *Model Instance Types*. The Model Instance Types are a set of `ModelInstanceElements` representing instances of primitive types, collections and objects defined in the model (cf. Fig. 3). [Discuss the following sentence. Right or wrong? What about type matching?] The most important difference between Model Types and Model Instance Types is that Model Types abstract modelling concepts whereas Model Instance Types abstract reflection capabilities of different model instances (cf. Fig. 4 Mn and Mn-1 layer). During interpretation, the OCL2 interpreter uses the reflection mechanism to retrieve the `Type` of a `ModelInstanceElement`, to access `Property` values, or to invoke `Operations`.

Each kind of model instance that shall be connected with DresdenOCL is adapted via a *Model Instance Adapter* component (cf. Fig. 4, Mn layer). The model instance adapter component contains adapters that map elements of a concrete model instance to Model Instance Types and has to create *Model Object Adapters* for the runtime objects of the adapted model instance (cf. Fig. 4, Mn-1 layer). Like the model adapter component, the model instance adapter component also creates adapters for objects on demand. Adapted objects are cached to improve the performance and to avoid phenomena like *Object Schizophrenia* [19]. [Claas: Reference Okay? Christian: Didn't florian send the original source? Always use the most concrete you know]

4 Case Studies

In this section we present three case studies to demonstrate the benefits of our generic adaptation architecture.

4.1 The Royal and Loyal System Example

As a first case study, we modelled and implemented the *Royal and Loyal System Example* as defined in [3]. The case study originally was designed by WARMER AND KLEPPE to teach the Object Constraint Language. It consists of 13 UML classes (including inheritance and enumeration types) and 130 constraints. We specified the royal and loyal system with a UML2 model (VP1) build with the Eclipse Model Development Tools (MDT) [6]. The model was implemented and instantiated in Java (VP2). Consequently, constraints were evaluated on Java objects.

The adapters required for the royal and loyal case study are shown in Fig. 5. To parse the Royal and Loyal constraints in DresdenOCL, a *UML2 Model Adapter* component was implemented. It adapts the required concepts of the UML2 metamodel to the Model Types of DresdenOCL at the M2 layer. Hence, the royal and loyal class diagram was adapted as a model at the M1 layer. For the Java implementation, a *Java Model Instance Adapter* component was implemented, that adapts the Java model elements (classes of the package `java.lang.reflect`) to the Model Instance Types. Thus, the objects of the royal and loyal Java implementation were adapted as `ModelInstanceElements` in DresdenOCL at the M0

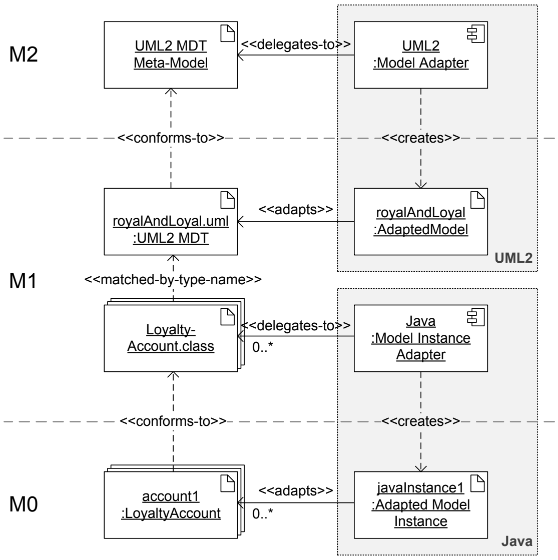

Fig. 5. Adapters used in the Royals and Loyals case study

layer. Elements of the class diagram as well as classes of the Java implementation are located at the M1 layer since the Java classes are only another representation of the UML class diagram.

The Royal and Loyal case study demonstrates that our generic adaptation architecture is able to support the common interpretation of OCL constraints defined on UML classes for Java objects.

