On OCL as part of the metamodeling framework MOFLON

C. Amelunxen, A. Schürr

Darmstadt University of Technology, Real-Time Systems Lab,
Merckstrasse 25, 64283 Darmstadt, Germany
[amelunx|schuerr]@es.tu-darmstadt.de
http://www.es.tu-darmstadt.de

Abstract. The metamodeling framework MOFLON combines MOF 2.0, OCL 2.0 and graph transformations to generate sophisticated metamodel implementations. In this paper we describe the role of OCL in MOFLON. Furthermore, we present a set of constraints which corrects, completes and improves MOF 2.0 for the application as graph schema language.

1 Introduction

Nowadays, model driven software development is mostly realized by the application of the Unified Modeling Language (UML) [Obj05b]. Although UML satisfies many popular needs, sometimes domain specific languages are required to meet special concerns. Domain specific languages can be designed using the Meta Object Facility (MOF) [Obj06]. In its latest version 2.0, MOF offers constructs for the modeling of static structures, which in its original purpose should be used to describe the abstract syntax of modeling languages. Such a model of a modeling language (called metamodel) can be used to build an editor for the application of the modeled language. The datamodel of the editor can be generated from the language's metamodel [Dir02].

The generated metamodel reflects the abstract syntax of the modeled language which could only lead to a static datamodel. Additionally, the MOF compliant metamodel can be enriched by constraints formulated in the Object Constraint Language (OCL) [Obj05a]. From such a combination of MOF and OCL, metamodels with an additional constraint evaluation mechanism can be generated. The constraint evaluation offers a more precise verification of the static semantics and can be used to build an analysis component on top of an editor's datamodel.

The MOFLON framework [AKRS06] extends this approach by the application of story driven modeling [Zün01]. The combination of MOF, OCL and graph transformation allows the generation of very sophisticated metamodels which are far more than just a simple datamodel. Due to the specification of behavior by graph transformations, the generated metamodel can cover all actions that are based on the datamodel. Thus, the additional environment of the generated metamodel, for instance an editor GUI, may consist of just a tight, straight

forward implementation to instantiate the generated metamodel. Due to the fact that any logic related code is generated from a copious specification, we achieve a high degree of flexibility and maintainability. The combination of MOF, OCL and graph transformations offers the possibility to generate metamodels, which contain code to analyze metamodel instances as well as code to transform metamodel instances. The combination of analysis and transformation capabilities additionally provides the opportunity to specify transformations that correct metamodel instances in case of a failed analysis (by so-called repair actions), or in other words to transform the model if OCL constraints are violated.

Figure 1 gives an overview of the architecture of MOFLON. In the center of MOFLON, there is a MOF 2.0 metamodel which was created by a bootstrapping process. Beside MOF, OCL plays an important part in MOFLON. In the following we will concentrate on the relevance of OCL inside the MOFLON framework. Section 2 presents a scenario in which MOFLON and especially OCL as part of MOFLON can be applied in a very useful manner. Section 3 introduces a set of OCL constraints which we need to complete MOF 2.0 for the application as graph schema language. Finally we end up with a conclusion and a short overview about future work in section 4.

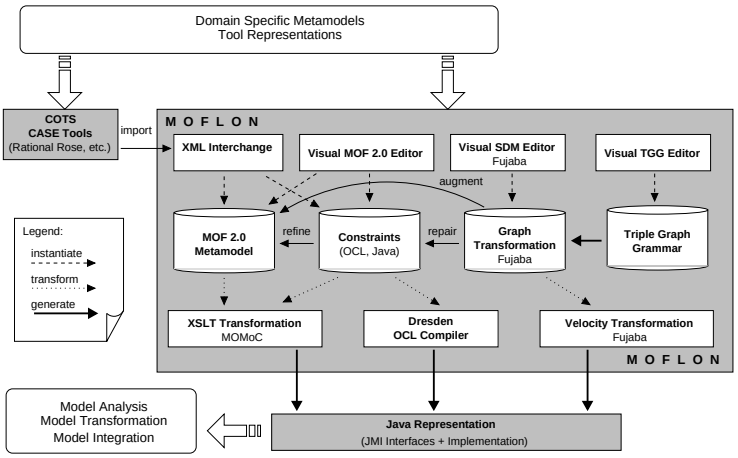

Fig. 1. Architecture of MOFLON

2 Application of MOFLON and OCL

One major problem of the model-based software and system developement is the surveilance of modeling guidelines which are indispensable in projects of a bigger size. The mere identification and formulation of design guidelines is a

