Extending Variability for OCL Interpretation

Claas Wilke, Michael Thiele, and Christian Wende
{claas.wilke,michael.thiele,christian.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 modelling languages. This includes Domain Specific Languages (DSLs) and meta-modelling languages like 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 technical space that models are realised in. Current OCL interpreters do not support such variability as their implementation is typically bound to a specific technical space like Java, Ecore, or a 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 technical spaces and evaluate our approach in three case studies.

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 modelling and meta-modelling languages [4] like the Meta Object Facility (MOF) [5] or Ecore [6].

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], stored in database systems [10], or described with XML [11].

Several OCL tools support *variability at the model level* [12, 13], i.e., constraint definition on different types of models. Yet, *variability at the model instance level*, i.e., constraint evaluation on different types of model instances, is

not provided. For OCL compilers as presented in [5, 8–10] such variability is not possible, as the code generated from OCL constraints needs to be bound to the technical spaces used at model instance level. However, for OCL interpreters, we argue that a decoupling of the semantics evaluation from a concrete model instance type 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. We implemented such an infrastructure in *DresdenOCL* [14].

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 combinations of 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

Originally, OCL was designed as a constraint language for UML. Recent work showed that OCL can be applied to various other modelling and meta-modelling languages. Thus, an abstract description of OCL and its relation to modelling languages is sensible.

2.1 The Generic Three Layer Architecture

As discussed in [8], OCL evaluation always involves three adjacent layers of the *MOF four layer metadata architecture* [15]. This leads to the notion of a *generic three layer architecture* for OCL as depicted in Fig. 1 (a). At layer Mn+1, OCL is bound to a concrete modelling language that has to provide concepts like types, navigable properties, and possibly operations. This binding allows the definition of OCL constraints on models that are described by the 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 are often realised in a different *technological space* [16] than their model. Then, there are two model representations at the layer Mn (cf. Fig. 1 (a)): the original constrained *model* and a *model realisation* implementing the model in a specific technological space. The model realisation is typically derived by a transformation of the model [17]. This leads to the problem that the connection between the different model representations at the Mn layer is hidden inside the transformation and may not be accessible from the OCL infrastructure. An algorithm matching the elements of the model realisation and the constrained model can re-establish this connection.

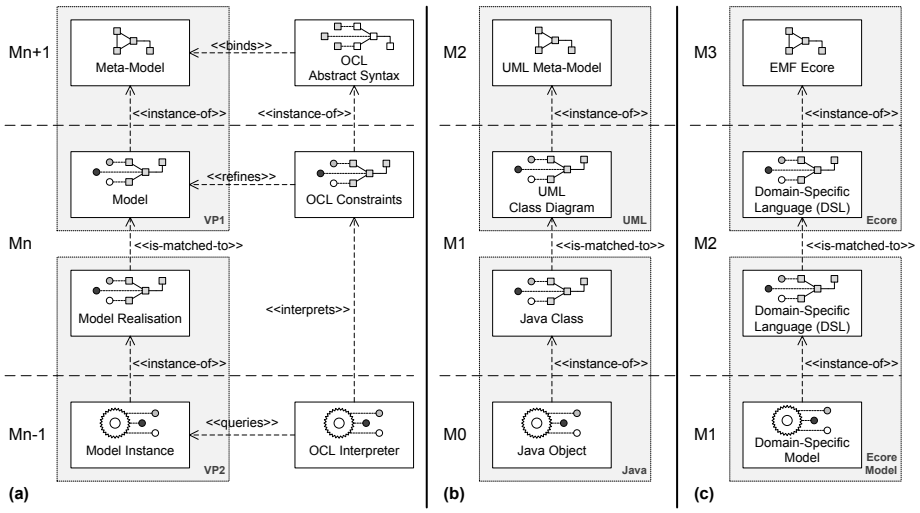

Fig. 1. The Generic Three Layer Architecture

Instances of this generic three layer architecture are shown in Fig. 1 (b) and (c). Fig. 1 (b) exemplifies the application of OCL to constrain UML class diagrams. At M2 (Mn+1) the OCL abstract syntax is bound to the UML meta-model. At M1 (Mn) constraints are defined against UML class diagrams. At this layer the original UML model is transformed to a Java-based model representation where UML classes correspond to Java classes. At M0 (Mn-1) the OCL constraints are evaluated against Java objects.