4.2 SEPA Business Rules

In our second case study we interpreted OCL business rules defined on an XML schema (VP1) for XML documents (VP2) conforming to this schema. The Nomos Software company provides a service to check business rules on financial *Single Euro Payments Area (SEPA)* messages that are used in financial transactions of bank offices as defined by the *European Payment Council (EPC)*, *ISO20022*, and the *Euro Banking Association (EBA)* [20–22]. SEPA messages are described and shipped as XML documents. Nomos Software uses OCL constraints defined on XML schemas to validate XML documents against a set of business rules that ensure the consistency of SEPA messages. We implemented an *XSD Model Adapter* component and an *XML model Instance Adapter* component for DresdenOCL to evaluate about 120 constraints that are provided with the online demo.[c0]

The adapter required for the SEPA case study are shown in Fig. 6. To parse the SEPA constraints into DresdenOCL, an *XSD Model Adapter* component was

[c0] http://www.nomos-software.com/demo.html

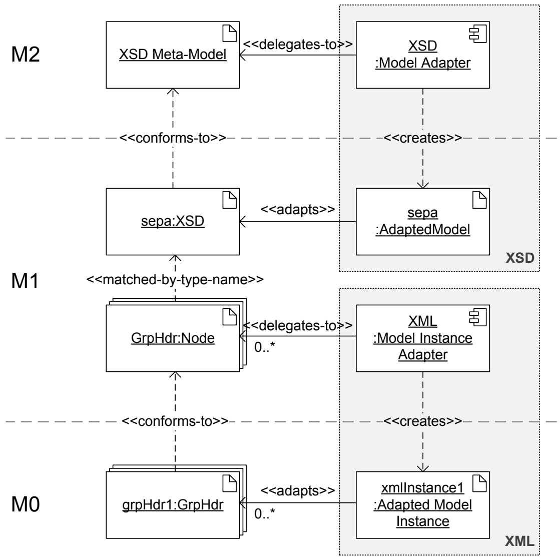

Fig. 6. Adapters used in the SEPA case study

implemented that adapts required concepts of the XSD meta-model to the Model Types of DresdenOCL at the M2 layer. Consequently, the *SEPA* XML schema was adapted as a model at the M1 layer. For the XML documents, an *XML Instance Adapter* component was implemented that adapts the XML model elements (mainly the class `org.w3c.dom.Node`) to the Model Instance Types. Thus, the nodes of the SEPA messages were adapted as `ModelInstanceElements` in DresdenOCL at the M0 layer.

The constraints were evaluated for three different XML files and the results have been successfully compared with the results of the Nomos demo. This shows that our model instance adaptation allows DresdenOCL to transparently interpret constraints on XML files as well. The OCL2 interpreter had not to be modified for the SEPA case study.

4.3 The OCL2.2 Standard Library

The last case study depicts the ability to load different model instances of one model in order to check for inconsistencies between both instances. In this example we checked Well-Formedness Rules (WFRs) for the OCL standard library of DresdenOCL. DresdenOCL's standard library is explicitly modelled as an instance of the Model Types, describing predefined OCL types like `Integer`, `OclAny` or `Sequence` and their associated operations. Hence, accessing those types is reduced to a simple model import while the model can conveniently be queried, validated or altered [12]. The WFRs can be used to check whether all OCL types are declared and whether they support all operations that are defined by the current OCL specification [1].

