Electronic Notes in Theoretical Computer Science 102 (2004) 63–75 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

OclType – A Type or Metatype ?

# Stephan Flake[1](#_bookmark1)

*C-LAB, Paderborn University Fuerstenallee 11*

*33102 Paderborn, Germany*

Abstract

While the type system proposed in the OCL Standard Library of the latest OCL 2.0 proposal seems to be considerably stable by now, there are still some deficiencies in the definition of operations for type casts and type conformance checks. This results from the fact that the types defined on the user-level are currently not well represented in the OCL Standard Library.

This article presents a new modeling approach to adequately capture these types in the OCL Standard Library through the UML core concept called powertype. The powertype concept allows to model a metaelement on the architectural user level M1. By this approach, we propose an enhanced structure of the OCL Standard Library that prescribes a controlled way for accessing the metalevel.

*Keywords:* OCL Standard Library, Powertype

# Introduction

While the adoption of the latest version 1.6 of the *Response to the UML*

*2.0 OCL Request for Proposals* (in the following referred to as the *OCL 2.0 proposal*) is an important step in the development of the language, there are still a number of issues to solve. We argue that it is very important that further enhancements of the OCL 2.0 proposal are made prior to finalization of the UML 2.0 standard. Otherwise, apparent flaws in the language definition will prevent tool developers and users from adopting OCL, such that OCL will basically remain a “scientific playground”. As another likely consequence, tool

1 Email: [flake@c-lab.de](mailto:%20flake@c-lab.de)

1571-0661 © 2004 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2003.09.004

builders might adopt their own solutions to unresolved flaws of the OCL 2.0 specification.

The OCL 2.0 proposal is currently neither consistent nor complete, e.g., the formal semantics of some newly introduced concepts such as OCL messages have not been considered sufficiently. But also concepts that were already present in OCL as part of UML 1.5 have still not been sufficiently defined, e.g., no semantics is provided for state-related operations, although the syntax allows to specify constraints over state activations.

Apart from those semantic issues, this article focuses on improved modeling of the predefined type OclType within the OCL Standard Library and correct specification of related operations. This work is therefore to be seen as a contribution to the finalization process of OCL for UML 2.0.

* 1. *OclType in the OCL 2.0 Proposal*

OCL is a typed expression language and includes a set of predefined types, such as String, Integer, and Real. They are kept in a type system that is part of the so-called OCL Standard Library residing on layer M1 of the UML 4-layer architecture [[4](#_bookmark20), Chapter 6].

Naturally, it does not make sense to formulate constraints without a UML model to refer to. In the remainder, we will call this UML model the *referred UML user model*. Each Classifier [2](#_bookmark2) defined within the referred UML user model represents a distinct OCL type and is implicitly included in the OCL Standard Library as a subtype of OclAny.

Among the predefined OCL types of the Standard Library, OclType has a special role. That type captures all basic types that are known in the OCL type system. [3](#_bookmark3) The interesting point is that OclType on the one hand is inherently a sort of metaelement belonging to the metalevel M2, while on the other hand it has to be accessible on the user level M1 to formulate constraints that need to reason about object types (e.g., type conformance checks).

The approach taken in the OCL 2.0 proposal models OclType as an enu- meration type [[4](#_bookmark20), Section 6.2]. Modeling OclType as an enumeration type has been chosen because access to the metalevel should no longer be supported, as opposed to the OCL definition as part of the current UML 1.5 standard [[5](#_bookmark21), Chapter 6]. Figure [1](#_bookmark4) graphically illustrates the proposed definition of OclType together with the predefined basic OCL types as enumeration literals. We have

2 More precisely, one has to speak of “each instance of metatype Classifier”, but we here adopt the terms used in the UML specifications.

3 We call all predefined non-parameterized OCL types *basic types*. This includes the user-

defined Classifiers of the referred UML user model and excludes parameterized types like collection types and OCL message types.

|  |
| --- |
| «enumeration» OclType |
| OclAny OclType OclState Boolean Real Integer String OclVoid (...) |
| =(typename:OclType) : Boolean  <>(typename:OclType) : Boolean |

Fig. 1. Enumeration Type OclType (as proposed in the OCL 2.0 proposal)

This ellipsis (...) indicates that all user-defined classifiers of a given UML model also become enumeration literals of OclType

used a commented ellipsis to indicate that additional enumeration literals have to be added based upon the referred UML user model.