problem which has to be handled manually without tool support. Opposed to that, the surveilance of the adherence to the formulated guidelines could be automated in the general case. In fact, the automation of the surveilance is in many cases mandatory just due the pure number and variety of the guidelines. Beside the automated surveilance and the automated identification and localization of modeling errors, the application of automated repair actions is a very desirable task [SDG+06]. As an example, we will demonstrate how MOFLON and especially OCL as part of MOFLON can help to generate code for the automated surveilance of design guidelines.

Figure 2 shows a very simple Matlab/Simulink model with a subsystem that is connected to a sink and a source through signals. The ports of the subsystem are named in adherence to a fictitious naming convention which demands that names of in-ports end with the suffix *_in*. First of all, the automatic surveilance of this modeling guideline requires a metamodel of Matlab/Simulink. A very simplified metamodel of Matlab/Simulink is depicted in Figure 3.

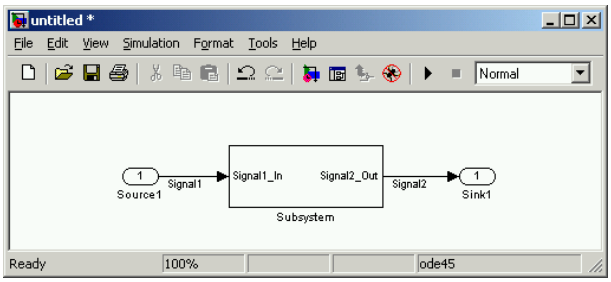

Fig. 2. Example of a design guideline in Matlab/Simulink

The metamodel covers only the elements which are necessary for the surveilance of the mentioned design guideline. *Blocks* can be connected with each other through *Ports* and *Signals*. A *Signal* connects exactly two ports, one in-port and one out-port. All these elements are named elements. At this point, OCL can be used in its original purpose to add precision by stating for instance that a *Port* can either refer to a *Signal* as *outSignal* or *inSignal*. Such a specification can only be used for code generation of static datamodels.

Additionally, MOFLON provides the feature to specify behavior using story driven modeling. Figure 4 shows the specification of the method *checkGuideline* which is intended to check if the mentioned design guideline is kept on a *Subsystem*. The behavior of the method is visually specified by a combination of activity diagrams and graph transformation rules. There is one graph transformation rule per activity (called story in this context). The transformation rule in the first story of Figure 4 matches all signals which are connected to an in-port of the subsystem the method is called on. Since only the ports which violate the mentioned design guideline should be handled by the transformation, the matching has to be controlled in such a way that only those ports are matched whose

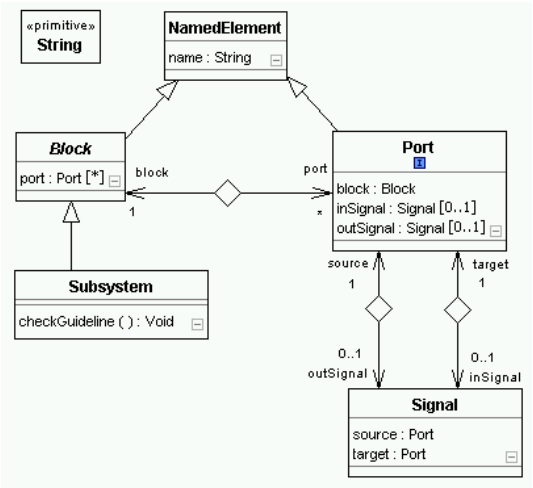

Fig. 3. Simplified metamodel of Matlab/Simulink

name do not end with the demanded suffix. At this point, we would also like to adopt OCL to be able to formulate constraints for the matching of attribute values which are more powerful than just a simple evaluation of attribute values. In general, OCL can also be used to determine the set of matched objects as a textual alternativ to complex graphical notations.