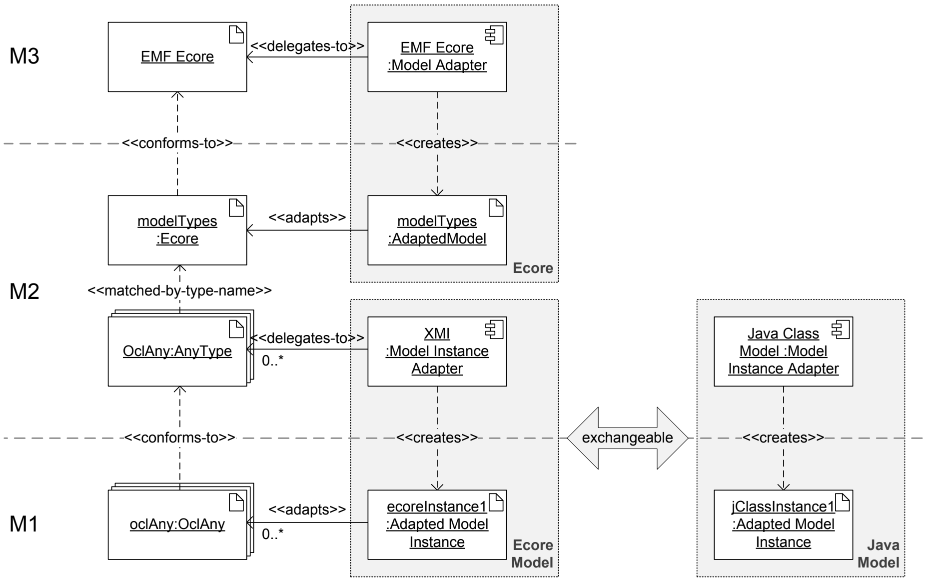

Fig. 7. Adapters used in the Standard Library case study

Although modelling the standard library leads to great flexibility, the standard library still needs an implementation that provides its dynamic semantics. This implementation is realised in Java. As there is no code generator for the Model Types, this can lead to inconsistencies between the modelled standard library and the according Java implementation. We propose to use OCL to check that all modelled types have an equivalent Java implementation and all modelled operations are also present in the Java interfaces.

Since the OCL standard library has been built conforming to the Model Types, an *EMF Ecore Model Adapter* component was required to parse OCL constraints defined on the Model Types (cf. Fig. 7, VP1). To evaluate the constraints on the modelled standard library, we implemented an *XMI Model Instance Adapter* component for instances of Ecore-based meta-models stored as XMI-files (VP2). For the Java-based standard library a *Java Class Model Instance Adapter* component was created. This adapter allowed us to load Java classes as a model instance (VP2) and to check for inconsistencies with the modelled standard library. The same WFRs used for the standard library model were evaluated for this instance.

This case study shows that the implemented OCL interpreter is not only independent of specific model realisation techniques, but can also use adapters of different meta layers. Thus, constraints defined on models, meta-models, and even meta-meta-models can be interpreted.

4.4 Future Case Studies

For future case studies we plan to implement new model adapters for variation point VP1 including *WSDL* and *SQL-DDL*. As further model instance adapters for variation point VP2, we intend to support *C#* and *Relational Databases*.

5 Lessons Learnt

In this section we highlight some problems that occurred during the design and implementation of our generic adaptation architecture for DresdenOCL. We present our solutions to these challenges and possible improvements.

Type Matching As models and model instances are only loosely coupled, model instance elements must be matched to a model type when they are imported into DresdenOCL. Currently, this type matching is realised by a simple type name match (including the names of their enclosing namespaces where possible). E.g., the Java class `LoyaltyAccount` is matched to the UML class `LoyaltyAccount` in the royal and loyal class diagram. This matching algorithm can be rather complex and often information is hidden inside the adapters that could be used to improve the matching. E.g., when adapting an instance of an EMF Ecore model, one could use the *Generator Model* provided by Ecore to retrieve further type matching information. We plan to improve this process that is currently scattered over multiple classes of each model instance adapter component by introducing type matching strategies that can be implemented using the *Chain of Responsibility* pattern [23]. The chain could start by trying to match the types using an implementation specific matcher and end by trying to simply match the type names as currently done.

Element Unwrapping Another problem when using adapters for Model Instance Types is the unwrapping mechanism of adapted elements when invoking operations on the `ModelInstanceElements`. E.g., to invoke an operation of a Java implementation we require `java.lang.Objects` as parameters instead of `ModelInstanceElements`. This unwrapping mechanism is easy for elements that have been adapted before, as they simply can be unwrapped again. Unfortunately, during interpretation of OCL constraints, new instances of primitive types or new collections can be created by the standard library (e.g., when invoking the OCL operation `size()` on a collection that returns an `Integer` instance). Thus, a model instance adapter has to provide operations to reconvert primitive types and collections into elements of the adapted model instances. In some cases this can become rather complicated as the adaptation between types of the instance and the model instance type interfaces has not to be bijective. For example, Java `ints` and `java.lang.Integers` are both mapped to `ModelInstaceIntegers`. During unwrapping, the Java model instance adapter component has to reflect whether the method to invoke requires an `int`, an `Integer` or another Java integer-like type instance. The unwrapping mechanism of an adapted instance can be considered as the most complicated and error-prone part of the complete model instance adaptation. Fortunately, model instances providing only structural information do not need this unwrapping mechanism as operations do not exist at all.

