Model and Object Verification by Using Dresden OCL

Birgit Demuth
Department of Computer Science
Technische Universität Dresden
Dresden, Germany
e-mail: birgit.demuth@tu-dresden.de

Claas Wilke
Department of Computer Science
Technische Universität Dresden
Dresden, Germany
e-mail: info@claaswilke.de

Abstract[1]

The Object Constraint Language (OCL) is a formal language standardized by the OMG (Object Management Group) that allows the specification of constraints on MOF- (Meta Object Facility) or EMF (Eclipse Modeling Framework) Ecore-based models. After about ten years of research and OCL prototyping in the area of using formal methods in practical software engineering, OCL is appreciated by the industry and tool vendors. An often used OCL library is the Dresden OCL Toolkit. OCL can be applied both on the meta-model (M2) and on the model layer (M1).

In this paper we present use cases for OCL in the context of the Dresden OCL Toolkit. We show how the user is able to specify precise semantics both on the meta-model and the model layer by OCL, and how these OCL constraints can be verified on models respectively on objects.

1. Introduction

The Object Constraint Language (OCL) is part of the Unified Modeling Language (UML) which is standardized by the Object Management Group (OMG) [1, 2]. OCL allows the specification of constraints on MOF- (Meta Object Facility) [3] or EMF (Eclipse Modeling Framework) [4] Ecore-based models. Since the release of OCL 2.0 in 2004, OCL can be regarded as both, a query and a constraint language. Besides the definition of invariants, and pre- and postconditions, OCL can be used to enrich models with new fields, operations, and derived classes. OCL can be used to enrich models at different layers of the *MOF Four Layer Metadata Architecture* [3].

On the meta-model layer, OCL is mostly used to specify well-formedness rules (WFR) that must be hold for models. At first, OCL was used for the Unified Modeling Language (UML). However, today OCL is also a recognized technique to specify the semantics of other MOF- or Ecore-based meta-models and domain specific languages (DSLs). On the model layer, OCL often helps to specify constraints (invariants as well as pre- and postconditions) on (business) objects, particularly if the power of UML concepts is not enough to express precise semantics.

The Dresden OCL Toolkit [5] provides a set of tools which enable the developers of UML tools to extend their tools with OCL support. The recent version of the toolkit provides an OCL2 Parser, an OCL2 Interpreter, and an OCL-to-Java Code Generator. Because the toolkit's architecture is based on a *pivot model*, the toolkit can be adapted to different meta-models and DSLs. Thus, the toolkit can be used to load, parse, and verify OCL constraints on different layers of the MOF Four Layer Architecture [3]. This paper will present a set of use cases which illustrate the wide-spread application of OCL and the Dresden OCL Toolkit.

The remainder of this paper is structured as follows: In Section 2, we present the Dresden OCL Toolkit and its recent version, Dresden OCL2 for Eclipse. A short introduction to the pivot model will be given. In Section 3, we present the two different approaches to verify OCL constraints, the *interpretative* and the *generative approach*. In Section 4, we present different use cases of the Dresden OCL Toolkit based on these approaches. The last section concludes the work.

2. The Dresden OCL Toolkit

The Dresden OCL Toolkit [5] is one of the software projects of the *Software Technology Group* at the Technische Universität Dresden, Germany.

2.1 Development History

The intention of the development of the Dresden OCL Toolkit was to provide a library provided with a set of OCL tools which can be reused by UML tool builders to extend their tools with OCL support. The toolkit has been designed for openness and modularity and is provided as open source at the project website [5]. It has been developed and evolved during many student theses since 1998. Today, the toolkit is one of the major software projects at the Software Technology Group, and three different versions of the toolkit have already been released.

Proceedings of the Russian-German Workshop "Innovation Information Technologies: theory and practice", July 25-31, Ufa, Russia, 2009