Considering the example, the violation of the design guideline is detected by an OCL constraint which checks whether the name of the port ends with the demanded suffix. In cases where such a match is found, the control flow of the activity diagram activates the second story in which an adequate repair action is formulated. An adequate repair action for the mentioned guideline is to set the name of the port to the name of its connected signal followed by the demanded suffix. Again, this manipulation[1] can be expressed by an OCL expression. The long term aim is to generate fully functional code for OCL constraints in the metamodel as well as for the OCL constraints in graph transformations. The outlined scenario is work in progress. Currently, MOFLON is able to generate JMI compliant metamodels enriched with code for the evaluation of invariants and code for the execution of SDM transformations.

Beside the integration of OCL into the matching and manipulation of the transformation rules and the application of OCL as constraint language for MOF, there is a third and very basic aspect how OCL is involved into the MOFLON approach. Since MOFLON applies MOF 2.0 as graph schema language, the static semantics of MOF are essential. They are the precondition for a proper application of OCL as mentioned before. Considering the graph transformation rules in Figure 4, the attribute *name* is used in the context of the class *Port*. An analysis which determines, if such a usage is possible has to be able to query all

[1] The manipulation of the attribute is indicated by the font color (green).

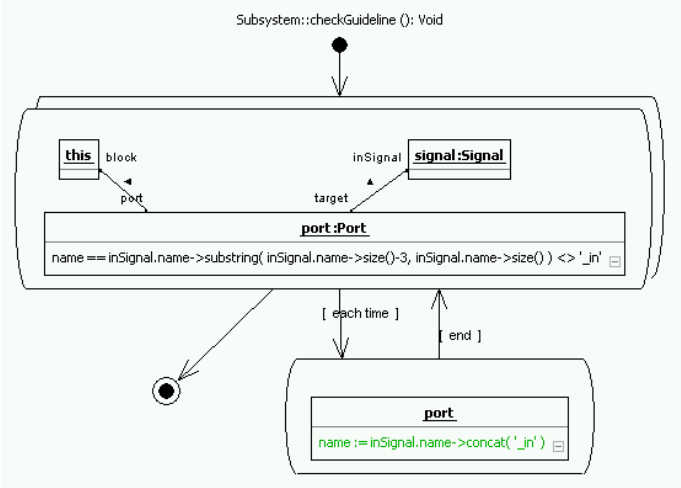

Fig. 4. OCL in graph transformations

inherited attributes of a class. The determination of inherited elements is part of the static semantics of MOF 2.0. In fact, the determination is formulated by an OCL constraint. Thus, the correct and complete static semantics are crucial for the complete integration of OCL in MOFLON. Therefore, in the following, this third aspect is described in detail.

3 Semantic completion of MOF 2.0 with OCL

As mentioned before in the context of Figure 1, MOF 2.0 is the central element of MOFLON. MOF 2.0 acts as metamodeling language for the static semantics of languages specified in MOFLON as well as schema language for the application of graph transformations. As such, the correct and precise semantics of MOF 2.0 are essential.

MOF 2.0 as the metametamodel of OMG's four-level metamodeling hierarchy is self describing. This means, that the static semantics of MOF 2.0 are basically described by MOF 2.0 metamodels but also additionally enhanced by OCL constraints and natural human language. The metamodels describing MOF 2.0 can be used to generate code for the basic features of MOF. Basic features comprise fundamental correlations like for instance the fact that associations are connected with classifiers through association ends. Such a correlation can be expressed by the metamodeling capabilities of MOF itself. But advanced and in many cases very important facts like the fact that only binary associations are allowed respectively in other words the number of association ends of one association has to be exactly two, can only be expressed with OCL.

