# 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 constraint UML models to a constraint language that is applied to various object-oriented modelling languages including various Domain Specific Languages (DSLs) and metamodeling languages like the Meta Object Facility (MOF) or Ecore. Consequently, it is rather common provide variability in OCL parsers to work for different modelling languages. A second, variability dimension relates to the infrastructure that models are implemented and instantiated in. Current OCL interpreters do not support such variability but are typically bound to a specific implementation infrastructure, e.g. Ecore, Java, or a specific model repository. In this paper, we propose a generic adaptation architecture for OCL that hides meta-models, models and model instances behind well-defined interfaces. We present how the implementation of such architecture for DresdenOCL enables reuse of the same OCL interpreter for various model implementation and instantiation infrastructures. 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, MDSD, Modelling, Constraint Interpretation, Variability, Adaptation.

[terminology: artefact, element, type, concept, class. . . ]

## 1 Introduction

[Christian: REVISE INTRO!] 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]. In recent years, OCL has been applied to other modelling languages including meta-meta-models like the Meta Object Facility (MOF) or Ecore and various Domain Specific Languages (DSLs) [**?**] [Claas: Reference is missing in bib-File; Michael: I am still looking for a good reference - any ideas?].

For instance, OCL can be used to define business rules on a UML model that are checked for model instances including UML object model ~~element~~s,

Java objects or XML files. In another example, OCL can be used to define *Well-Formedness Rules (WFRs)* on an *XML Schema Definition (XSD)* that are evaluated for instances of the definition, e.g., XML files. Other instances of the XSD model are possible.

There are two variation points that can be identified. Firstly, OCL constraints can be defined on different models as long as the model has object-oriented modelling concepts like types and navigable properties. When integrating OCL with such a model, the concrete and abstract syntax of OCL is not affected. As OCL is statically typed, the abstract syntax only needs access to the types of the model it is defined on. Thus, a generic implementation of an OCL parser that relies on an abstraction of the model's underlying type system can be reused to parse OCL constraints on different types of models [3]. Secondly, model instances may vary specific kinds of models. As OCL constraints are defined on a model and evaluated on elements of the model's instances, the variability of different model and model instance combinations has to be considered by an OCL infrastructure as well.

Generally, two approaches exist to evaluate OCL constraints on instances of constraint models: the *Generative Approach* and the *Interpretative Approach* [4]. Using the generative approach, constraints are translated into queries or executable code and added to the model instance. This approach has some drawbacks. Firstly, the model instance might not offer means to express dynamic semantics with the implication that the constraints cannot be translated. Secondly, instrumentation of the model instance with the translated code might be not possible or far from trivial. Lastly, variations at model instance level can only be achieved with a completely new compilation. Using the interpretative approach, an OCL interpreter visits nodes of the constraint's abstract syntax and executes their appropriate semantics. Model instances are only ~~read~~requested for property values and operation results and thus neither need to have dynamic semantic nor instrumentation capabilities. Furthermore, since model instances are only read, the complete OCL semantics remains unaffected from the specifics of model instances and therefore can be reused.

Thus, when interpreting OCL constraints, an OCL infrastructure has to consider two variation points. Some OCL infrastructures allow the specification of OCL constraints on different types of models. However, in practice, OCL interpreters are typically bound to a specific type of model instances or to fixed combinations of models and model instances [references?] and hence do not address the second variation point to vary model instance types.

In this paper, we propose an architecture for an OCL infrastructure 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. We present an implementation of our approach for *DresdenOCL* [5]. *DresdenOCL* is able to parse OCL constraints on different types of models such as UML, EMF Ecore, Java and XML Schema and interpret them for different model instances such as Java objects, XML files or UML diagrams.