* 1. *Operations that make use of OclType as a Parameter Type*

In Table [1](#_bookmark5), we list the set of OCL types and corresponding operations that make use of OclType [[4](#_bookmark20), Sections 6.2.1 and 6.3.1]. Note that the literal T represents an arbitrary basic OCL type.

Table 1

Appearance of OclType in the OCL 2.0 Proposal

|  |  |
| --- | --- |
| OCL Type | Operation Signature |
| OclAny | oclAsType (typename:OclType) : T  oclIsTypeOf (typename:OclType) : Boolean oclIsKindOf (typename:OclType) : Boolean |
| OclModelElement | = (object:OclType) : Boolean  <> (object:OclType) : Boolean |
| OclType | = (object:OclType) : Boolean  <> (object:OclType) : Boolean |

There are a number of problems already with the signatures of these op- erations. First, the signatures of operations = and <> are simply wrong for OclModelElement. Type OclModelElement is seen as an enumeration type in the OCL 2.0 proposal, where for each *element* in a given UML user model there is a corresponding enumeration literal. OclModelElement must there- fore comprise all user-defined Classifiers, but also all other model elements like Statechart states, transitions, events, operations, signals, etc. While it is questionable whether this type is necessary and makes sense at all, the cor- responding operation signatures must be corrected, i.e., the formal parameter types must be OclModelElement instead of OclType.

In OCL expressions, it is sometimes necessary to access the *type* of a user- defined class, e.g., to perform a type cast or to check for a certain subtype. In this context, operation oclAsType(typename:OclType) is used. That op- eration returns an object that is re-typed to a type specified by actual pa- rameter typename. [4](#_bookmark7) Conceptually, this means that the metalevel M2 has to be accessed to reason about type conformance between the current object type and the specified target type. As this is not possible with the approach taken in the OCL 2.0 proposal, formal specifications in form of OCL postcon- ditions cannot be provided for operations oclAsType(typename:OclType), oclIsTypeOf(typename:OclType), and oclIsKindOf(typename:OclType). Note that the corresponding passages are marked with “-- TBD” in the OCL

2.0 proposal.

In the following sections, we will consider the definition and usage of type OclType within different approaches in the literature (Section [2](#_bookmark6)) and the cur- rent OCL 2.0 proposal (Section [3](#_bookmark8)). In Section [4](#_bookmark14), we then show how the *powertype* concept can be introduced to better integrate types defined at the user-level into the OCL Standard Library type system on level M1. Section [5](#_bookmark16) concludes this article.

# Related Work

There are different possibilities to provide access to Classifiers. The current standard as defined in UML 1.5 uses OclType and refers to it as a *metatype* with access for the modeler [[5](#_bookmark21), Section 6.8.1.1]:

*All types deﬁned in a UML model, or pre-deﬁned within OCL, have a type. This type is an instance of the OCL type called OclType. Access to this type allows the modeler limited access to the meta-level of the model.*

In UML 1.5, there are even predefined operations provided for OclType to further access metalevel features, e.g., operations to get a list of attribute names, association end names, and direct as well as indirect supertype names (however, operations to extract subtypes are missing).

Modelers are thus able to access the metalevel in OCL expressions. Note that this breaks up the 4-layer architecture underlying the UML modeling approach. This problem has been identified and more recent OCL language definitions discard these operations.

Baar and Ha¨hnle suggest to model OclType as a pure metatype without

4 Although there are arguments that type cast operators should not occur in a specification language, the OCL 2.0 proposal gives an example in which such an operation is necessary to eliminate ambiguities [[4](#_bookmark20), Section 2.5.8].

access for the modeler [[1](#_bookmark17)]. Operations as mentioned above do not need to be defined on OclType in their approach, as the UML core metamodel already provides such means, either directly or indirectly (by navigation along associa- tions in the UML core metamodel). It is also worth noting in this context that their metamodel does not allow nested collections which are now permitted in the OCL 2.0 proposal.

Richters proposes a metatype called OclTypeType with OclType as its only instance on level M1 [[6](#_bookmark22), Section 6.4]. Richter’s OclType introduces a metaelement which is placed *above the level of “ordinary” OCL types*, i.e., the instances of OclType are all of the OCL types. The most useful operation of this type is allInstances() to extract the set of all currently existing objects for a given type. Moreover, all Classifiers of the referred UML model are duplicated and stored by an additional type called ObjectType on the M1 level. Semantically, the domain of an instance of ObjectType is the set of object identifiers defined for the Classifier and its children.

