Enhancing the Message Concept of the Object Constraint Language

Stephan Flake

C-LAB, Paderborn University, Fuerstenallee 11, 33102 Paderborn, Germany

E-mail: `flake@c-lab.de`

Abstract

The textual Object Constraint Language (OCL) is an official part of the Unified Modeling Language (UML). A new concept in the recently adopted OCL version 2.0 is the notion of OCL messages that enable modelers to put restrictions on messages sent.

However, this concept shows some shortcomings with respect to the existing OCL language concepts. On the one hand, the proposed syntax does not quite conform to the established notation of OCL. On the other hand, the formal OCL semantics still lacks an integration of OCL messsages.

This article reviews the syntax and semantics of OCL messages and presents a new approach to better integrate this concept with the rest of OCL 2.0.

1 Introduction

The Object Constraint Language (OCL) is a declarative expression language that enables modelers to formulate constraints in the context of a given UML user model [12]. Recently, OCL version 2.0 has been adopted by the Object Management Group (OMG) as part of the new UML 2.0 standard [9]. OCL is mainly used to specify invariants attached to classes and pre- and postconditions of operations, but OCL is also applied to formulate well-formedness rules in the metamodel definition of the official UML specification.

As an application example, assume that we have a model with classes `Machine` and `Buffer` and an association between these classes (see Figure 1). The following invariant requires that each instance of class `Machine` has at least one associated buffer:

```
context Machine
inv: self.buffers->notEmpty()
```

The class name that follows the `context` keyword specifies the class for which the following expression should hold. The keyword `self` refers to each object of the context class. Attributes, operations, and associations can be accessed by dot notation, e.g., `self.buffers` results in a (possibly empty) set of instances of `Buffer`. The arrow notation indicates that a collection of objects is manipulated by one of the predefined OCL collection operations. For example, operation `notEmpty()` returns true when the accessed set is not empty.

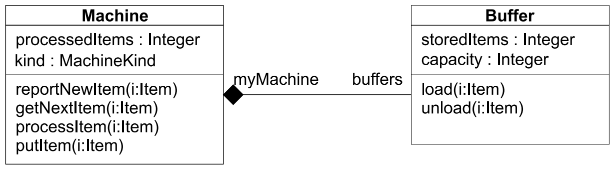

Figure 1. Sample Class Diagram

In operation postconditions, modelers can put restrictions on messages sent. For example, consider the following requirement for buffer objects.

> *During execution of operation* `load()`, *a reporting message has to be sent to the machine to which the buffer belongs.*

A corresponding operation specification in OCL may look like this:

```
context Buffer::load(i:Item)
pre: storedItems < capacity
post: myMachine^reportNewItem(i)
```

The expression `myMachine^reportNewItem(i)` is a boolean expression that results in true when at least one message `reportNewItem(i)` has been sent to the associated machine object during operation execution.

However, the syntax and semantics of such message specifications have some significant shortcomings that are discussed in the remainder of this article. In particular, we consider the following issues as being problematic, both in terms of usage by UML modelers as well as w.r.t. the underlying semantics:

```
msg.hasReturned() : Boolean
  -- Returns true if the message denotes an operation call and if the invoked operation has already returned.

msg.result() : OclAny           -- Note: the actual return type is the return type of the invoked operation.
  -- Returns the result of the invoked operation if the message denotes an operation call and the invoked
  -- operation has already returned. Otherwise the operation returns OclUndefined.

msg.isSignalSent() : Boolean
  -- Returns true if the message represents a signal.

msg.isOperationCall() : Boolean
  -- Returns true if the message represents an operation call.
```

Figure 2. Operations on OCL Messages

- Each OCL message expression requires the explicit specification of a destination object.
- The syntax used for message operators, i.e., `^` and `^^`, is unnecessarily cryptic.
- The expressions for common message specifications are unnecessarily complex.
- Concerning the evaluation of OCL expressions, a message specification can refer to a destination object that might has already been destroyed.
- The OCL semantics description becomes quite complex due to the required data structure that stores the history of messages sent (cf. [4]).