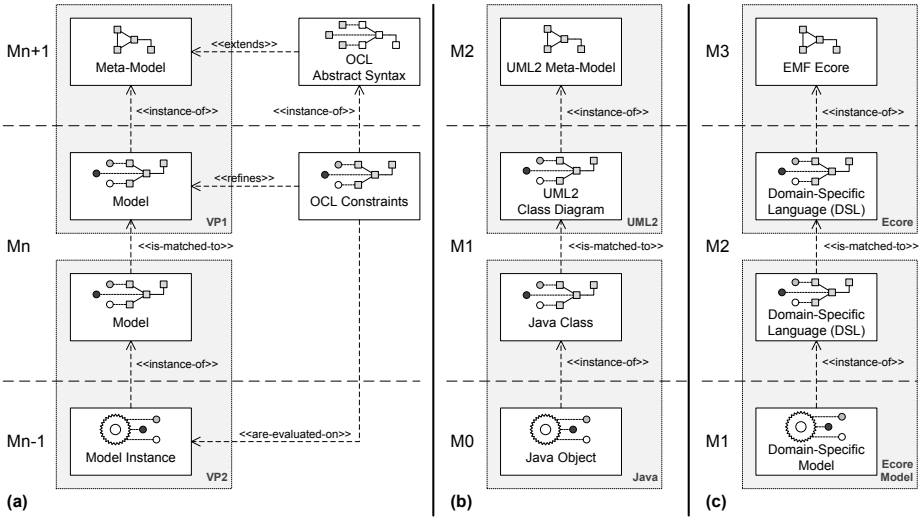


**Fig. 1.** Variation Points in OCL Parsing and Interpretation

[TODO The paper is structured as follows...]

## 2 The Generic Architecture of Dresden OCL2 for Eclipse

[try to be in-line with the terminology used in the abstract]

[First motivate variation, adapt section heading accordingly] [explain required variability, feature model approach, resulting from related work on OCL applications, differentiate that vp1 and vp2 may vary independently in some cases, not in others..] [explain theory of ocl metalevel relativity :) ] [introduce variation points using Fig.2]

[Fig 1. add headings to subfigures: a) Variation points in OCL Application b) Applciation of OCL in UML modeling c) Application of OCL in Ecore metamodeling ] [Fig 1. can/should we change order to b), c) , a)? ]

[Michael: still too much caption text; in (b), is the UML2 meta-model really MOF?, I thought MOF is on M3] The *Generic Three Layer Meta data Architecture*. (a): 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 instantiates the constrained model. The generic architecture can be parametrized at two different variation points. Different models and their meta-models can be bound to VP1, different model instances can be bound to VP2. (b): UML-Example for layers M2, M1 and M0. VP1 is bound to the UML meta-model (MOF), VP2 is bound to Java. (c): EMF-Example for layers M3, M2 and M1. VP1 is bound to EMF Ecore, VP2 is bound to Ecore-based models.

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*. [Is

that important to understand the paper?; Michael: agreed, just refer to it, but leave explaination]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.

Fig. 1 depicts the *Generic Three Layer Meta-Data Architecture* aligning OCL constraints with the MOF layers. ~~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 using their abstract syntax, i.e., the OCL meta-model. They are evaluated for runtime objects at the model instance layer. In order to ~~navigate through the model or to call operations on~~define constraints on models OCL needs to navigate on model elements. To bind OCL constraints on the navigation structure in the model they are defined on, the abstract syntax of OCL [this is true for UML, but sounds quite technical]extends the model's meta-model.

[we need to incorporate the motivation for implementation language variation here. Reviewers need to understand the problem to understand our solution.]

[This section does not clarify the contribution of THIS paper. We should sell the instance adaptation as a improvement of previous DresdenOCL versions

- motivation: different model implementations are useful
- context: identify variation points in fig 1, modeling langauge, implementation language
- problem: neither dresdenOCL nor others are able to cope with differnt model implementations
- solution: we suggest additional adaptation interface
- feasibility: we contribute implementation of adaptation approach in Dresden OCL

]

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 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.

## 3 Implementation

In this section we discuss the implementation of a *Generic Adaptation Architecture* to realize the variation points identified in the previous section. First, we present *Model Adaptation* to address VP1. Afterwards, we dicsuss how *Model Instance Adaptation* realises the second variation point (VP2).

### 3.1 Model Adaptation

To enable the definition of OCL constraints for various modelling languages, DresdenOCL provides a set of common interfaces abstracting structures that are required to navigate and query object-oriented models. These interfaces –