The approach we present in this article is similar w.r.t. Richters’ approach in the sense that OclType is a metaelement on level M1, but we follow a modeling approach that avoids the additional metatype OclTypeType as well as Classifier duplication by means of ObjectType.

# A Review of OclType’s Operations

As already mentioned in Section [1](#_bookmark0), OclType is residing on level M1 as a subtype of OclAny in the OCL 2.0 proposal. OclType is regarded as an enu- meration comprising the Classifier names of the referred UML user model. Besides the operations = and <> that are explicitly re-defined for OclType, there are 6 more operations that are inherited from OclAny (see Table [2](#_bookmark9)).

Table 2

Predefined Operations for OclAny

= (object2:OclAny) :

<> (object2:OclAny) : oclIsNew () :

oclIsUndefined () :

oclAsType (typename:OclType) : oclIsTypeOf (typename:OclType) : oclIsKindOf (typename:OclType) :

oclInState (statename:OclState) :

allInstances () :

Boolean Boolean Boolean Boolean T

Boolean Boolean Boolean

Set(T)

We now briefly review the semantics of these operations with respect to

OclType. Note here that we distinguish between primitive *values* of datatypes,

such as Integer values or the enumeration literals of OclType, and *objects* as instances of classes that can dynamically be created and destroyed.

* Operation oclIsNew() must return false when applied to datatype val- ues, as primitive values cannot dynamically be created or destroyed. This of course applies to the enumeration literals of OclType. (Or should this op- eration maybe return OclUndefined in these cases? The OCL 2.0 proposal does currently not specify the result.)
* Operation oclIsUndefined() must by definition always return false when applied to datatype values.
* Operation oclAsType(typename:OclType) returns an object that is re- typed to a type specified by actual parameter typename. We here only mention that there are some contradicting requirements in the OCL 2.0 proposal concerning up- and downcasts. This will be discussed in more detail in Subsection [3.1](#_bookmark10).
* Generally, operation oclIsTypeOf(typename:OclType) evaluates to true if it is applied to an object or datatype value that is of the type specified by typename.

When this operation is applied to an enumeration literal that itself de- notes an OCL type, say Integer, evaluation results in true only iff the argu- ment type is OclType. Thus, expression Integer.oclIsTypeOf(OclType) evaluates to true, but for all parameter values different from OclType the result is false. This semantics applies to all types of the OCL type system. But is this the desired result? Maybe not, as one might want in this con- text – in a more intuitive way – to reason about the (meta)type of a *type*, say Integer, and not an enumeration literal. The problem now is that the types of OCL types reside on the metalevel M2, e.g., the (meta)type for Integer is Primitive. But such metatypes are not known in the OCL type system on level M1, such that it does not make much sense to ap- ply operation oclIsTypeOf(typename:OclType) to any such enumeration

literal.

* Operation oclIsKindOf(typename:OclType) evaluates to true if it is ap- plied to an object or value whose type conforms to the type specified by typename. The same problem as for operation oclIsTypeOf() appears when applied to enumeration literals that denote an OCL type.
* Operation oclInState(statename:OclState) does not make sense when applied to datatype values and enumeration literals. In such cases, it should therefore return OclUndefined. A complete formal specification of opera- tion oclInState(statename:OclState) has already been developed and can be found in [[3](#_bookmark19)].
* Operation allInstances() returns all instances of the type to which it is applied. It may only be used for Classifiers (or: types) that have a finite number of instances, e.g., user-defined classes [[4](#_bookmark20), Section 6.2.1]. [5](#_bookmark11)

The following example is taken from the OCL 2.0 proposal. It restricts the number of instances of OCL type OclVoid to 1.

context OclVoid

inv: OclVoid.allInstances()->size() = 1

OclVoid is an OCL type, i.e., in an OCL expression as above, OclVoid evaluates to an enumeration literal of type OclType. Now, what is the result of applying operation allInstances() to an enumeration literal?

As another example, now assume a user-defined class Person and the OCL expression Person.allInstances(). Again, Person is identified as an enumeration literal of OclType, so what is the result of evaluating the corresponding OCL expression, i.e., what is the result of applying opera- tion allInstances() to an enumeration literal? This is clearly the wrong architectural level and not intended.

To conclude, there are several type-related difficulties introduced by the attempt to model OclType as an enumeration type in order to separate the predefined OCL types from the metalevel. On the other hand, it is not possible to properly formulate expressions with operations that reason about object types without (limited) access to the metalevel. It is also not possible to give a semantics of operations that reason about OCL types without accessing the metalevel.