The example in Fig. 1 (c) demonstrates the application of OCL constraints to define well-formedness rules (WFRs) for DSLs built with EMF Ecore. The generic three layer architecture is lifted one layer: At M3 (Mn+1) the OCL abstract syntax is bound to the Ecore meta-meta-model. At M2 (Mn) a new DSL is defined using Ecore and OCL-based WFRs. Since EMF contributes a runtime infrastructure for models and model instances no transformation is required at M2. The model and the model representation are identical. At M1 (Mn-1) the WFRs are evaluated against models built with the defined DSL.

2.2 Variation Points of OCL Interpretation

With instantiation of our generic three layer architecture for various OCL applications found in literature, we identified two variation points. The first variation point (VP1) describes variability at Mn+1 and Mn w.r.t. the modelling language used to specify constrained models. The second variation point (VP2) relates to variability at Mn and Mn-1 t w.r.t. the technological space model instances are realised and validated in (cf. Fig. 1 (a)).

With the generic three layer architecture it is possible to vary several models and model instances independently. For example the same UML model can be

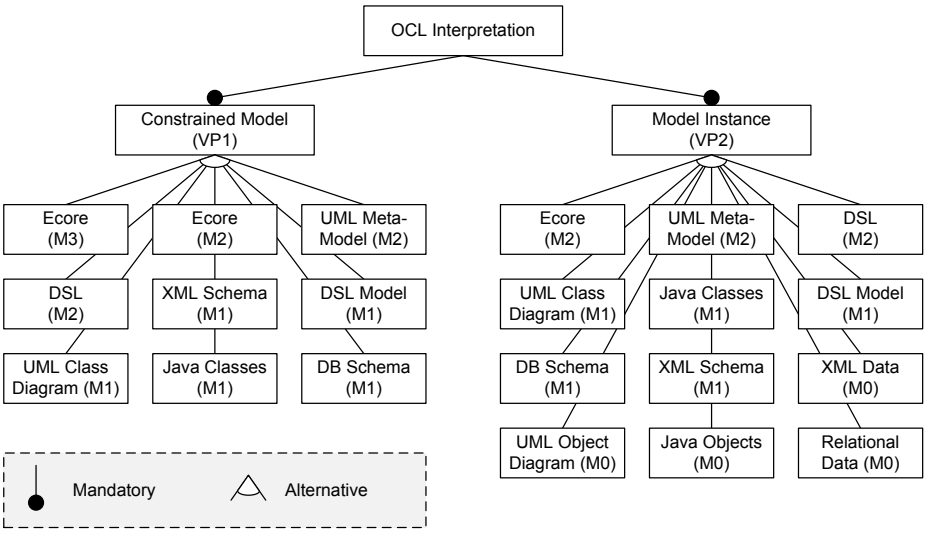

Fig. 2. Features of OCL Interpretation

combined with OCL evaluation on model instances realised in Java or a relational database, or the OCL evaluation on XML-based model instances can be used for models built with the UML or the Ecore meta-model. The feature model depicted in Fig. 2 documents the variation space of OCL applications found in literature. The only constraint a variant configuration of the generic architecture has to satisfy is that the model instance bound to VP2 is located exactly one meta-layer below the model bound to VP1.

Several OCL tools provide support for variation on VP1 [6, 13, 14, 18]. Yet, those tools do not address VP2 as their supported models require specific instances (typically both located in the same technological space). We argue that this tight coupling can be reduced to avoid the reimplementation of OCL interpreters for different technological spaces. In the following we contribute an implementation of the generic three layer architecture that supports VP1 and VP2.

3 Implementation

In this section we discuss the implementation of a generic adaptation architecture for DresdenOCL to realise the variation points identified in the previous section. First, we present *model adaptation* to address VP1. Afterwards, we discuss how *model instance adaptation* supports VP2.