The importance of OCL constraints can also be pointed out by considering the example of inheritance. The metamodel of MOF 2.0 only states that a classi-

fier can be generalized by a classifier. The fact that the generalization hierarchy has to be free of cycles can only be stated with OCL. Without the evaluation of OCL constraints, regardless of their implementation as hand written or fully generated code, a MOF 2.0 editor would be able to create arbitrary (e.g. cyclic) generalization dependencies. Although the importance of OCL constraints for the correct and complete static semantics of MOF 2.0 is often neglected, it has to attract careful attention for the purpose of applying MOF 2.0 as schema language for graph transformations.

Since MOFLON is developed by applying a bootstrapping process, the importance of a correct MOF 2.0 specification increases even more. We started our bootstrapping process with a simplified MOF 2.0 metamodel and a JMI compliant code generator (MOMoC [Bic04]). Based on the generated metamodel, we built an graphical editor and used this editor to improve the simplified MOF 2.0 metamodel of the editor. But even with the complete metamodel the editor does not prevent cyclic generalization dependencies, for instance. The bootstrapping process can only lead to an editor which reflects the complete static semantics of MOF 2.0 if the code generation also generates evaluation code for OCL constraints. Since, the metamodel consists of MOF and OCL, the bootstrapping can only be finished if both parts are reflected in the generated code. This can only be achieved if both parts are in a state which allows the application of a code generator. Hence, we had to analyze the MOF 2.0 specification for specification errors and impreciseness to be able to formulate a set of OCL constraints which formalizes MOF 2.0 up to a degree that is required for the application of graph transformations.

An application of a validation tool for syntax and type checking like done in [BGG04] for the UML 2 Superstructure might act as a basis. But since we are primarily interested in detecting impreciseness and specification leaks, an automated validation is not sufficient as imprecise semantics can be expressed even by proper and accurate constraints. Thus, we focused on a careful manual analysis to concentrate on semantical errors instead of detecting syntactical errors by the application of an OCL validater. Syntactical errors will at least be detected when we finish the bootstrapping process by the integration of generated evaluation code.

The result of our analysis is a set of OCL constraints which corrects and completes MOF 2.0 for the application as graph transformation language. The complete set of constraints cannot be presented in this paper. It is completely available at [Rea05]. In the following we provide information on the constraint set by introducing the error categories with some exemplary errors and the organization of the constraint set. The presented errors should provide an impression of the kind of errors that are part of the MOF 2.0 specification.

The presented examples refer to [Obj03] respectively [Obj04] where mentioned. In fact, both documents are not the latest available specifications, but since the combination of the presented corrections and improvements with the referred specification forms a complete and consistent specification a continuous adaption to the actual OMG documents does not seem reasonable to us. Never-

theless, a spot check of some selected errors leads to the result that some minor issues are fixed but major errors still exist.

3.1 Syntactical errors

The first and most obvious category of errors is the category of syntactical errors. Syntactical errors are more or less trivial to detect since they arise from a wrong usage of OCL. An automatic analysis would have been able to find such errors as well. In the following we will give some examples of typical syntactical errors in the MOF specification, apart from mere typos.

The cause of many errors is the wrong usage of OCL methods like, for instance, the application of the method *includes* with a collection passed as parameter instead of a single object (see [Obj04], p. 79). Another representative error is demonstrated by the application of squared brackets to access elements by their index: $forAll(i|op.ownedParameter[i].type.conformsTo(...))$ (see [Obj03], p. 149). Sometimes methods are used in a wrong context like the concatenation of strings by applying the method *union* (which should be applied to collections) instead of applying the method *concat*. Beside those obvious syntactical errors there are methods which are specified but never used (*bestVisibility*, see [Obj03], p. 93). Of course, this can hardly be considered as error, but at least it might confuse the reader of the specification.

3.2 Semantical errors

The category of semantical errors covers errors that are caused by any other reason than just the wrong application of OCL syntax. That may also comprise automatically detectable errors like wrong navigation in the metamodel. The navigation *association.owningAssociation* in the context of the class *Property* (see [Obj04], p. 131) for instance, is not possible. In fact, each of both association ends could be used to start navigation but a combination of both is not possible since both address the same class.