Furthermore, it does simply not make sense to define allInstances() as an operation of the base type OclAny, as that operation should only be applied to types and not to objects. But with the currently proposed OCL type system, M1 types cannot be accessed, as user-defined classes (such as Person) and OCL types (such as OclVoid) are treated only as enumeration literals of the enumeration type OclType. It is simply not type conform to apply operation allInstances() to an enumeration literal, say Person, in order to obtain the set of objects of the class which the literal stands for.

* 1. *Re-typing Objects in the OCL 2.0 Proposal*

We now take a closer look at re-typing or casting with OclAny’s operation oclAsType(typename:OclType). That operation returns the same object, but the object’s type is reset to the type specified by actual parameter typename.

5 However, note that this issue is still being discussed controversally. The basic problem is that OCL does not have a notion of infinity to sustain an “executable flavor”, e.g., to perform code generation for assertions [[2](#_bookmark18)].

The signature of operation oclAsType() is not treated consistently in the OCL 2.0 proposal, as the return type of that operation is specified differently:

* + - In Section 2.5.9 of the OCL 2.0 proposal, the return type is specified as “instance of OclType”, and
    - in Section 6.2.1, the return type is the unnamed type T.

Both return types are not quite correctly specified. On the one hand, an instance of OclType is an enumeration literal and can therefore not be the re-typed object as intended. On the other hand, the unnamed type T has no concrete meaning in the signature, as T appears together with parameterized types only, but there actually is no such type involved. The correct signature therefore is

OclAny::oclAsType(typename:OclType) : OclAny .

As operation oclAsType() returns *the same object* possibly with a different type, this implies that the object is an instance of (a subtype of) OclAny, and this is all we can assume for the signature of that operation.

But another semantical issue arises in this context. In Section 2.4.6 of the OCL 2.0 proposal, it is required that operation oclAsType() may only be used to re-type an object to one of its *subtypes*, i.e., only *down-casts* are al- lowed. But on the other hand, operation oclAsType() is actually used in the OCL 2.0 proposal to also *up-cast* objects for accessing overridden properties of supertypes [[4](#_bookmark20), Secion 2.5.8]. This issue has to be solved by an OCL specifi- cation of operation oclAsType(). Note here that the current specification of operation oclAsType() does not take into consideration that the result may be OclUndefined [[5](#_bookmark21), Section 6.2.1].

No matter which of the two applicable semantics is finally chosen (down- cast only or both up- and down-cast), it is important to understand that such a postcondition *must* access the metalevel M2. This is because one has to reason about type conformance, which is modeled on the metalevel M2 through metatype Classifier and its operation conformsTo().

The OCL constraint shown in Figure [2](#_bookmark12) is an attempt to provide an OCL postcondition for operation oclAsType(typename:OclType) in the context of the current OCL 2.0 proposal. We here consider the more general case of up- *and* down-cast semantics. When only down-cast semantics is intended, we simply have to leave out the constraint condition in line 22.

We now give a brief explanation of the postcondition. First, variable selfTypes keeps the names of types that the self object is an instance of (lines 2 – 5). Note that the result may be a *set* of enumeration literals, as an object in UML may generally be a direct instance of more than one type (or: class). Certainly, this is a point that needs to be further discussed: The

1: context OclAny::oclAsType(typename:OclType) : OclAny 2: def: selfTypes =

3: OclType.allInstances()

4: ->select(t:OclType |

5: self.oclIsTypeOf(t)) : Set(OclType) 6: post: let

7: argClassifier =

8: Classifier.allInstances()

9: ->select(c:Classifier |

10: c.name = typename.toString()) 11: ->any(true) : Classifier,

12: selfClassifiers =

13: Classifier.allInstances()

14: ->select(c:Classifier |

15: selfTypes->exists(t:OclType |

16: c.name = t.toString())

17: ) : Set(Classifier)

18: in

19: if selfClassifiers

20: ->exists(c:Classifier |

21: argClassifier.conformsTo(c)

22: or c.conformsTo(argClassifier)) then 23: result = self and result.oclIsTypeOf(typename) 24: else

25: result = OclUndefined and result.oclIsTypeOf(OclVoid) 26: endif

Fig. 2. Specification of oclAsType() with up- and down-cast semantics

question is whether all types of the object are considered when re-typing (in the sense that the actual parameter typename must conform to one of the set of types only), or whether the unique type that is determined by evaluating the preceding OCL expression is considered. We here take the more general case that all of the object’s types are considered when re-typing.