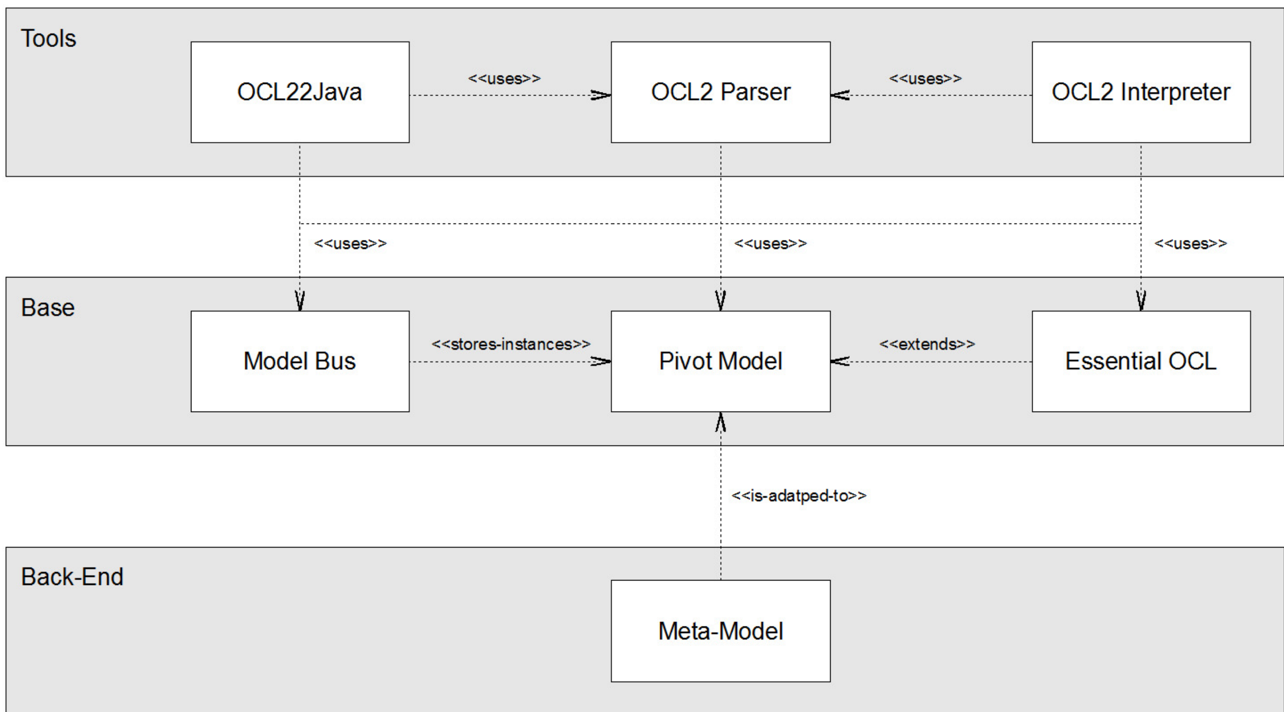

Figure 1: The architecture of Dresden OCL2 for Eclipse

The first and second version of the toolkit were released in 1999 and 2005. Both versions supported syntax and type checking of OCL constraints defined on UML class models, Java and SQL code generation, as well as Java code instrumentation [6].

In 2007, the toolkit was rewritten and released as *Dresden OCL2 for Eclipse*. The implementation of a *pivot model* as an intermediate meta-model allows alignment of meta-models of arbitrary domain-specific modeling languages (DSLs). The pivot model made the recent version of the toolkit independent from specific repositories and meta-models [7]. Today, adaptations to the UML2 meta-model of the *Eclipse Model Development Tools Project* [8] and to the *Eclipse Modeling Framework (EMF) Ecore* meta-model [4] are supported. Via the support of the UML2 meta-model, models created with the case tools *TopCased* [9], *MagicDraw® UML* [10], and *Visual Paradigm* [11] can be imported (via XMI export). Dresden OCL2 for Eclipse currently provides an OCL2 Parser to load and verify OCL constraints, an OCL2 Interpreter, and an OCL-to-Java Code Generator. Figure 3 shows a screenshot of Dresden OCL2 for Eclipse, showing the *Model* and *Model Instance Browser*, and the OCL2 Interpreter.

2.2 The Architecture of the Toolkit

The recent toolkit has been developed as a set of Eclipse/OSGi plug-ins. The architecture of Dresden OCL2 for Eclipse is shown in Figure 1. The architecture can be separated into three layers: The *back-end*, the *base,* and the *tools* layer.

The back-end layer represents the repository and the meta-model which can easily be exchanged because all other packages of the toolkit communicate with the pivot model instead of the meta-model. The pivot model delegates all requests to the employed meta-model. A common meta-model is the UML2 meta-model of the *Eclipse Model Development Tools Project* [7].

The second layer is the toolkit's base layer which contains the *Pivot Model*, *Essential OCL,* and the *Model Bus*. The use of the pivot model was mentioned before. The package Essential OCL implements the OCL Standard Library by extending the pivot model. The model bus loads, manages, and provides access to meta-models, models, and model instances the user wants to work with.

The third layer contains all tools that use the packages of the second layer to load, verify, and interpret OCL constraints. The layer contains the OCL2 Parser for syntax analysis and type checking of OCL constraints, the OCL2 Interpreter which can be used to interpret OCL constraints on meta-model or model instances, and the OCL22Java Code Generator which transforms OCL constraints into aspect-oriented AspectJ [12] code. Note that the OCL2 Parser is located in the tools layer, but that the other tools are not fully independent of the OCL2 Parser. Syntax and type checking is required for both, interpretation and code generation.

3 Interpretation vs. Generation

Generally, two different approaches exist to verify OCL constraints: an *interpretative* and a *generative approach*.