Another annoying but quite common kind of errors are wrong derivation rules. Some attributes of the metaclasses are derived by an OCL constraint. Those derived attributes are usually quite important, like for instance the derived set *inheritedMember*. It specifies all elements inherited from the general classifiers. The derivation rule for *inheritedMember* uses the specified method *hasVisibilityOf* which in turn also uses the attribute *inheritedMember* in its specification (see [Obj04], p. 85).

In fact, such an error would also be detected by an analysis tool whereas the following error is not that obvious and could only be detected by a human analysis. The derived attribute *importedMember* determines the named elements that are (transitively) imported from several namespaces into the importing namespace. Both kinds of import (package import and element import) have a visibility to determine whether the elements imported via a certain import will be exported from the importing namespace or not. In other words, the visibility of an import can be used to control the transitivity of the import. Thus, the

derivation rule for *importedMember* has to take the visibilities of the involved import relationships into account, which it does not (see [Obj03], p. 143).

There are also errors in form of specification leaks that arise from the complex package structure. The method *conformsTo* of the metaclass *Classifier* for instance is defined in the package *Generalizations*. The central package of the UML Infrastructure respectively MOF is the package *Constructs* which reuses several packages except the *Generalizations* package. Thus, the method *conformsTo* is not available in MOF although it is used. Beside the constraints that are wrong, the specification is also vitiated by bad namings and by missing constraints like for instance a constraint that prevents a namespace from importing itself.

3.3 Customizations and Improvements

A chance for customization and improvement does not necessarily require an error. On the one hand customizations and improvements contribute to a clear and consistent specification by clarifying complex and confusing constraints. On the other hand, additional constraints improve the completeness of the specification by handling special cases as well as optional and implicit demands. In fact, some actually correct constraints can be combined to a single constraint like for instance the two invariants for the determination of a named element's qualified name (see [Obj03], p.78).

Beside such trivial improvements there are also improvements with an important impact on the complete specification like exemplarily described in the following. As mentioned before, there are two kinds of import. First of all there is the commonly known package import that adds all the elements of the imported namespace with visibility set to *public* to the importing namespace. Second, there is the element import that adds only a single element with visibility set to *public* to the importing namespace. So obviously, the package import is just a shortcut notation for element imports on each element of the imported namespace and as such, both variants should be exchangeable (see [Obj03], p.145). But a problem arises due to the transitivity of the imports since an element import can use alias names for the imported element in the importing namespace. Thus, the constraints which determine the elements of a namespace must be able to query the alias name of an imported element. Without the loss of transitivity, this can only be achieved by extending the metaclass *ElementImport* by an attribute to determine previous element imports.

Indeed, a lot of modifications have to be made to take such extensive changes into account but without doing so, the specification would neither be correct nor consistent. Not all improvements contribute to the correctness of the specification. We propose also constraints whose adherence is not necessary but desirable. There are also situations that are implied by other situations. Such implications can also be expressed via OCL constraints. Therefore, we had to introduce several categories and levels of errors which are described in the following.

3.4 Categorization

The complete set of constraints we propose for a correct and consistent specification of MOF 2.0 can be found at [Rea05] in form of several tables. In the following we describe the classifications we made. The classifications are all reflected in the tables. All constraints are formulated in OCL 2.0.

First of all, the constraints are classified based on their error category. Additionally it is indicated whether the constraint is an existing constraint of the specification or an proposed extension. Table 1 summarizes the several categories.

Description	**Shortcut**	**Number**
Existing constraint is correct[2]	ok	1
Existing constraint ...[3]		
... is syntactical wrong.	SYN	2
... is semantical wrong or problematic.	SEM	3
... can or has to be customized.	CUS	4
Proposed constraint ...[4]		
... solves syntactical error.	SYN	5
... solves semantical errors or problems.	SEM	6
... improves or customizes.	CUS	7
... adds additional precision.	+SEM	8
... adds an additional customization.	+CUS	9

Table 1. Categorization of constraints regarding their kind of error.

As mentioned before, not every constraint has to be met for a correct metamodel instance. For some constraints, the adherence of the metamodel instances to the constraint is just desirable but not necessary. Therefore, we had to introduce a second categorization which is orthogonal to the categorization introduced first. We categorize the constraints into three different types of constraints. Table 2 introduces the three types of constraints.