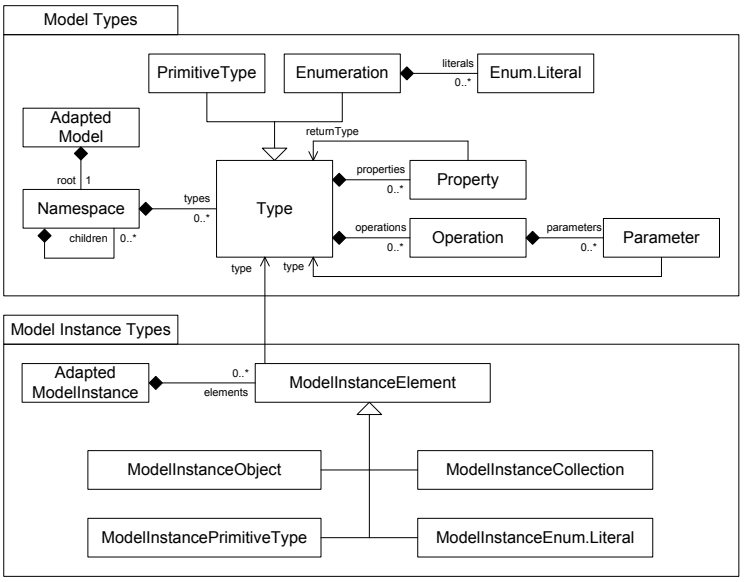


**Fig. 2.** Interfaces for model and model instance adaptation

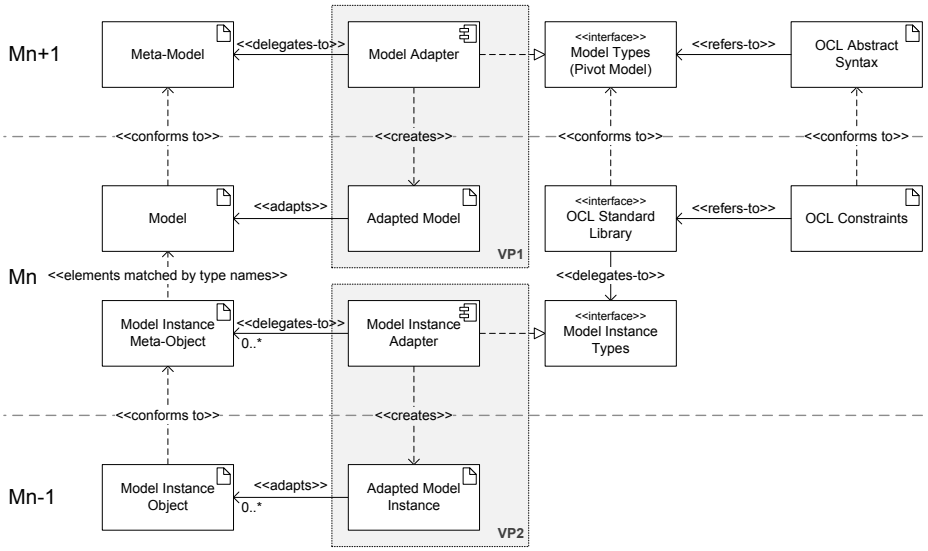


**Fig. 3.** The *Generic Adaptation Architecture* of DresdenOCL

called *Model Types* (or *Pivot Model*) [3] – standardize the basic concepts such as types, properties, operations and parameters that bind OCL constraints to a concrete modelling language (cf. Fig. 2). DresdenOCL only uses these concepts

to parse and statically analyse OCL constraints, e.g., the OCL2 parser invokes the operation `getType()` to access the `Type` of an `Operation` or `Property`.

For every ~~meta~~-model that shall be connected with DresdenOCL, a *Model Adapter* component has to be implemented (cf. Fig. 3, Mn+1 layer). It contains individual adapters that map concepts of the modelling language 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 provide a factory to create adapters on demand resulting in an *Adapted Model* (cf. Fig. 3, Mn+1 and 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 using OCL.

### 3.2 Model Instance Adaptation

To realise variation point (VP2), our Generic Adaptation Architecture re-applies the adaptation pattern for model instance adaptation. Similar to the adaptation of different models, we hide model instances behind interfaces. This enables the reuse of the same OCL interpreter for the dynamic evaluation of OCL constraints on model instance in various implementation infrastructures.

[What is the difference between a model instance type and a modelInstanceElement? Confusing. Revise and check for overlaps with previous sect.] To provide means for model instance adaptation, we introduced the *Model Instance Types*. [Michael: I don't get the next sentence] The model instance types are different interfaces for instances of standard types in OCL such as primitive types, collections and objects (cf. Fig. 2). All these interfaces inherit a common interface `ModelInstanceElement`. The most important difference between the model types and the model instance types is that `ModelInstanceElements` provide a *Reflection* [6] mechanism whereas model type elements only allow to reason on relationship between concepts of the same meta-level. During interpretation, the OCL2 interpreter uses these reflection mechanisms to retrieve the `Type` of a `ModelInstanceElement`, 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. 3, Mn layer). The model instance adapter component contains adapters that map elements of a concrete model instance to model instance types and has to provide a factory that creates *Model Object Adapters* for the runtime objects of the adapted model instance (cf. Fig. 3, Mn-1 layer). Like the model adapter, the model instance adapter also creates adapters for objects on demand. Adapted objects are cached to improve the performance and to avoid phenomena like *Object Schizophrenia* [7]. [Claas: Reference Okay? Christian: Didn't florian send the original source? Always use the most concrete you know]