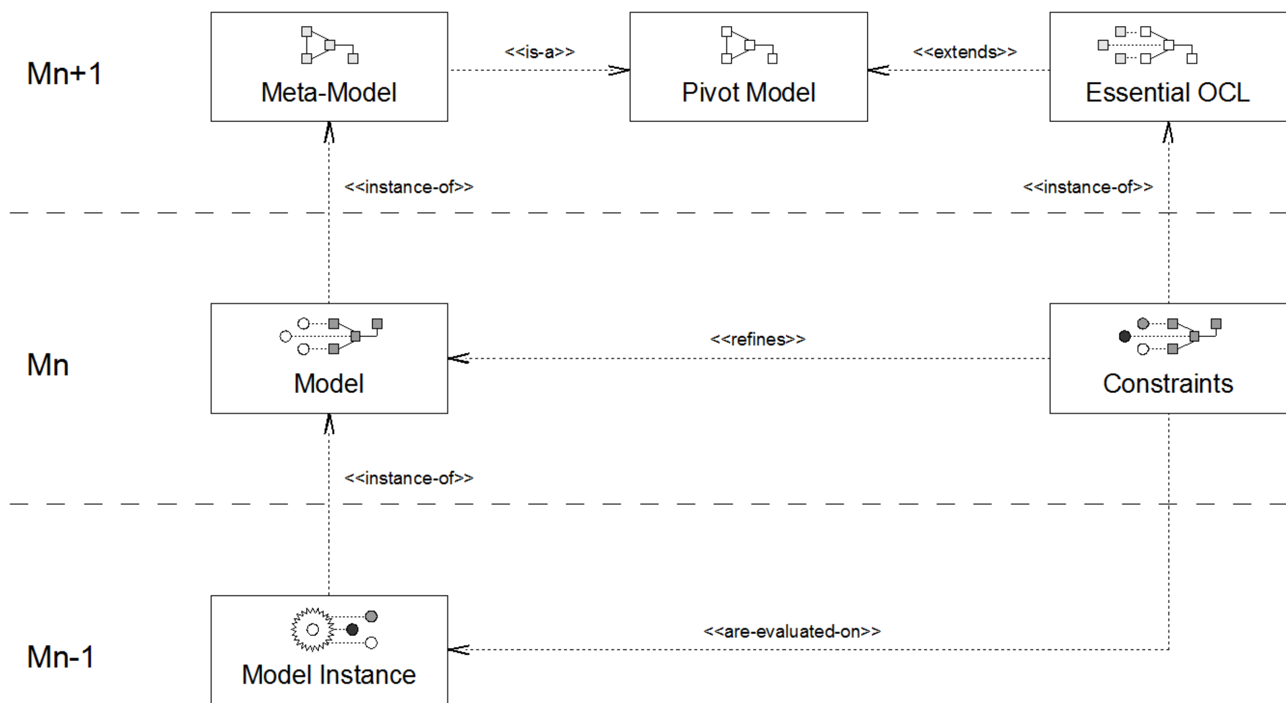

Figure 2: The Generic Three Layer Metadata Architecture: The relationships between meta-models, models and model instances in the scope of an OCL specification.

The interpretative approach verifies constraints by interpreting them on a model and its objects. The generative approach instead generates code or queries which can be executed to verify the constraints after generation. Both approaches are explained more detailed in this section. Additionally, this section explains which resources are required to describe models and specify constraints for verification.

3.1 Modeling OCL

The Object Constraint Language [1] is a language which always depends on another modeling language (usually the UML [2]). Without another language used for modeling, it does not make any sense to define constraints because OCL is used for constraint specification but not for modeling itself. Thus, besides OCL, a modeling language is required to define a model on which OCL constraints shall be specified.

Each modeling language is defined in another language, its *meta-modeling language*. For example, the Unified Modeling Language is defined using the *Meta Object Facility (MOF)* [3], the standardized meta-meta language of the OMG. The MOF is used to describe the UML meta-model that can be used to model UML models. Generally spoken, each model requires a meta-model that is used to describe the model. The model can be instantiated by model instances (for example an UML class diagram could be instantiated by an UML object diagram). The model can be enriched with OCL constraints that are defined on the model (using an OCL meta-model) and can then be verified for model instances of the model.

The OMG introduced the *MOF Four Layer Metadata Architecture* [3] which is used to arrange and structure the meta-model, the model, and its model instances into a layered architecture. Generally, four layers exist, the *meta-meta-model layer (M3)*, *the meta-model layer (M2)*, the *model layer (M1)*, and the *model instance layer (M0)*. OCL constraints can be defined on both, meta-models and models to verify models or model instances. Thus, the four layer metadata architecture can be generalized to a *Generic Three Layer Metadata Architecture* in the scope of an OCL definition (see Figure 2). On the *Mn+1 layer* lies the meta-model that is used to define the model that shall be constrained. The used meta-model, or DSL, has to be adapted to the pivot model. The Dresden OCL Toolkit provides a utility framework for easy pivot-model adapter generation. On the *Mn layer* lies the model which is an instance of the meta-model that is enriched by the specification of OCL constraints. Finally, on the *Mn-1 layer* lies the model instance on which the OCL constraints shall be verified. Please note, that in the context of such a generic layer architecture, a model instance can be both a model (like an UML class diagram) or an object (like a Java object). Thus, Dresden OCL2 for Eclipse can be used for both model and object verification, depending on whether a meta-meta-model or a meta-model is adapted to the pivot model.