Type of constraint	**Shortcut**
Mandatory for valid metamodel instances.	mand.
Optional for valid metamodel instances though useful	opt.
Implicit constraint	impl.

Table 2. Types of constraints regarding their role in the specification.

Constraints of type mand. have to be met for valid metamodel instances. If one single constraint of type mand. is broken the complete metamodel instance is invalid. Whereas constraints of type opt. do not contribute to the correctness

of the metamodel instances at all. A metamodel instance can be valid, although some or even all optional constraints are broken, as long as all mandatory constraints are met. But, the adherence to all optional constraints leads to useful metamodel instances without senseless constructs like for instance a namespace importing itself.

Beside these two quite obvious types, there is a third type i.e. the implicit constraints. An implicit constraint does not contribute to the correctness of metamodel instances as well. It indicates whether a metamodel instance contains constructs that are not explicitly drawn in the diagram but are implied by other constructs. For instance, considering the subsetting of association ends, the uniqueness of association ends which determines if multi-valued association ends may contain duplicates or not, is subject of implicit constraints. If one association end subsets another association end which is unique, it is implied that the subsetting association end is also unique. If it is mentioned in the diagram that the subsetting association end may contain duplicates, it indeed never will contain duplicates since the subsetted association end prevents such a situation by its uniqueness. Such implied correlations can be detected by OCL constraints.

Considering an editor which is build on top of a metamodel that is enriched by constraints of all three types, during evaluation, a violated mandatory constraint would rise an error message, a violated optional constraint would rise a warning message, whereas a violated implicit constraint would just cause an information message. That is how the constraint set will be integrated into MOFLON.

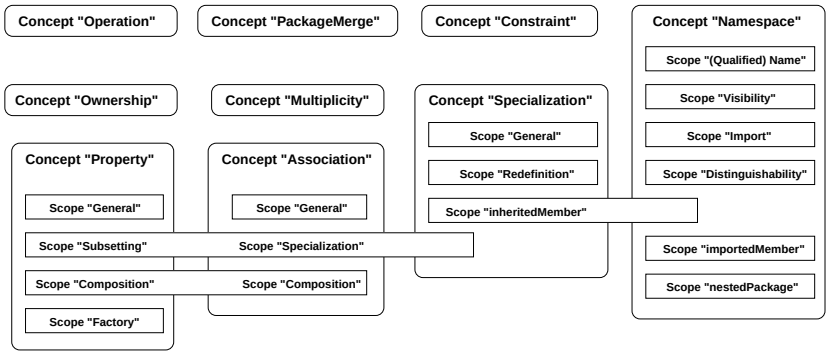

Fig. 5. Concepts and scopes of constraints

For a better structuring, the tables are separated according to their concept and the scope they cover. The constraints are grouped into concepts like for instance Association or Namespace. Those concepts do not map onto a package structure. They just group several constraints that contribute to the realization of the concept they are grouped into independent from their context. Constraints of the concept Association for instance are taken from the context

Core::Constructs::Association as well as from *Core::Constructs::Property*. Concepts are divided in several scopes to achieve a more flexible structure since some scopes contribute to more than just one concept. Figure 5 shows the complete set of concepts and scopes.

All in all, 90 constraints have been analyzed. As a result, 48 constraints (53%) have been considered free of any errors and 42 constraints (47%) were erroneous. Our constraint set provides 101 improvements in form of 86 OCL constraints. One half (51) of the improvements refer to existing constraints. The other half (50) improves the specification additionally. Fig. 6 shows the absolute numbers of constraints distributed over the categories of Table 1. The y-axis indicates the number of the category, the x-axis represents the number of occurrences in the proposed constraint set.