Variable argClassifier keeps the unique Classifier that has the name specified by actual parameter typename (lines 7 – 11). Note that it is already required by well-formedness constraints that Classifier names can be uniquely determined by their names. Additionally, we have to assume that operation toString() can be applied to enumeration literals. We here use that oper- ation to generate Strings from enumeration literals that can in particular be compared with Classifier names of the referred UML user model (lines 10 and 16).

Variable selfClassifiers represents the set of Classifiers on level M1 whose names conform to the elements of selfTypes (lines 12 – 17). Note that we access the metalevel M2, as instances of Classifier have to be selected (lines 9 – 11 and 14 – 17). The condition expression in lines 19 – 26 then either succeeds (line 23) if the specified parameter type denoted by typename is a

«powertype»

OclType

*OclAny*

OclMessage

«datatype»

*Collection*

{complete, overlapping}

(...)

«datatype» Real

«datatype» OrderedSet

T

«datatype» Sequence

T

This ellipsis (...) indi- cates that all user-defined classifiers on level M1 are subtypes of OclAny

«enumeration» Boolean

«datatype» String

«datatype» Set

T

«datatype» Bag

T

«enumeration» OclState

«enumeration» OclVoid

T

T

«datatype» Integer

Fig. 3. OCL Standard Library Types Proposal

subtype or supertype of the object’s type(s). If not, the result is OclUndefined (line 25).

The constraint shown in Figure [2](#_bookmark12) could directly be applied in the OCL

2.0 proposal. Specifications for related operations such as oclIsTypeOf() and oclIsKindOf() can be formulated in a similar way. Unfortunately, two modeling levels have to be accessed, which is not desired in the OCL 2.0 proposal. Therefore, we are going to propose a different approach in the next section that finds a more elegant (and UML compliant) way to capture OCL types on modeling layer M1.

# OclType as a Powertype

So far, none of the approaches for modeling OclType has considered the *pow- ertype* concept. Powertype is a UML core concept and denotes a dependency relationship among a generalization [[5](#_bookmark21), Section 3.36]. A powertype is modeled as part of a class diagram residing on level M1. Basically, a powertype is a user-defined metaelement whose instances are classes of the user model. Thus, a powertype gives access to specialized types as instances.

Figure [3](#_bookmark13) shows OclType as a powertype for OclAny. This means that the subtypes of OclAny are *instances* of OclType, e.g., the type Integer (not the Integer values) is an instance of OclType. The powertype concept is thus ideal to represent types as instances of OclType within the OCL Standard Library type system.

* 1. *Re-typing Objects in the Powertype Approach*

We assume that the operations allInstances() and conformsTo(t:OclType) are defined for the powertype OclType, similar to the metalevel operations in the current UML 1.5 standard. Note that we allow to apply operation allInstances() to OclType itself (as a “class operation”) as well as to its instances, i.e., types like OclState, OclVoid, and user-defined classes. How- ever, when applying that operation to types with an infinite value set, such as Integer or Real, the operation returns OclUndefined, as required in the OCL 2.0 proposal [[4](#_bookmark20), Section 6.2.1]. As operation allInstances() is defined for powertype OclType, it is no longer necessary for type OclAny, such that we propose to remove it from that type.

The result of OclAny’s operation oclAsType() can then be specified by the postcondition shown in Figure [4](#_bookmark15) (again, we provide up- and down-cast semantics and consider the set of types of an object).

1: context OclAny::oclAsType(typename:OclType) : OclAny 2: post: if OclType.allInstances()

3: ->select(t:OclType | self.oclIsTypeOf(t))

4: ->exists(t:OclType | typename.conformsTo(t)

or t.conformsTo(typename)) then 5: result = self and result.oclIsTypeOf(typename)

6: else

7: result = OclUndefined and result.oclIsTypeOf(OclVoid) 8: endif

Fig. 4. Specification of oclAsType() with the Powertype Approach

Accessing the metalevel is still necessary, e.g., for evaluating the result of operation allInstances() and conformsTo(). But now all operations that need to access the metalevel are defined for OclType, i.e., we have a clear separation of concerns.