The remainder of this article is structured as follows. In Section 2, we outline the concept of OCL messages as defined in OCL 2.0. Section 3 then discusses the identified shortcomings of the current definition of OCL messages. In Section 4 we then present our redefinition of the OCL message concept. Related work is outlined in Section 5, and Section 6 concludes the article.

2 OCL Messages

Based on the work by Kleppe and Warmer [6, 7], OCL messages have been newly introduced in OCL 2.0 to specify behavioral constraints over messages sent by objects. An OCL message refers to a particular signal sent or a (synchronous or asynchronous) operation called. While signals sent are asynchronous and the calling object simply continues its execution, synchronous operation calls make the invoking operation wait for a return value. An asynchronous operation call is like sending a signal, such that a potential return value is simply discarded. For more details about messaging actions we refer to the action semantics of UML [10, Section 2.24].

2.1 Syntax

The parameterized type `OclMessage(T)` is part of the OCL Standard Library, where the template parameter `T` refers to an operation or signal. A concrete `OclMessage` type is therefore described by (a) the referred operation or signal and (b) all formal parameters of the referred operation or all attributes of the referred signal, respectively. Four operations on OCL messages are predefined (see Figure 2).

OCL messages are obtained by the message operator `^^` that is attached to a *destination object*. For example, the expression `myMachine^^reportNewItem(i)` results in the sequence of messages `reportNewItem(i)` that have been sent to the object determined by `myMachine` during execution of the considered operation – recall that the considered expression must have been specified in an operation postcondition. Each element of the resulting sequence is an instance of type `OclMessage(T)`. For example, the exact type of the OCL expression `myMachine^^reportNewItem(i)` is `Sequence(OclMessage(reportNewItem(i:Item)))`.

One can make use of so-called *unspecified values* to indicate that an actual parameter does not need to have a specific value. Unspecified values are denoted by question marks, e.g., `myMachine^^reportNewItem(?:Item)`.

The *hasSent operator* `^` can be used to check whether a message has been sent. This has already been illustrated for the OCL expression `myMachine^reportNewItem(i)` in Section 1. Note that this operator can easily be derived from the message operator `^^`. Each expression of the form `destObj^msgName(parameters)` can be replaced by `destObj^^msgName(parameters)->notEmpty()`.

2.2 Semantics

The OCL 2.0 specification provides two semantic descriptions. The first semantics is a *metamodel-based* approach, i.e., the semantics of an OCL expression is given by associating each value defined in the semantic domain (i.e., the `Values` package) with a type defined in the metamodel (i.e., the `AbstractSyntax` package), and by associating each evaluation with an expression of the abstract syntax. Given an overall snapshot of the running system, these associations allow to yield a unique value for each OCL expression, which is the result value of OCL expression evaluation. Secondly, a *formal semantics* is defined based on the mathematical notion of an *object model*. This is discussed in more detail in Section 3.

A semantic integration of OCL messages with the rest of OCL is currently only available in the metamodel-based semantics [9, Section 10.2]. In this context, the `Values` package has a class for *local snapshots*. Local snapshots are kept as an ordered list which allows to access the *history* of the values of an object, e.g., attribute values at the beginning of an operation execution. In particular, local snapshots keep track of the sequence of messages an object has sent. However, there is no dynamic semantics, such that it is undefined which snapshots of a running system are actually stored, i.e., it is not clear *how* local snapshots are created and handled at runtime. Moreover, there is no official formal semantics of OCL messages available, which motivated our previous work [4].