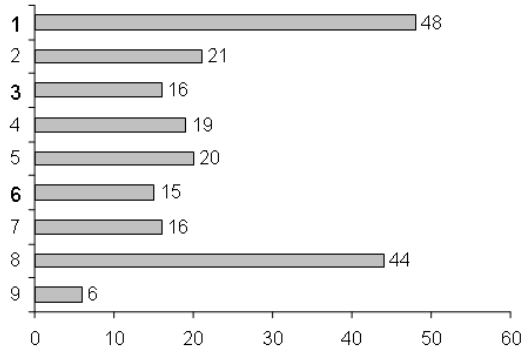

Fig. 6. Occurrences of errors grouped by their category

4 Conclusion and Future Work

The presented set of OCL constraints is the result of an intensive analysis of the semantics of MOF 2.0 as a whole. It is the result of a manual analysis since we are primarily interested in a semantical consistent version of MOF 2.0 for the application as graph schema language. We will use this set of constraints to complete the bootstrapping process of MOFLON. Therefore, we had to integrate a OCL 2.0 parser and code generator. After an evaluation of available and appropriate tools we decided to integrate the Dresden OCL toolkit [LO04]. Since the OCL code generator component is still work in progress we are only able to generate evaluation code for a simple subset of the constraints. After an adaptation of the constraints to the capabilities of the code generation we will generate an analysis component for the MOFLON framework which checks edited metamodels to their full compliance to MOF 2.0. The current integration of the toolkit is based on the provided integration interfaces. The long term aim is

to exchange the toolkit's model repository (MDR) with a integrated metamodel that is generated with MOFLON.

Acknowledgement

The authors are grateful to Sascha Rüter for his outstanding work that contributes to this report.

References

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink and J. Warmer, editors, *Model Driven Architecture - Foundations and Applications: Second European Conference*, volume 4066 of *Lecture Notes in Computer Science (LNCS)*, pages 361–375, Heidelberg, 2006. Springer Verlag, Springer Verlag.

[BGG04] Hanna Bauerdick, Martin Gogolla, and Fabian Gutsche. Detecting OCL Traps in the UML 2.0 Superstructure: An Experience Report. In Thomas Baar, Alfred Strohmeier, Ana Moreira, and Stephen J. Mellor, editors, *Proc. 7th Int. Conf. Unified Modeling Language (UML'2004)*, volume 3273 of *Lecture Notes in Computer Science (LNCS)*, pages 188–197, Berlin, 2004. Springer Verlag, Springer Verlag.

[Bic04] Bichler. *Codegeneratoren für MOF-basierte Modellierungssprachen*. PhD thesis, Universität der Bundeswehr München, 2004. German.

[Dir02] Ravi Dirckze. *Java^{TM} Metadata Interface (JMI) Specification, Version 1.0*. Unisys, June 2002.

[LO04] S. Löcher and S. Ocke. A Metamodel-Based OCL-Compiler for UML and MOF. In P. Schmitt, editor, *Workshop Proc. OCL 2.0 - Industry standard or scientific playground?*, volume 102 of *Electronic Notes in Theoretical Computer Science*, pages 43–61. Elsevier, November 2004.

[Obj03] Object Management Group. *UML 2.0 Infrastructure Specification*, September 2003. ptc/03-09-15.

[Obj04] Object Management Group. *UML 2.0 Infrastructure Specification*, November 2004. ptc/04-10-14.

[Obj05a] Object Management Group. *OCL 2.0 Specification*, Juni 2005. ptc/2005-06-06.

[Obj05b] Object Management Group. *UML 2.0 Superstructure Specification*, August 2005. formal-05-07-04.

[Obj06] Object Management Group. *Meta Object Facility (MOF) Core Specification*, January 2006. formal/06-01-01.

[Rea05] Real-Time Systems Lab, Darmstadt University of Technology. *Set of OCL constraints for a consistent MOF 2.0*, 2005. http://www.es.tu-darmstadt.de/download/publications/amelunxen/Constraints.pdf.

[SDG+06] I. Stürmer, H. Dörr, H. Giese, U. Kelter, A. Schürr, and A. Zündorf. *Das MATE Projekt - visuelle Spezifikation von MATLAB-Analysen und Transformationen*. Gesellschaft für Informatik, Bonn, 2006. submitted for publication, in German.

[Zün01] Albert Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001. Habilitation Thesis.