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

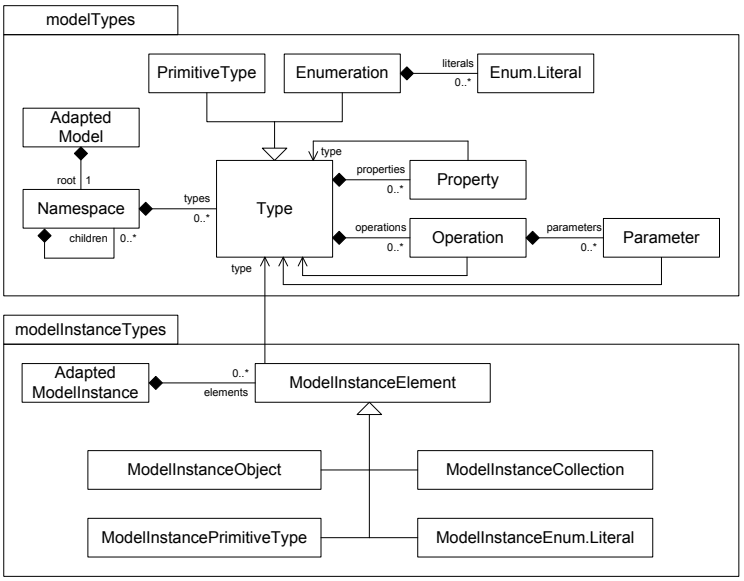

Fig. 3. Interfaces for model and model instance adaptation

and query models. These interfaces – called `modelTypes` (or `pivotModel`) [12] – define the basic concepts such as `Type`, `Property`, `Operation` and `Parameter` 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 OCL 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's meta-model to corresponding concepts of the `modelTypes`. E.g., the UML meta-model concept `UMLClass` is adapted to the `modelTypes` concept `Type`. Furthermore, the model adapter component has to create instances of these 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 in various technical spaces.

To provide means for model instance adaptation, we introduced the `modelInstanceTypes`. The `modelInstanceTypes` are a set of `ModelInstanceElements`

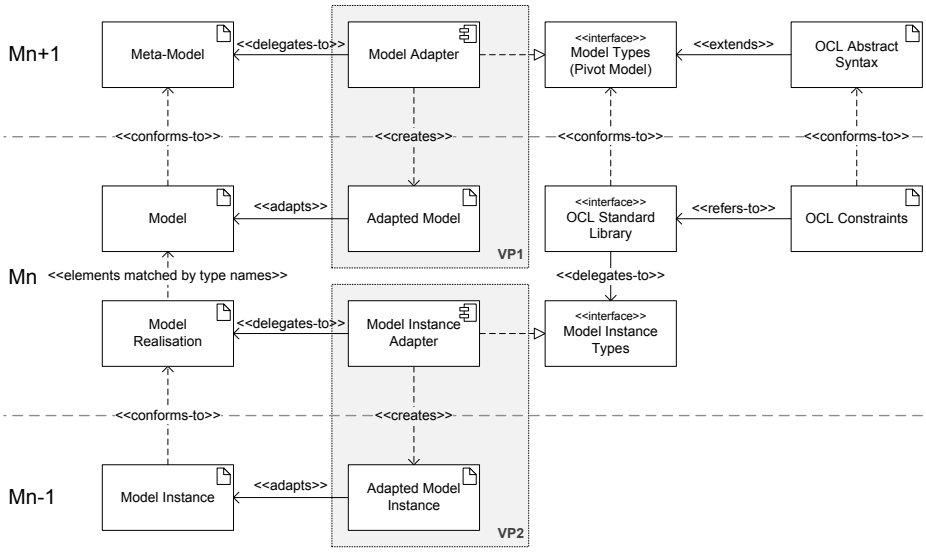

Fig. 4. The Generic Adaptation Architecture of DresdenOCL

representing instances of primitive types, collections and objects defined in the model (cf. Fig. 3). The most important difference between the `modelTypes` and `modelInstanceTypes` is that `modelTypes` abstract modelling concepts whereas `modelInstanceTypes` abstract reflection capabilities of different model instances (cf. Fig. 4, Mn and Mn-1 layer). During interpretation, the OCL 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 has to create `ModelInstanceElements` for the runtime objects of the adapted model instance on demand (cf. Fig. 4, Mn-1 layer). Adapted objects are cached to improve the performance and to avoid phenomena like *object schizophrenia* [19].

4 Case Studies

In this section we present three case studies to demonstrate the benefits of our generic adaptation architecture. The examples use different combinations of models and model instances located at different layers of the MOF four layer metadata architecture to illustrate the variability of our approach.

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]. This example was designed by Warmer and Kleppe