3 Review of the Formal OCL Semantics

The *formal OCL 2.0 semantics* is defined by a set-theoretic approach called *object model* based on work by Richters [11]. The object model of OCL 2.0 is a tuple

$$\mathcal{M} = \langle \; \mathit{CLASS}, \mathit{ATT}, \mathit{OP}, \mathit{ASSOC}, \prec, \mathit{associates}, \mathit{roles}, \mathit{multiplicities} \; \rangle$$

with a set $CLASS$ of classes, a set ATT of attributes, a set OP of operations, a set $ASSOC$ of associations, a generalization hierarchy $\prec$ over classes, and functions $associates$, $roles$, and $multiplicities$ that give for each $as \in ASSOC$ its dedicated classes, the classes' role names, and multiplicities, respectively.

In the remainder of this article, we call an instantiation of an object model a *system*. A system changes over time, i.e., the (number of) objects, their attribute values, and other characteristics change during system execution. *System states* keep corresponding information to be able to evaluate OCL expressions. In OCL 2.0, a system state $\sigma(\mathcal{M})$ is formally defined as a triple $\sigma(\mathcal{M}) = \langle \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{ASSOC} \rangle$ with the set Σ_{CLASS} of currently existing objects, the set Σ_{ATT} of attribute values of the objects, and the set Σ_{ASSOC} of currently established links that connect the objects.

However, the information stored in this system state triple is not sufficient to evaluate expressions that reason about messages sent; messages are not considered at all in the formal model so far. We therefore added appropriate components to the object model and system states. Thus, the resulting *extended system state* is a tuple

$$\sigma(\mathcal{M}) = \langle \; \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{ASSOC}, \Sigma_{CONF}, \Sigma_{currentOp}, \Sigma_{currentOpParam}, \Sigma_{sentMsg}, \Sigma_{sentMsgParam}, \Sigma_{inputQueue}, \Sigma_{inputQueueParam} \; \rangle$$

that now additionally comprises

- the set Σ_{CONF} of active state configurations over active objects (see [5] for more details about OCL and UML State Diagrams),
- for each currently existing object, the set $\Sigma_{currentOp}$ of its currently executed operations,
- for each current operation execution, the set $\Sigma_{sentMsg}$ of sent messages, and
- for each currently existing object, the set $\Sigma_{inputQueue}$ of received messages that are still waiting to be dispatched.

Parameter values of executed operations and sent/received messages are kept in separate structures for technical reasons. The resulting structure of system states has become comparatively complex, but all listed components are in fact necessary in order to (a) provide a formal semantics for OCL messages and (b) give a dynamic semantics of OCL. We defined a dynamic OCL semantics by means of *traces*, i.e., sequences of system states, based on a set of *noteworthy changes* that identify all changes relevant for the evaluation of OCL constraints [4]. While that work is primarily intended to complete the formal semantics of the OCL 2.0 standard, this article reviews and enhances the concept of OCL messages.

4 Our Approach

To motivate our approach, we first review a message specification found in the OCL 2.0 specification [9, Section 7.7.2]:

```
context Person::giveSalary(amount : Integer)
post: let message : OclMessage = company^getMoney(amount)
      in
      message.hasReturned()   -- getMoney was sent and returned
      and
      message.result() = true  -- getMoney returned true
```

Unfortunately, this postcondition has a type mismatch; the expression `company^getMoney(amount)` does not return an OCL message, but rather a boolean value, as the hasSent operator is applied. We therefore revise the postcondition and use the message operator `^^` to extract the corresponding message(s) sent. Additionally, we adjust the type of variable `messages` to be a sequence of messages:

```
context Person::giveSalary(amount : Integer)
post: let messages : Sequence(OclMessage) =
                                   company^^getMoney(amount)
      in
      messages->forAll(msg:OclMessage | msg.hasReturned())
      and
      messages->forAll(msg:OclMessage | msg.result() = true)
```

The postcondition above now requires that *all* messages `getMoney(amount)` sent to object `company` have already returned with result value `true`. But this does not have the originally intended meaning any more. Instead, the actual requirement is that (a) all messages `getMoney(amount)` have already returned and (b) *exactly once* the return result is `true`. Returning `true` stands for getting the money from the company – and the money must not be granted more than once by the company. The correct specification is then as follows.

```
context Person::giveSalary(amount : Integer)
post: let messages : Sequence(OclMessage) =
                                   company^^getMoney(amount)
      in
      messages->forAll(msg:OclMessage | msg.hasReturned())
      and
      messages->select(msg:OclMessage | msg.result() = true)
              ->size() = 1
```