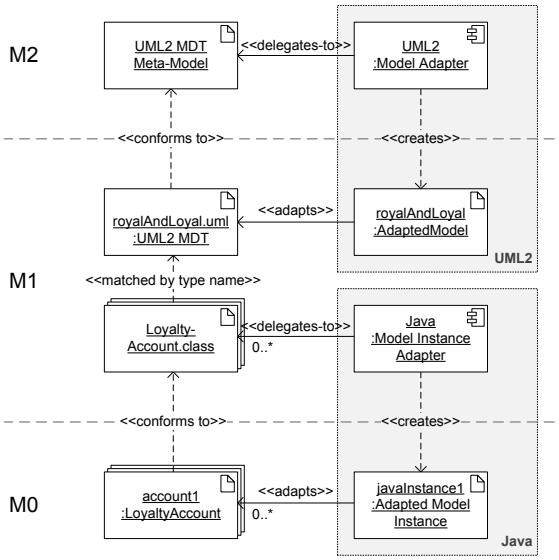


**Fig. 4.** Adapters used in the Royals and Loyals case study

## 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 [8]. The case study was originally 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 specify the royal and loyal system using UML2 (model) of the Eclipse Model Development Tools (MDT) project [9]. The model was implemented and instantiated in Java (model instance). Consequently constraints were evaluated on Java objects.

The adpaters required for the royal and loyal case study are shown in Fig. 4. To parse the Royal and Loyal constraints in DresdenOCL, a *UML2 Model Adapter* component was implemented, that adapts the required meta-model elements from the MDT UML2 meta-model to the model types of DresdenOCL at the M2 layer and creates instances of these adapters for all UML models loaded into DresdenOCL. Consequently, the royal and loyal class diagram was adapted as a model at the M1 layer. [Michael: The next sentence is kind of confusing, as we are on model instance layer and speak of Java meta-models... Isn't java.lang.reflect rather the model of Java?] For the Java implementation, a *Java Model Instance*

*Adapter* component was implemented, that adapts the Java meta-model elements (classes of the package `java.lang.reflect`) to the model instance types and creates instances of these adapters for all Java objects loaded into DresdenOCL. Consequently, the objects of the royal and loyal Java implementation were adapted as `ModelInstanceElements` in DresdenOCL at the M0 layer. Both, classes from the class diagram and from the Java implementation are located at the M1 layer because the Java classes are only other representations of the classes described by the UML class diagram!

The Royal and Loyal case study demonstrates that our generic adapter 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 interpret OCL consistency constraints defined on an XML schema (model) for an XML documents (model instance) 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)* [10–12]. 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 constrain the consistency of these SEPA messages. We implemented an XSD model adapter and an XML model instance adapter for DresdenOCL to evaluate these constraints on the XSD and XML files that are provided with an online demo of the Nomos service for *Pain.008.001.01* SEPA messages.[c0] About 120 constrains that are provided by the demo were used to test our implementation.

The adaptater required for the SEPA case study are shown in Fig. 5. To parse the SEPA constraints in DresdenOCL, an *XSD Model Adapter* component was implemented, that adapts the required meta-model elements from the XSD meta-model to the model types of DresdenOCL at the M2 layer and creates instances of these adapters for all XML schemas loaded into DresdenOCL. Consequently, the *Pain.008.001.01* XML schema was adapted as a model at the M1 layer. [Michael: again: model instance, but speak of XML meta-model] For the XML documents, an *XML Instance Adapter* component was implemented, that adapts the XML meta-model elements (mainly the class `org.w3c.dom.Node`) to the model instance types and creates instances of these adapters for all XML node instances loaded into DresdenOCL. Consequently, the nodes of the Pain.008.001.01 XML messages were adapted as `ModelInstanceElements` in DresdenOCL at the M0 layer.