Automated Adapter Creation The adaptation process of models and model instances contains parts that are similar for each adaptation and thus can be automated. To improve the model adaptation process, we developed a code generator for the adaptation of model adapter components. The code generator requires an annotated meta-model describing the relation of meta-model concepts to the Model Types (e.g., the UML2 meta-class `Classifier` is annotated as a `Type`). The code generator generates the skeleton code for all required adapters that has to be completed manually. For the Model Instance Types, such a code generator is currently missing, but could be implemented as well.

Adaptation Testing We developed two generic JUnit test suites, that can be used to test the adaptation of a model or model instance, respectively. The test suites are initialised with a model or model instance that contains all the adapted concepts that shall be tested. The test suites then check if all required methods to retrieve `Types`, `Operations`, `Properties` for the variation point VP1 are implemented appropriately. The reflection mechanism provided by variation point VP2 is tested as well. These generic test suites helped us to ensure that all existing adaptations behave in the same expected manner and to easily detect wrong adaptations of some elements. Furthermore, these test suites can be used to ensure the absence of specific bugs in all adaptations by adding new test cases if such a bug is detected in one of the adaptations.

6 Related Work

In the following we will discuss alternative tools to parse, interpret, or compile OCL constraints and the means they provide to support variability in models and model instances:

- The *USE* tool [24] contributes an OCL simulator that can evaluate OCL constraints against model snapshots. It is bound to UML class, UML object and UML sequence diagrams and does not provide means for model or model instance adaptations. Nevertheless, a case study proofed that it is possible to create snapshots from Java runtime objects that can be evaluated with USE [25].
- The *OCLE* tool [26] interprets OCL constraints on UML models. Furthermore, it provides a compiler to generate a Java implementation from a constrained UML model and the according OCL constraints. Model adaptation is not supported. Although OCLE does not allow for real model instance adaptation, XML files can be treated as model instances by transforming them into UML object diagrams.

- The *MIP OCL2 Parser* [27] is a Java library for parsing OCL constraints provided by the Institute for Defense Analyses. Constraints are checked syntactically and semantically against a UML class diagram. To use the parser, one must provide a Java implementation of the abstract UML model expected by the parser. Thus, the MIP parser provides very limited means for model adaptation. Since MIP does not contribute an interpreter or compiler for constraints, model instance adaptation is not relevant.
- The OCL interpreter and compiler provided by the *Kent Modeling Framework (KMF)* supports model adaptation via a central *Bridge* model [13]. Both, the compiler and the interpreter depend on a Java-based representation of model instances. Thus, model instance adaptation is not supported.
- The *Epsilon Validation Language (EVL)* introduced in [18] is quite similar to OCL. It comes with an interpreter that can be used for various EMF-based languages. Thus, model adaptation is possible while model instances are bound to EMF.
- A standard OCL interpreter for EMF is provided by *MDT OCL* [6]. It is also tightly integrated with EMF and supports model adaptation for various EMF languages. The interpreter directly supports model instances represented in EMF. MDT OCL's architecture is highly extensible and could be adapted to other model instances using *Java Generics* [28]. However, we are not aware of any such adaptations.

This analysis of related work shows that variability at model level is considered useful and has already been implemented in various OCL tools. Supporting variability at the model instance level – as suggested in this paper – is a consequent continuation of our previous and other's related work.

7 Conclusion

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

The most important difference between Model Types and Model Instance Types is that Model Types abstract modelling concepts whereas Model Instance Types abstract reflection capabilities of different model instances.

8 Acknoledgements

We want to thank Tricia Balfe of NOMOS SOFTWARE for providing data for the XML case study and for continuous feedback during adaptation of the case study. Furthermore, we'd like to thank all people that are or were involved into the DresdenOCL project.