The example already exhibits some of the shortcomings of the current approach in OCL 2.0. Firstly, the syntax ^ and ^^ for message specifications easily leads to errors in the specification. The two different operators are very similar in appearance but have totally different results; one denotes a boolean expression, while the other results in a sequence of OCL messages. Secondly, a unique destination object has to be specified together with each referred message. Instead, one might often be interested in a specific message sent to different object (e.g., broadcasts). In such cases a message specification becomes rather complex.

Assume now that a person can have more than one employer, such that `self.companies` refers to the set of `Company` objects that represent the person's employers. In the context of an operation `collectBonus()` that determines the total amount of bonus payments, we require that at least one message `getBonus()` is sent to each employer and that all these messages have returned at the time of postcondition evaluation.

```
context Person::collectBonus()
post: let messages : Sequence(OclMessage) =
          self.companies->collect(c:Company |
                            c^^getBonus(self.maritalStatus))
      in
      messages->forAll(msg:OclMessage | msg.hasReturned())
      and
      self.companies->forAll(c:Company |
                c^^getBonus(self.maritalStatus)->notEmpty())
```

We can directly express the desired, i.e., flattened, sequence of all messages sent to all associated companies with the predefined `collect()` operation.[1] But still, the expression is quite cumbersome to formulate and relatively difficult to understand. For this kind of specification, one would prefer to simply specify the message name without the need to refer to an explicit destination object each time.

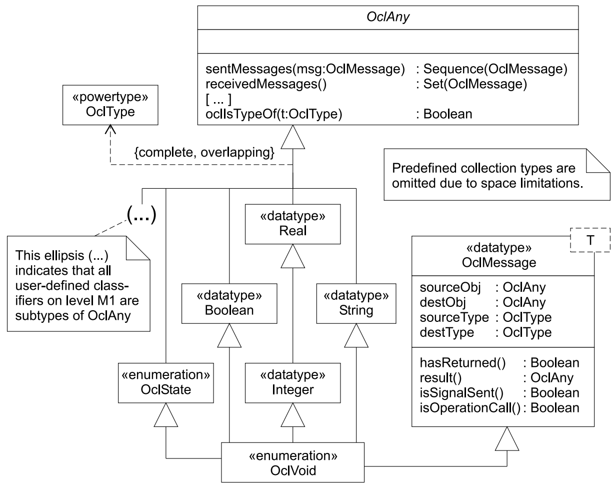

Figure 3. Redefined Type OclMessage

4.1 Redefinition of OCL Messages

We suggest a different way to obtain a sequence of messages sent. We define new attributes for type `OclMessage`, i.e., attributes that refer to the source and destination object and to the types of the source and destination object (the latter attributes are for technical purposes as explained in the remainder). The resulting type definition is illustrated in Figure 3. Note that we make use of an enhanced OCL type system that allows to refer to OCL types on the UML user level M1 [3].

With a new operation named `sentMessages()` defined for the general type `OclAny`, which is the supertype of all OCL types, the `collectBonus()` example can then be specified as follows.

```
context Person::collectBonus()
post: let messages : Sequence(OclMessage) =
          self.sentMessages(getBonus(self.maritalStatus))
      in
      messages->forAll(msg:OclMessage | msg.hasReturned())
      and
      self.companies = messages.destObj->asSet()
```

Firstly, the cryptic and error-prone message operator ^^ can simply be replaced by a new operation on the general supertype `OclAny` as demonstrated above. Secondly, we

[1] Note that one might also assume that the resulting structure is nested, i.e., the result is of type `Set(Sequence(OclMessage))`, but operation collect automatically returns the flattened collection. However, as nesting of collections is necessary in many other cases, OCL 2.0 now provides a corresponding operation `collectNested()`.

avoid the explicit specification of a destination object in front of a message declaration. Instead, the new attribute `destObj` for OCL messages leads to a simplified, yet better understandable, formulation of OCL messages, especially in the case of broadcasted and multicasted messages. Moreover, this notation is in line with the established OCL syntax that uses only dot/arrow notation for navigation and applies operation names with arguments. Note here that our formal semantics of OCL messages [4] has only marginally to be adjusted w.r.t. the formal definition of the message tuples.