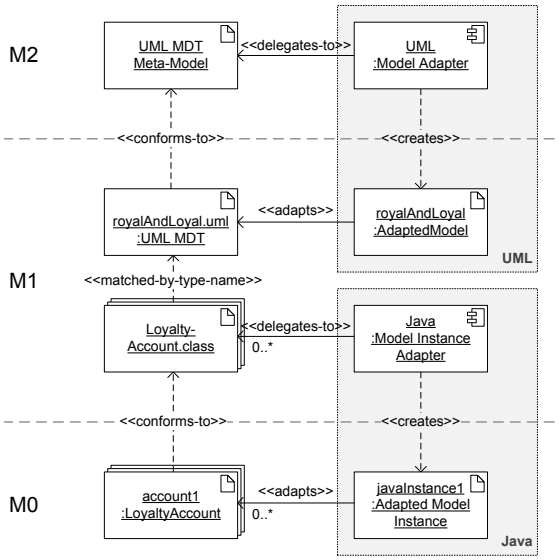

Fig. 5. Adapters used in the Royal and Loyal case study

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 UML model (VP1) built 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 *UML model adapter* component was implemented. It adapts the required concepts of the UML metamodel to the `modelTypes` 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 `modelInstanceTypes`. Thus, the objects of the royal and loyal Java implementation were adapted as a model instance in DresdenOCL at the M0 layer. Since the UML classes are transformed to Java classes, both are located at the M1 layer. Hence, the Java model realisation has to be matched with the UML model during interpretation.

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.

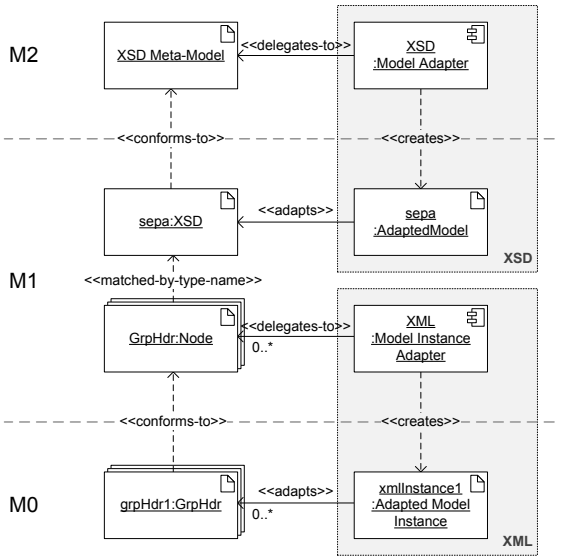

Fig. 6. Adapters used in the SEPA case study

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 to ensure the consistency of SEPA messages. We evaluated about 120 constraints that are provided with the online demo.[1]

The adapters required for the SEPA case study are shown in Fig. 6. To parse the SEPA constraints into DresdenOCL, the *XSD model adapter* component adapts required concepts of the XSD meta-model to the `modelTypes` of DresdenOCL at the M2 layer. Consequently, the SEPA XML schema was adapted as a model at the M1 layer. The *XML instance adapter* component adapts the XML model elements (mainly the class `org.w3c.dom.Node`) to the `modelInstanceTypes`. Thus, the nodes of the SEPA messages were adapted as model instances 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 demonstrates that our model instance adaptation allows DresdenOCL to transparently interpret constraints on XML files as well, since the OCL interpreter had not to be modified for the SEPA case study.

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

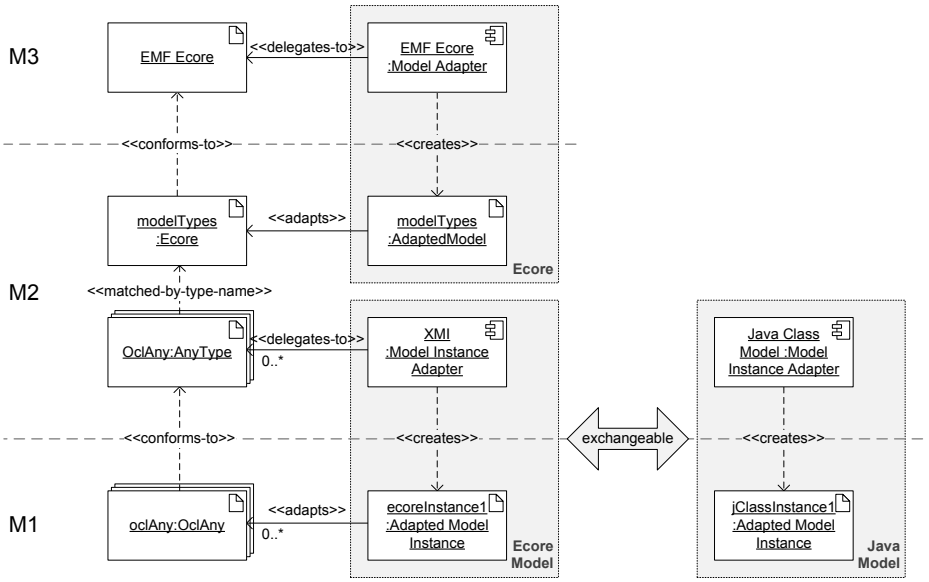

Fig. 7. Adapters used in the Standard Library case study

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 these 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 `modelTypes`, 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].