* 1. *Other Enhancements to the OCL Standard Library*

Generally, the diagrams in the OCL 2.0 proposal could use more of the stan- dard graphical model elements provided by the UML. As a case study, Figure [3](#_bookmark13) therefore shows some more enhancements in the graphical notation of the OCL Standard Library type system, i.e.,

* + - An ellipsis is notated by (...) in UML [[5](#_bookmark21), Section 6.8.1.1]). It is used to indicate the existence of additional children (or: subtypes) that are not shown in a particular diagram.

In our case, we can make use of the ellipsis to indicate *already existing subtypes* of OclAny, as OCL expressions only make sense over a given UML

model with user-defined classes. We added a comment to that ellipsis to explain that it represents all user-defined classes of the referred UML user model.

* + - Type OclAny is shown in italics to indicate that it is an abstract class. The same applies to the parameterized type Collection(T).
    - Additional standard constraint keywords complete and overlapping clas- sify the generalization among OclAny.

The annotation complete means that all of the children of OclAny have been declared (whether they are shown or not shown) [[5](#_bookmark21), Section 3.50]. Note that this does not contradict to the ellipsis used to represent the user- defined classes; we again argue that OCL expressions only make sense over a given UML model.

The annotation overlapping indicates that an instance may be a direct or indirect instance of two or more of the children. Due to OCL type OclVoid, the regarded generalization is overlapping.

* + - Predefined OCL types are annotated by standard stereotypes to further indicate whether they are data types or enumerations.
    - Type OclModelElement is omitted, as we consider it as being superfluous in the current OCL 2.0 proposal.
    - Finally, parameterized class OrderedSet(T) is introduced, as this kind of collection type has recently been added to OCL 2.0.

It has only additionally to be considered whether the undefined value OclUndefined needs to be an instance of OclType as well. But so far, we do not see a necessity to have this.

# Conclusion

In this article, we focused on deficiencies in the latest OCL 2.0 proposal

w.r.t. the definition of operations that reason about type conformance among OCL types, e.g., oclAsType() and oclIsTypeOf(). We identified inconsis- tencies among operation signatures and re-typing semantics and provided a specification of oclAsType() that allows for both up- and down-casts.

Furthermore, we propose to use the powertype concept to model the types accessible in OCL expressions on the modeling layer M1. In this context, a number of other open issues could be solved, e.g., we propose that operation allInstances() should no longer belong to OclAny, as it does not make sense to apply it to objects and datatype values. Instead, it naturally belongs to the powertype OclType.

Industrial acceptance of OCL will be hard to achieve with an OCL 2.0

standard that has obvious definiencies already in the fundamental definitions such as the discussed operations. We are aware that there are a lot of other open issues to discuss in the OCL 2.0 proposal, in particular w.r.t. the two semantics definitions that are currently neither consistent nor complete.

We think that a completed, consistent OCL semantics is very important, such that we build upon the formal OCL semantics of M. Richters’ mathemat- ical object model [[6](#_bookmark22)]. While our research concerning the role of Statecharts in OCL is basically completed [[3](#_bookmark19)], we currently focus on a formal semantics of OCL messages.

# Acknowledgement

This work receives funding by the Deutsche Forschungsgemeinschaft in the course of the Special Research Initiative 614 “Self-optimizing Concepts and Structures in Mechanical Engineering”.

# References

1. T. Baar and R. H¨ahnle. An Integrated Metamodel for OCL Types. In R. France et al., editors, *Proc. of OOPSLA 2000, Workshop Refactoring the UML: In Search of the Core*, Minneapolis, Minnesota, USA, 2000.
2. A. D. Brucker and B. Wolff. HOL-OCL: Experiences, Consequences and Design Choices. In *UML 2002 – 5th International Conference, Dresden, Germany, September/October 2002*, volume 2460 of *LNCS*, pages 196–211. Springer, 2002.
3. S. Flake and W. Mu¨ller. Semantics of State-Oriented Expressions in the Object Constraint Language. In *15th Internat’l Conf. on Software Engineering and Knowledge Engineering (SEKE 2003), San Francisco, USA*, pages 142–149. Knowledge Systems Institute, USA, July 2003.
4. A. Ivner, J. Ho¨gstro¨m, S. Johnston, D. Knox, and P. Rivett. Response to the UML2.0 OCL RfP, Version 1.6 (Submitters: Boldsoft, Rational, IONA, Adaptive Ltd., et al.). OMG Document ad/03-01-07, January 2003.
5. OMG, Object Management Group. Unified Modeling Language 1.5 Specification. OMG Document formal/03-03-01, March 2003.
6. M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universit¨at Bremen, Bremen, Germany, 2001.