4.2 Message Destination Objects

A more serious problem arises when a message destination object does not exist anymore at the time of postcondition evaluation. Explicitly referring to such an object in a postcondition does then not make sense. The constraint cannot be evaluated, as the message specification results in an undefined expression. Nevertheless, a message to that object might actually have been sent to that object.

In contrast, our approach captures this situation. We can separately check the value of the attribute `destObj`. If it has the predefined OCL value `OclUndefined` (i.e., the only instance of type `OclVoid`, see Figure 3), the destination object is no longer existing. In fact, this even gives us the chance to explicitly require that certain message destination objects must still exist or must have been destroyed.

Additionally, the attributes referring to the source and destination *types* of messages allow to restrict the kind of participants of message exchanges. For example, we can require that messages `getBonus()` may only be sent to objects of type `Company`:

```
context Person::collectBonus()
post: let messages : Sequence(OclMessage) =
          self.sentMessages(getBonus(?:Status))
      in
      messages->forAll(msg:OclMessage |
                       msg.destType().oclIsTypeOf(Company))
```

Similarly, we can restrict receptions of messages in preconditions or even invariants. Accessing received messages is discussed in the next section.

4.3 Received Messages

While it is already possible to reason about messages *sent* in OCL 2.0, there is currently no means to access and reason about the messages *received* by an object.

At this point we have to discuss whether it is really necessary to formulate constraints on received messages with OCL. First of all, there are already other UML means to specify behavioral constraints over received messages, e.g., Protocol State Diagrams and Sequence Diagrams. However, it might be necessary to specify *invariants* over received messages that go beyond the specification means of State Diagrams, e.g., to define a priority scheme after reception of two different signals or to specify a more complex reaction after reception of an external signal. This issue is of particular interest in the domain of embedded real-time systems, where additional real-time properties have to be considered. But as UML and OCL are intended for general purpose modeling, there is no inherent notion of time, such that a dedicated UML profile should be considered in this case.

However, causal relationships concerning requests and acknowledgments might still need to be modeled and are of interest in the context of OCL specifications as well. This soon leads to temporal extensions of OCL that have already been proposed, e.g., in [1]. Unfortunately, such extensions make use of temporal logics to provide a formal semantics, which is definitely out of the scope of the OCL standard in its current state. We therefore stick to our first-order predicate semantics presented in [4]. We make use of the system state component $\Sigma_{inputQueue}$ to provide a semantics for our new operation `receivedMessages()` on type `OclAny` (cf. Figure 3).

Such a semantics is given in the form of a denotational interpretation function $I[[op]](\langle\sigma(\mathcal{M}),\beta\rangle)$ for an operation signature $op = (\omega : t_c \times t_1 \times \ldots \times t_n \to t) \in OP$ over a system state $\sigma(\mathcal{M})$ and an OCL-specific variable assignment β.[2] In operation signatures, ω is the operation name, c is the class for which the operation ω is defined, and t_c is the respective OCL type. $t_1, \ldots, t_n$ are the parameter types, and t is the result type of the operation.