3.2 The Interpretative Approach

A first approach for OCL verification is the *interpretative approach*. The interpretative approach uses an interpreter to interpret constraints on a model instance (Mn-1) by working on its model (Mn). The interpretative approach includes the following steps:

1. A model is described (Mn) using a meta-model adapted to the pivot model (Mn+1).
2. The model is enriched with OCL constraints which are defined on the types and operations defined in the model (Mn). We assume that an OCL specification includes the syntax and type checking process provided by the OCL2 Parser.
3. A model instance for which the OCL constraints shall be verified must be defined or generated (Mn-1).
4. The OCL2 Interpreter interprets the constraints defined on the model for the model instance, working on the model and its objects (Mn and Mn-1).

In contrast to the generative approach (explained in the following), the interpretative approach includes the verification of model instances. The result of the interpretative approach is a set of interpretation results, normally Boolean values like `true` or `false`.

3.3 The Generative Approach

A second approach is the *generative approach*. The generative approach uses a code generator or a model transformation framework to generate a new model and queries or code which can be executed for constraint verification. The generative approach includes the following steps:

1. A model is described (Mn) using a meta-model adapted to the pivot model (Mn+1).
2. The model is enriched with OCL constraints which are defined on the types and operations defined in the model (Mn), including syntax and type checking by the OCL2 Parser.
3. A new model and queries or code are generated by using templates or transformation rules defined on the meta-model elements for the model and its constraints (Mn+1 and Mn).

The generated queries, or code, can be executed to verify the specified OCL constraints. Note, that the verification of constraints is not part of the generative approach but has to be initialized externally. Depending on the form of model, query, or code which is generated, the constraints are verified at Mn-1 (by code execution), or at Mn-1 with the use of model information at Mn (by querying on model instances).

4 Use Cases of Dresden OCL

This section presents several OCL use cases supported by the Dresden OCL Toolkit. Analogous to the two approaches to verify OCL constraints presented in Section 3, the use cases are separated into two groups, interpretative and generative use cases. The different use cases will be shortly presented and illustrated with examples.

4.1 Interpretative Use Cases

Interpretative use cases are based on the interpretative approach which has been presented in Section 3.2. Possible interpretative use cases are *model verification*, *testing*, *design by contract/run-time verification*, *simulation/animation*, and *querying*.

Model Verification: Using the OCL2 Interpreter, constraints can be verified on models. The constraints are defined on the meta-model (Mn = M2) which shall be used for modeling. The constraints are interpreted during the modeling process and if constraint violations occur, the user is informed that the model contains invalid constructs. Different reasons for model verification exist:

OCL constraints can be used to describe so called *well-formedness rules (WFRs)* that specify, what is required and not allowed in all instances of such a meta-model. For example one WFR of the UML2 meta-model defines, that all features owned by an interface must be public [2]:

```
context Interface
self.feature->forAll(f | f.visibility = #public)
```

The task of UML tools is to verify the consistency of the users' software models according to the UML meta-model. Another case of model verification is the definition of WFRs on a *domain-specific language (DSL)*. One example is the *Plugin Modeling Language (PML)*, which can be used to model Eclipse plug-ins [13]. One WFR on the PML meta-model requires, that any plug-in in a PML model has an id which is not empty [13]:

```
context Plugin
inv: not self.id.oclIsUndefined()
```

The PML is provided as an example model (based on the EMF Ecore meta-model) for Dresden OCL2 for Eclipse. Figure 3 shows Dresden OCL2 for Eclipse: The *Model Browser* shows the PML meta-model and the WFRs mentioned above which have been loaded as a model into the toolkit (Mn+1). The *Model Instance Browser* shows two plug-in instances of the PML for which the WFR shall be checked. The *OCL2 Interpreter* shows the results of the WFR's interpretation which succeeded for one of the two plug-ins.