These constraints of the Nomos service demo were evaluated for three different XML files and the results have been successfully compared with the results of the Nomos service demo. This shows that our model implementation adaptation

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

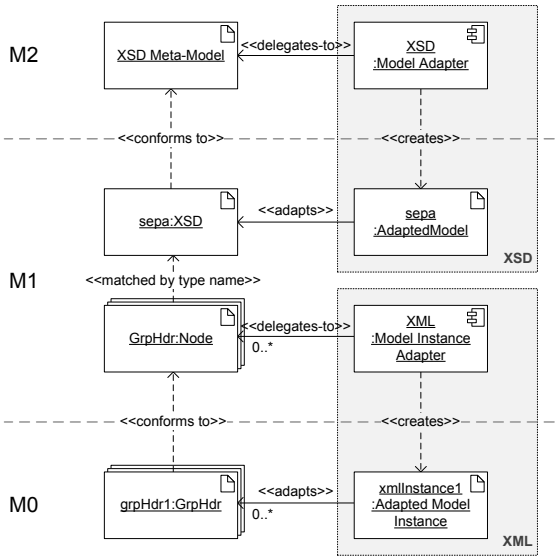


**Fig. 5.** Adapters used in the SEPA case study

allows DresdenOCL to transparantly 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 predefined OCL types is reduced to a simple model import while the model can conveniently be queried, validated or altered [3]. 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.

Although modelling the standard library leads to great flexibility, the standard library still needs an implementation that provides its dynamic semantics. This implementation is provided through Java code. As the *Model Types Editor* of Dresden OCL does not support code generation, this can lead to inconsistencies between the modelled standard library and the according Java interfaces. We propose to use OCL to check that all modelled types have an equivalent Java interface and all modelled operations are also present in the Java interface. Furthermore, this approach allows us to check whether the standard library supports all operations that are defined in the current OCL standard.

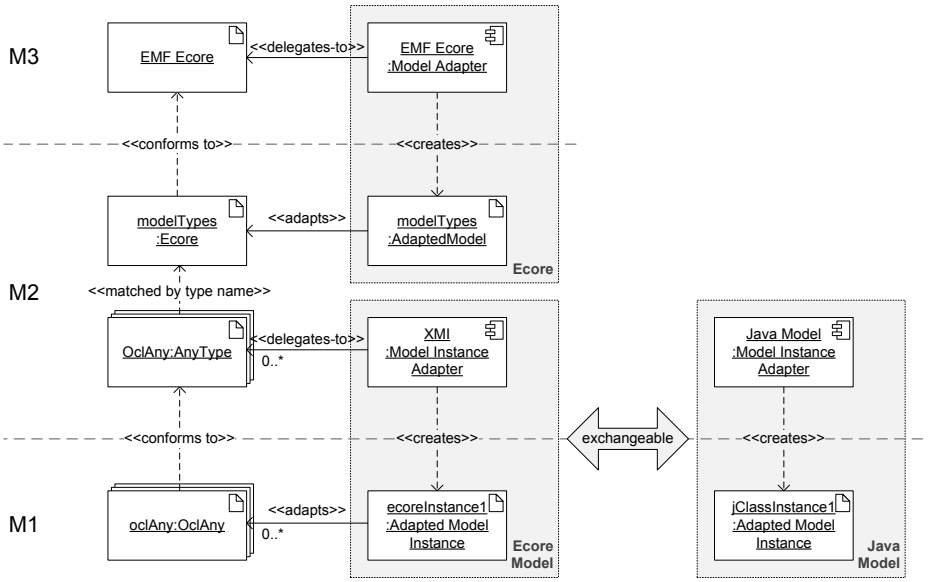


**Fig. 6.** [Michael: I know that this is not correct, but we should replace essentialOcl with model types for consistency] In the standard library case study, VP1 is bound to the *EMF Ecore Model Adapter*. Consequently, the EssentialOcl ecore model is adapted as a model at M2. VP2 is bound to the *XMI Model Instance Adapter* and thus, each element of the modelled Standard Library instance is adapted as a model instance object at M1.

The required adaptations for the standard library case study are shown in Fig. 6. To parse the WFRs in DresdenOCL, an *EMF Ecore Model Adapter* component was implemented, that adapts the required meta-model elements from the ecore meta-model (EMOF) to the model types of DresdenOCL at the M3 layer and creates instances of these adapters for all ecore-based meta-models loaded into DresdenOCL. Consequently, the [Michael: Model Types?] *EssentialOCL* meta-model was adapted as a model at the M2 layer. For the standard library instance, an *XMI Model Instance Adapter* component was implemented, that adapts the model istances of ecore-based meta-models (stored as XMI-files) to the model instance types and creates instances of these adapters for all elements of these models loaded into DresdenOCL. Consequently, the elements of the standard library were adapted as `ModelInstanceElements` in DresdenOCL at the M1 layer. Thus, we are able to evaluate the WFRs for the standard library model and can prove the correct structure of it according to the current OCL specification. [Claas: Emphasize the model layers used for interpretation here. Big difference to case studies 1 and 2!]