Although modelling the standard library leads to great flexibility, an implementation that provides its dynamic semantics is still required. This implementation is realised in Java. As there is no code generator for the `modelTypes`, the manual implementation 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 `modelTypes`, an *EMF Ecore model adapter* component was required to parse OCL constraints defined on the `modelTypes` (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 modelled standard library were evaluated for this instance.

This case study demonstrates that the implemented OCL interpreter is not only independent of specific model technical spaces, but can also use adapters for 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 VP1 including *WSDL* and *SQL-DDL* and further model instance adapters for VP2 including *C#* and *relational databases*.

5 Lessons Learnt

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

Type Matching As models and model instances can be located in different technical spaces, `ModelInstanceElements` must be matched to a model type when they are imported into DresdenOCL (cf. Fig. 1 (a)). Currently, this type matching is realised by a simple type name match (including the names of their enclosing namespaces whenever possible). E.g., the Java class `LoyaltyAccount` is matched to the UML class `LoyaltyAccount` in the royal and loyal case study. This matching algorithm can be rather complex and often information that could be used to improve the matching is hidden inside the adapters. E.g., when adapting an instance of an EMF Ecore model, one could use the *generator model* provided by Ecore to retrieve information used in the Ecore to Java transformation. We plan to improve this process 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 a model instance specific matcher that regards model transformation information whenever possible and ends by trying to simply match the type names as currently done.

Element Unwrapping Another problem when using adapters for `modelInstanceTypes` is the unwrapping mechanism of adapted elements when invoking operations on the `ModelInstanceElements`. E.g., to invoke an operation of an adapted Java object 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 `modelInstanceTypes` 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 they provide no operations.

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 creation of model adapter components. The code generator requires an annotated meta-model describing the relation of meta-model concepts to the `modelTypes` (e.g., the UML 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 `modelInstanceTypes`, 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 whether all required methods to retrieve `Types`, `Operations`, `Properties` for the variation point VP1 are implemented or whether the reflection mechanism provided by VP2 is supported appropriately. 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 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 for 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. In [28], a first approach to reuse OCL semantics at the model instance level for various model realisations was proposed. This approach is limited to model instances defined at the same meta-layer as their models and operation calls are not supported.
- 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* [29]. However, we are not aware of any such adaptations.

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

7 Conclusion

In this paper we presented a generic approach for OCL interpretation that addresses both model and model instance variability. Various OCL infrastructures support model variability, whereas – to the best of our knowledge – none of the existing OCL infrastructure supports complete model instance variability. Our

approach addresses this problem by abstracting from domain-specific concepts and by introducing well-defined interfaces for models and their instances. With our implementation of such a generic adaptation architecture, the same OCL interpreter was applied to three case studies that are located at different modelling layers and use different combinations of models and model instances. We avoided new implementations of the OCL standard library for various different technical spaces and hence contribute a reusable OCL interpreter.

For future work, we plan to improve our approach by addressing the issues mentioned in Sect. 5. We are interested in evaluating the performance impact of our adapter-based approach for OCL interpretation. Therefore, we plan a benchmark comparing our interpreter with other interpreters and compilers, and a continuation of our previous work [10] on extensible OCL compilation.

8 Acknowledgements

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 would like to thank all people that are or were involved in the DresdenOCL project.

References

1. OMG: Object Constraint Language, Version 2.2. Object Management Group (OMG), Needham (February 2010)
2. OMG: Unified Modeling Language™, 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 ECEASST., 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 ECEASST., 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. Dresden OCL Toolkit. `http://dresden-ocl.sourceforge.net/`
15. OMG: Meta-Object Facility (MOF^{TM}), Version 1.4. Object Management Group (OMG), Needham (April 2002)
16. Kurtev, I., Bézivin, J., Aksit, M.: Technological spaces: An initial appraisal. CoopIS, DOA (2002)
17. Eclipse Modeling Framework (EMF) Project. `http://www.eclipse.org/modeling/emf/`
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. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
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. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Towards Using OCL for Instance-Level Queries in Domain Specific Languages. Volume 5 of ECEASST., Technische Universität Berlin (2006)
29. Damus, C.W.: MDT OCL Goes Generic - Introduction to OCL and Study of the Generic Metamodel and API. In: EclipseCon 2008, Slides of the presentation. (2008)