Besides the definition of WFRs on existing data-types, some meta-models like UML [2] can also be extended by defining stereotypes. New *UML profiles* can be created which specialize the meta-model for a certain context. E. g., using the case tool *Magic Draw® UML* [10], it is possible to verify OCL constraints on models which are specified on meta-model stereotypes. Another example

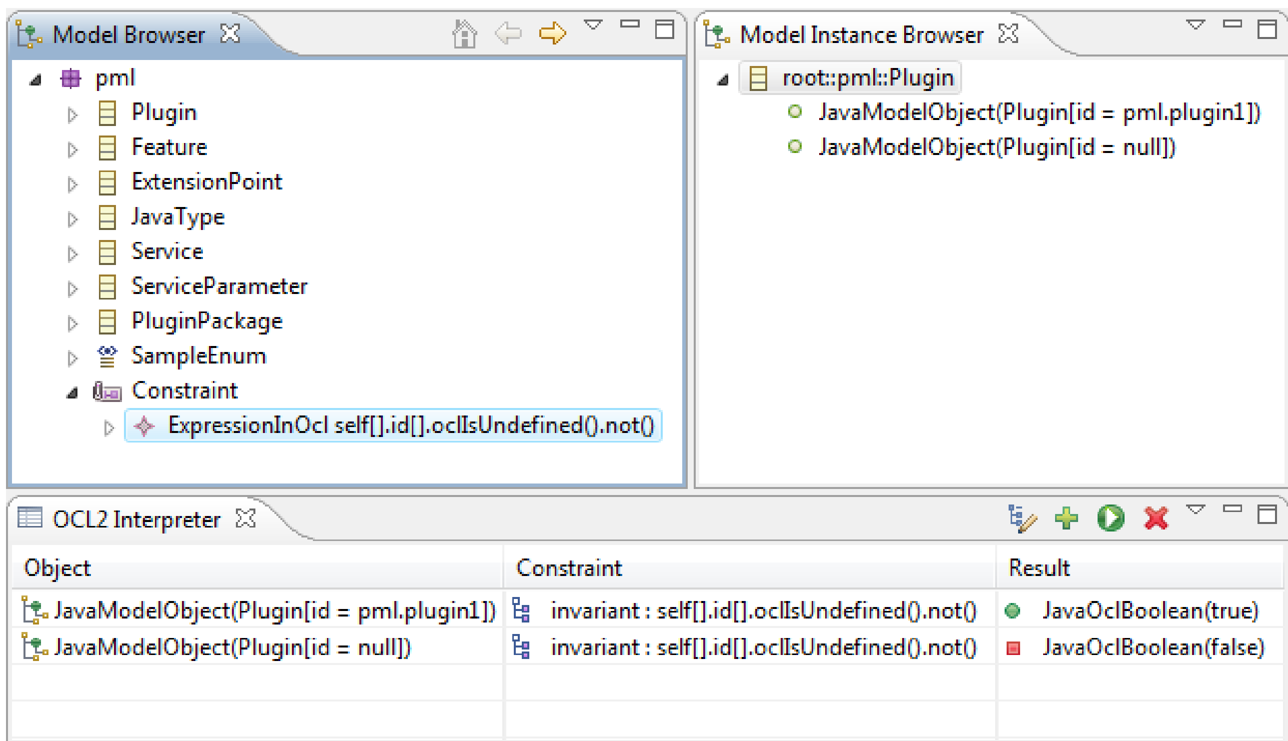

Figure 3: A Screenshot of Dresden OCL2 for Eclipse: The Model Browser showing the PML meta-model, the Model Instance Browser containing two Plug-ins, and the OCL2 Interpreter verifying, whether or not the plug-ins' id is set.

that uses stereotypes to define WFRs on meta-models is the *Agricultural Case Study* by Pinet et al. who use OCL to define an UML profile <<geographic>> for GIS data stored in rational data bases [14, 15].

Model verification could also be used to verify additional *modeling guidelines* for a specific context. For example a WFR could ensure that a class in an UML model should inherit from no, or only one, other class of the model if the model should be implemented in Java:

```
context Classifier
inv SingleInheritance:
self.generalization->size()<=1
```

Simulation/Animation: The interpretative approach can be used to animate or simulate models. A model, or meta-model, is modeled using a graphical editor (Mn = M1 or Mn = M2) and afterwards, the OCL Interpreter is used to animate or simulate the model. Simulation/animation can be realized for stateful models like UML activity diagrams or state charts [16], but is possible for UML class diagrams as well by instantiating the model with manually or automatically created model instances which are used as *snapshots* [17]. Although model instances are interpreted during model simulation or animation, the aim of this use case is not testing the model instances but testing the model. This is the major difference to the following use case *testing*. One of the first case tools which supported model animation using the interpretative approach was the case tool *USE (UML-based Specification Environment)* [17, 18]. A case study investigating how models should be tested by using USE can be found in [19].

Testing: Another use case of the interpretative approach is testing model instances. The OCL2 Interpreter can be used to check constraints defined on a model (Mn = M1) for individual model instances during the software development process. The constraints can be verified by the software engineer using interpretation on especially generated or defined model instances to check whether or not the defined constraints are fulfilled for the developed software implementation.