[Claas: Do we really have to mention this lack. Lets talk about the interpretation results instead.] In order to check for inconsistencies with the Java implementation, the Java interfaces for predefined OCL types have to be loaded with the

*Java Class Model Instance Adapter*. Then, the same WFRs used for the standard library model before can be checked for this instance. Unfortunately, the *Java Class Model Instance Adapter* does not exist yet, but will be implemented in the near future.

### 4.4 Future Case Studies

[Michael: We claim that we have independent model and model instance and now we propose a tight coupling between those?] 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).

## 5 Challenges ~~in Generic~~of a Variable OCL Infrastructure~~s~~

[Michael: Is it a "generic adapter architecture" or do we have another name including "variability"?] In this section we highlight some problems that occurred during the design and implementation of our generic adapter architecture for an OCL infrastructure. We also present some solutions to these challenges.

*Type Matching* As ~~mentioned above,~~ models and model implementations are only loosely coupled~~. M~~, model instance elements must be matched to a model type when they are imported into DresdenOCL. Currently, this type matching is realised by ~~simply matching the names of the types~~a simple type name match (including the names of their enclosing namespaces where possible). E.g., the Java class `LoyaltyAccount` is matched to the class `LoyaltyAccount` in the royal and loyal class diagram. Often, this solution is error prone [Michael: this sounds a little bit exaggerated] and further information is hidden behind 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 reflection mechanisms provided by Ecore to retrieve the types of EObjects. We plan to improve our matching that is currently scattered over multiple classes of each model instance adapter component ~~and~~ by introducing type matching strategies that can be implemented using the *Chain of Responsibility* pattern [13]. 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 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()` on a `collection` that returns an `Integer` instance). Thus, a model instance adapter has to provide operations to reconvert primitive types and collections to elements of the adapted model instances. In some cases this can become rather complicated because 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.

*Automated Adapter Creation* ~~Adapting different types of meta-models, models, and model instances enables the reuse of DrescenOCL for various combinations of modelling and implementation languages. Nevertheless, the~~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 meta-models to the model type concepts. The code generator requires an annotated meta-model describing the relation of meta-model concepts to the model type concepts. E.g., the UML2 meta-class `Classifier` is annotated as a `Type` concept. The code generator generates the skeleton code for all required adapters and thus avoids manual implementation of these adapters. 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 initialized 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` etc. are implemented 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 some elements. Furthermore, these test suites can be used to ensure the absence of specific bugs in all adaptations if such a bug is detected in one of the adaptations by adding new test cases.

## 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 [14] 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.
- The *OCLE* tool [15] 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* [16] 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 [17]. 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* [9]. It is also tightly integrated with EMF and supports model adaptation for various EMF languages. The interpreter directly supports model instances represented in EMF. Its architecture is highly extensible and could be adapted to other implementation infrastructures using *Java Generics* [19]. 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, thus, 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

## 8 Acknoledgements

We want to thank Tricia Balfe of Nomos Software for providing their 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.

[The bib file needs to be checked for lower and upper case, article titles should be enclosed in curly brackets...]

## 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. 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)
4. 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)
5. Dresden OCL Toolkit. `http://dresden-ocl.sourceforge.net/`
6. Maes, P.: Concepts and Experiments in Computational Reflection. In: Proceedings of OOPSLA'87. Volume 22 of ACM SIGPLAN Notices., New York, ACM Press (October 1987) 147–155
7. Aßmann, U.: Invasive Software Composition. 2nd edn. Springer Verlag, Heidelberg (2003)
8. Warmer, J., Kleppe, A.: The Object Constraint Language - Getting Your Models Ready for MDA. 2nd edn. Pearson Education Inc., Boston (2003)
9. Eclipse Model Development Tools. `http://www.eclipse.org/modeling/mdt/`
10. ISO: Payments Standards - Initiation - UNIFI (ISO 20022) Message Definition Report. International Organization for Standardization (ISO), Geneva (October 2006)
11. 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)
12. Euro Banking Association (EBA). `https://www.abe-eba.eu/`
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. 2nd edn. Addison-Wesley Professional, Indianapolis (1995)
14. 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
15. OCLE2.0 - Object Constraint Language Environment. `http://lci.cs.ubbcluj.ro/ocle/`
16. MIP OCL Parser (MIP MDA Tools). `http://mda.cloudexp.com/`
17. 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

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. 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)