References

1. OMG: Object Constraint Language, Version 2.2. Object Management Group (OMG), Needham (February 2010)
2. OMG: Unified Modeling Language^{TM}, OMG Available Specification, Version 2.2. Object Management Group (OMG), Needham (February 2009)
3. Warmer, J., Kleppe, A.: The Object Constraint Language - Getting Your Models Ready for MDA. 2nd edn. Pearson Education Inc., Boston (2003)
4. Akehurst, D., Howells, W., McDonald-Maier, K.: UML/OCL - Detaching the Standard Library. In: OCLApps 2006: OCL for (Meta-) Models in Multiple Application Domains, MODELS 2006, Genova, Italy. (2006)
5. Loecher, S., Ocke, S.: A Metamodel-based OCL-compiler for UML and MOF. In: UML 2003-The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA. (2003)
6. Eclipse Model Development Tools. `http://www.eclipse.org/modeling/mdt/`
7. Amelunxen, C., Königs, A., Rötschke, T., Schürr, A.: Metamodeling with MOFLON. In: Applications of Graph Transformations with Industrial Relevance, Springer (2008) 573–574
8. Demuth, B., Wilke, C.: 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, Ufa State Aviation Technical University (2009)
9. Arnold, D.: C# Compiler Extension to Support the Object Constraint Language Version 2.0. In: Master Thesis, Carleton University, Ottawa, Ontario. (2004)
10. Heidenreich, F., Wende, C., Demuth, B.: A Framework for Generating Query Language Code from OCL Invariants. In Akehurst, D.H., Gogolla, M., Zschaler, S., eds.: Ocl4All - Modelling Systems with OCL. Volume 9 of ECAST., Technische Universität Berlin (2008)
11. Sakr, S., Gaafar, A.: Towards Complete Mapping between XML/XQuery and UML/OCL. In: Proceedings of the IADIS e-society 2004 conference (ES 2004), Avila, Spain. (2004)
12. Bräuer, M., Demuth, B.: Model-Level Integration of the OCL Standard Library Using a Pivot Model with Generics Support. In Akehurst, D.H., Gogolla, M., Zschaler, S., eds.: Ocl4All - Modelling Systems with OCL. Volume 9 of ECAST., Technische Universität Berlin (2008)
13. Akehurst, D., Patrascoiu, O.: Ocl 2.0-Implementing the Standard for Multiple Metamodels. In: OCL2.0 - "Industry standard or scientific playground?" - Proceedings of the UML'03 workshop, Citeseer (2003) 19–25
14. OMG: Meta-Object Facility (MOF^{TM}), Version 1.4. Object Management Group (OMG), Needham (April 2002)
15. Kurtev, I., Bézivin, J., Aksit, M.: Technological spaces: An initial appraisal. CoopIS, DOA **2002** (2002)
16. Smith, B.: Procedural Reflection in Programming Languages. PhD thesis, MIT (1982)
17. Dresden OCL Toolkit. `http://dresden-ocl.sourceforge.net/`
18. Kolovos, D., Paige, R., Polack, F.: Detecting and Repairing Inconsistencies across Heterogeneous Models. In: Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation, IEEE Computer Society (2008) 356–364

19. Aßmann, U.: Invasive Software Composition. 2nd edn. Springer Verlag, Heidelberg (2003)
20. ISO: Payments Standards - Initiation - UNIFI (ISO 20022) Message Definition Report. International Organization for Standardization (ISO), Geneva (October 2006)
21. EPC: SEPA Business-To-Business Direct Debit Scheme Customer-To-Bank Implementation Guidelines, Version 1.3. Number EPC131-08. European Payments Council (EPC), Brussels (October 2009)
22. Euro Banking Association (EBA). `https://www.abe-eba.eu/`
23. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. 2nd edn. Addison-Wesley Professional, Indianapolis (1995)
24. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based Specification Environment for Validating UML and OCL. Science of Computer Programming **69**(1-3) (2007) 27–34
25. Occello, A., Dery-Pinna, A.M., Riveill, M.: Validation and Verification of an UML/OCL Model with USE and B: Case Study and Lessons Learnt. In: Proceedings of the Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on Software Testing, Verification, and Validation (ICST), Lillehammer, Norway, IEEE Digital Library (April 2008) 113–120
26. OCLE2.0 - Object Constraint Language Environment. `http://lci.cs.ubbcluj.ro/ocle/`
27. MIP OCL Parser (MIP MDA Tools). `http://mda.cloudexp.com/`
28. Damus, C.W. MDT OCL Goes Generic - Introduction to OCL and Study of the Generic Metamodel and API. Slides of Presentation at the EclipseCon 2008 (2008)