Run-Time Verification: Additionally, the interpretative approach can be used to verify constraints during software runtime. Such verification is commonly known as *design by contract* [20] or *run-time verification* [21]. OCL constraints are defined on a model (Mn = M1) and the OCL2 Interpreter is integrated into a runtime environment which interprets the constraints for all instances of the model during their execution. Note that in contrast to testing in using design by contract, the constraints are interpreted during software run-time and not only during software testing in the software development process. Runtime verification is subject of current research at the *Software Technology Group* and will be implemented by adapting the OCL2 Interpreter of

Dresden OCL2 for Eclipse to the contracting language Treaty [22, 23].

Another research project on run-time verification has been realized at the University of Nice-Sophia Antipolis, France. The OCL2 Interpreter of Dresden OCL (2005 release) was used to verify adaptations of components at run-time [24, 25, 26]. Adaptations are considered as behavioral and assembly changes in a component composition at runtime. Different ways to adapt a component system were described as *adaptation patterns* and *safety properties* at the model level [26]. Safety properties were described as sets of OCL constraints which have to be ensured before adaptation. By interpreting the results of such constraints, the consistency of adaptations can be verified and the adaptation can be undone in case of consistency violation.

Querying: Querying with OCL is also possible using an OCL interpreter. OCL can be used to define new operations on a model (Mn = M1) which are executed and interpreted on model instances of the model. OCL is a powerful language that, with its navigation operator and especially its special operations like `allInstances()` and `oclIsNew()`, provides a strong basis to be used as a query language on both relational databases and run-time objects in object-oriented software systems. Kolovos et al. presented a case study which illustrated how OCL could be used to evaluate queries on relational databases [27]. For example, the OCL method `allInstances()` could be used to return all columns from a table in a database. In the UML case tool USE [18], OCL is used for querying as well. They use their interpreter to enable users to query on objects of snapshots during animation to find potential erroneous objects [17].

4.2 Generative Use Cases

Generative use cases are based on the generative approach presented in Section 3.3. Generative use cases can be divided into two major groups, *code generation* use cases and *model transformation* use cases. Generative use cases are *testing*, *design by contract/run-time verification*, *simulation/animation*, and *model transformation*.

Testing using Generated Code: The generative approach can be used to generate test code to verify constraints for objects during software development. OCL constraints are defined on a model (Mn = M1) for which a code implementation shall be tested (M0). The OCL22Java Code Generator can be used to generate test code (for example JUnit code [28]), which can be executed to verify the constraints for specific created model instances. Note that generated code for testing is executed by the software developers during the software development process and is not used during software runtime.

The Java Code Generator of Dresden OCL2 for Eclipse does not support JUnit code generation yet. But the template-based code generation could be easily adapted to generate JUnit code. Such an implementation has been investigated by a thesis at the Swiss Federal Institute of Technology, Zurich [29].

Design by Contract or Run-time Verification using Generated Code: Similar to testing is the generation of constraint code to realize *design by contract* [20] or *run-time verification* [21]. But in contrast to testing, the generated constraint code is not only executed during the software development process, but also during software run-time. Again, the constraints are defined on the model (Mn = M1) and executable code is generated using the templates defined on the meta-model (M2).

Different solutions exist to realize run-time verification or design by contract using generated code. The old Java Code Generator of the Dresden OCL Toolkit (2005 release) supported *code instrumentation* of Java code, which generated the assertion code directly into Java source code [30, 31]. Dresden OCL2 for Eclipse provides an OCL22Java Code Generator which generates AspectJ [12] code that can be woven into existing Java code to ensure the specified constraints at software run-time [32]. Some of the advantages of an aspect-oriented approach are that the verification code can be woven into both, source and byte code, and that the concern of verification is cleanly separated from the business logic.

Simulation/Animation using Generated Code: Generated code of the Code Generator could be used to animate or simulate model elements as well as an OCL interpreter. The modeled meta-model, or model (Mn = M2 or Mn = M1), enriched with OCL constraints could be transformed into executable code snippets which are used to animate or simulate the model in a graphical editor during modeling [16].

For example, Dresden OCL has been integrated into the modeling case tool *MagicDraw® UML* [10], which uses the toolkit's Java Code Generator (2005 release) to enable UML model and object animation.

Model Transformation: Another major use case of the generative approach is model transformation. Model transformation uses a generation or transformation framework to transform a model into another model defined on another meta-model. A model (Mn = M1) is transformed using transformation rules defined on its meta-model (M2). The OCL constraints specified on the model are transformed as well. Some model transformations are UML/OCL to SQL schema transformation, and UML/OCL to XML/XQuery transformation (both supported with the second version of the Dresden OCL Toolkit, 2005 release) [10]. Furthermore, other model transformations like UML/OCL

to *SBVR* [34], or vice versa, have been evaluated in research projects [35].

Note that depending on the target meta-model, model and constraints can be transformed together into a new model (e.g., SBVR), or the OCL constraints can be transformed into additional queries which work on the transformed model (e.g., XML/XQuery or SQL).

Another approach of model transformation was developed at the *Software Technology Group* by integrating OCL constraints into the model transformation process of *Fujaba* [36, 37]. OCL support was integrated into Fujaba's story diagrams which are used to transform method specifications into executable code. In this context, OCL was not used to specify constraints on a model but to specify the model's semantics platform independently. The expressions defined in OCL in the method's semantic specification are used to transform the model into different platform specific models like Java or C++ code [37].

5. Conclusion

The Dresden OCL Toolkit looks back on a decade of research and development. Our intention was, and is still, to make several OCL tools available to other tool developers as an open source library under the LGPL license. In this paper, we presented different use cases in the software development that could benefit from using OCL. Some of the known integrations with other tools and tool chains as well as projects using Dresden OCL were cited as examples.

Most of the today's UML tools provide a "constraint field" according to the UML standard. However, only few UML tools support the processing of OCL constraints up to now. A first step in the processing of OCL constraints (or more generally OCL expressions) is parsing and checking the static semantics such as checking if all names are valid attributes and association ends in the underlying model. Such "preprocessed" OCL constraints helps to document the semantics of models. But the actual purpose using OCL is to evaluate the constraints on models or objects. Dresden OCL provides two basic evaluation approaches: the interpretative and the generative approach. Another facet is the matter on what MOF layer OCL expressions should be specified. It can be used, for example, to check WFRs (specified at the meta-model layer and evaluated on model) or to check business rules (specified at the model layer and evaluated on objects). Besides these basic model and object verification use cases, further scenarios are possible. The second version of OCL provides features to query models and objects and to derive new elements. Therewith, OCL becomes a model transformation and query language.

We know about many use cases of the Dresden OCL toolkit and are always interested to get feedback using OCL both in the industrial and academic field. Moreover, we invite the open source community to make contributions by new or enhanced tools. We are aware that the use of a formal language in the general software practice is a long way. Only user-friendly and robust OCL support can convince software developers to use OCL in their daily work to make software systems more secure and better maintainable.

Acknowledgments

The authors would like to thank all people who have contributed to the Dresden OCL Toolkit project. Moreover, we thank Florian Heidenreich for his helpful comments to this paper.

References

1. OMG OCL Specification: http://www.omg.org/docs/formal/06-05-01.pdf
2. OMG UML Specification: http://www.omg.org/spec/UML/2.2/
3. OMG MOF specification: http://www.omg.org/ technology/documents/formal/mof.htm
4. Eclipse Modeling Framework (EMF): http://www.eclipse.org/modeling/emf/
5. Dresden OCL Toolkit: http://dresden-ocl.sourceforge.net
6. Demuth B. "The Dresden OCL Toolkit and its Role in Information Systems Development" In: *13th International Conference on Information Systems Development: Methods and Tools, Theory and Practice Conference, Advances in Theory, Practice and Education (ISD'2004)*, Vilnius, Lithuania, 9-11 September, 2004.
7. Bräuer M., Demuth B. "Model-Level Integration of the OCL Standard Library Using a Pivot Model with Generics Support". In: Giese H. (ed.*) Models in Software Engineering*, Springer-Verlag Heidelberg, 2008, pp. 182-193. (*Lecture Notes in Computer Science* No. 5002).
8. Eclipse Model Development Tools (MDT): http://www.eclipse.org/modeling/mdt/
9. Topcased: http://www.topcased.org/
10. MagicDraw® UML: http://www.magicdraw.com/
11. Visual Paradigm: http://www.visual-paradigm.com/
12. The AspectJ Project: http://www.eclipse.org/aspectj/
13. Bräuer M. "Design and Prototypical Implementation of a Pivot Model as Exchange Format for Models and Metamodels in a QVT/OCL Development Environment". Minor Thesis (Großer Beleg), Technische Universität Dresden, Germany, May 2007.
14. Pinet F., Duboisset M., Demuth B., Schneider M., Soulignac V., Barnabé F. "Constraints Modeling in Agricultural Databases". In: Papajorgji P. J., Pardalos, P. M. (eds.) *Advances in Modeling Agricultural Systems Series*. Springer-Verlag,

Heidelberg, 2009, pp. 1-11. (*Springer Optimization and Its Applications*, No. 25).

15. Pinet F., Kang M.-A., Vigier F. “Spatial Constraint Modelling with a GIS Extension of UML and OCL: Application to Agricultural Information Systems”. In Wiil, U. K. (Ed.) *Metainformatics*, Springer-Verlag, Heidelberg, 2005, pp. 160-178. (*Lecture Notes in Computer Science* No. 3511).
16. Kirshin A, Moshkovich D, Hartman A. “A UML Simulator Based on a Generic Model Execution Engine”. In: Kühne T. (ed.) *Models in Software Engineering*, Springer-Verlag, Heidelberg, pp. 324-326. (*Lecture Notes of Computer Science* No. 4364).
17. Richters M., Gogolla M. “Validating UML models and OCL constraints.“ In Evans A., Kent S., Selic B. (eds.) *UML 2000 - The Unified Modeling Language*, Springer-Verlag, Heidelberg, 2000, pp. 265-277 (*Lecture Notes in Computer Science* No. 1939).
18. USE: http://www.db.informatik.uni-bremen.de/ projects/use/
19. Aydal E. G., Paige R. F., Woodcock J. “Observations for Assertion-based Scenarios in the context of Model Validation”. In: Cabot J., Gogolla M., Van Gorp P. (eds.), *Proceedings of the 8th International Workshop on OCL Concepts and Tools (OCL 2008) at MoDELS 2008*, Electronic Communications of the EASST, Technische Universität Berlin, 2008.
20. Meyer B. “Object-Oriented Software Construction”, Ed. 2. Prentice Hall, Upper Saddle River, New Jersey, 1997.
21. Colin S., Mariani, L. “Run-Time Verification” In: Broy. M, Jonsson B., Katoen J.-P., Leucker M, Pretschner A. (eds.) *Model-Based Testing of Reactive Systems*, Springer-Verlag, Heidelberg, 2005. (*Lecture Notes in Computer Science* No. 3472).
22. The Treaty Project: http://code.google.com/p/treaty/
23. Wilke C. “Model-Based Run-Time Verification of Software Components by Integrating OCL into Treaty”. Diploma Thesis, Technische Universität Dresden, Germany, scheduled in September 2009.
24. Ocello A., Dery-Pinna A.-M., Riveill, M. “A Runtime Model for Monitoring Software Adaptation Safety and its Concretisation as a Service”. In: Bencomo N., Blair G., France., Muñoz F., Jenneret C. (eds.) *Proceedings of the 3rd Workshop on Models@run.time at the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS2008)*, Toulouse, France, 2008, pp. 67-76.
25. Ocello A., Dery-Pinna A.-M. “Safe runtime adaptations of components: a UML metamodel with OCL constraints.” In: Kniesel G., Mens T. (eds.) *Proceedings of the 1st International Workshop on Foundations of Unanticipated Software Evolution (FUSE)*, Barcelona, Spain, March 2004, pp. 69-83.
26. 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: *Software Testing Verification and Validation Workshop, 2008. ICSTW '08. IEEE International Conference on Software Testing, Verification, and Validation (ICST)*, IEEE Digital Library, Lillehammer, April 2008, pp. 113-120.
27. Kolovos D. S., Paige R. F., Polack F. A. C. “Towards using OCL for Instance-Level Queries in Domain Specific Languages” In Demuth B., Chiorean D., Gogolla M., Warmer J. (eds.) *Proceedings of the 6th OCL Workshop “OCL for (Meta-) Models in Multiple Application Domain” at the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS2006)*, Technische Universität Dresden, Germany, September 2006, pp. 26-37.
28. JUnit: http://www.junit.org/
29. Stock, M. “Automatic Generation of JUnit Test-Harnesses.” Semester Thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, March 2007.
30. Wiebicke, R “Utility Support for Checking OCL Business Rules in Java Programs”. Diploma Thesis, Technische Universität Dresden, Germany, December 2000.
31. Brandt, R. „Java-Codegenerierung und Instrumentierung von Java-Programmen in der metamodellbasierten Architektur des Dresden OCL Toolkit“. Minor Thesis (Großer Beleg), Technische Universität Dresden, Germany, September 2006. Published in German.
32. Wilke C. “Java Code Generation for Dresden OCL2 for Eclipse.” Minor Thesis (Großer Beleg), Technische Universität Dresden, Germany, March 2009.
33. 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.) *Proceedings of the Workshop Ocl4All: Modelling Systems with OCL at MoDELS 2007*, Technische Universität Berlin, Germany, 2008. (*Electronic Communications of the EASST (ECEASST)*, No. 9).
34. OMG SBVR Specification: http://www.omg.org/spec/SBVR/1.0/
35. Pau R., Cabot J. “Paraphrasing OCL Expressions with SBVR”. In: Kapetanios E., Sugumaran V., Spiliopoulou M. (eds.) *Natural Language and Information Systems*, Springer-Verlag, Heidelberg, 2008, pp. 311-316. (*Lecture Notes of Computer Science* No. 5039).
36. Fujaba Tool Suite: http://www.fujaba.de/
37. Stölzel M., Zschaler S., Geiger L. “Integrating OCL and Model Transformations in Fujaba” In Demuth B., Chiorean D., Gogolla M., Warmer J. (eds.)

Proceedings of the 6th OCL Workshop "OCL for (Meta-) Models in Multiple Application Domain" at the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS2006), Technische Universität Dresden, Germany, September 2006, pp. 140-150.