We define the semantics of OCL message operation `receivedMessages()` over a system state $\sigma(\mathcal{M})$ and variable assignment β in the context of a given currently existing object $\underline{oid} \in \Sigma_{CLASS,c}$. The semantics of operation `receivedMessages()` is formally notated by

$$I[[receivedMessages()]](\langle\sigma(\mathcal{M}),\beta\rangle)(\underline{oid}).$$

The evaluation result is simply determined by the set $\sigma_{inputQueue,c}(\underline{oid})$, where $\sigma_{inputQueue,c}$ is a function over the set $\Sigma_{inputQueue}$ of incoming messages that are waiting to be dispatched. We only have to add the corresponding parameter values stored in $\Sigma_{inputQueueParam,c}$ to each element of $\sigma_{inputQueue,c}(\underline{oid})$. However, a detailed formalization is omitted here only for the sake of concision.

We decided that operation `receivedMessages()` returns a *set* of messages rather than a sequence, as the latter would require some kind of ordering predicate on incoming messages. But the order of incoming events is a well-known semantic variation point in UML. One can use the built-in operation `sortedBy()` to induce a sequence of messages if this is desired.

[2]Variable assignment β determines values for OCL-specific variables, i.e., local variables defined in `let`-expressions and iterator variables used in collection expressions.

5 Related Work

A good overview of approaches that define a semantics for (parts of) different versions of OCL is given in [2]. However, our own recent work [4] so far provides the only formal integration of OCL messages into the rest of OCL.

We know of only one other proposal to enhance the notion of OCL messages, i.e., the work by Kyas and de Boer [8]. They distinguish between local and global specifications for OCL constraints. Additional built-in types such as `OclEvent` with attributes `sender`, `receiver`, and an event kind (send, receive, invoke, return) are introduced. Using these types, new predefined attributes `localHistory` and `globalHistory` are presented that allow to access the sequence of sent and received messages. This approach also avoids the rather cryptic message operators `^` and `^^`. However, an integration into the semantical OCL framework (either the metamodel or the formal semantics) is not described.

In contrast, we can provide a formal definition of our enhancement of the OCL message concept based on the formal notions of our previously proposed *extended object model* and *extended system states*.

6 Conclusion

We identified shortcomings in the syntactical and semantical definition of OCL messages and proposed corresponding enhancements that keep compliant to the established notation and language concepts. Our changes in the definition of OCL messages affect other parts of OCL, e.g., the type system that has to be adjusted to be able to refer to OCL types at the UML user level M1. The OCL community is aware of this problem in the current type system and we expect that this issue will be resolved in the context of the finalization of OCL 2.0.

The formal semantics of OCL 2.0 is relatively complex. However, the underlying logic is still restricted to pure first-order predicate logic, i.e., temporal logic is so far not applied. It should nevertheless be investigated in the future whether temporal logic should be considered both for direct application in user-defined OCL constraints and as an approach to formulate the underlying formal OCL semantics. This could, e.g., avoid the explicit storage of the history of messages sent.

Although there are already some OCL tools available (see http://www.klasse.nl/ocl for an overview), there is currently no tool available that supports OCL messages. We hope that our work can influence the development of appropriate tools in the near future.

Acknowledgments. The work described in this article receives funding by the DFG project GRASP within the DFG Priority Programme 1064 'Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen'.

References

[1] J. Bradfield, J. Küster Filipe, and P. Stevens. Enriching OCL Using Observational Mu-Calculus. In R.-D. Kutsche and H. Weber, editors, *5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002), April 2002, Grenoble, France*, volume 2306 of *LNCS*, pages 203–217. Springer, 2002.

[2] M. V. Cengarle and A. Knapp. OCL 1.4/1.5 vs. OCL 2.0 Expressions: Formal Semantics and Expressiveness. *Software and Systems Modeling (SoSyM), Springer*, 3(1):9–30, March 2004.

[3] S. Flake. OclType – A Type or Metatype? In *UML 2003 Workshop "OCL 2.0 – Industry Standard or Scientific Playground?", October 2003, San Francisco, CA, USA*, Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam, The Netherlands, 2003.

[4] S. Flake. Towards the Completion of the Formal Semantics of OCL 2.0. In V. Estivill-Castro, editor, *27th Australasian Computer Science Conference (ACSC 2004), Dunedin, New Zealand*, volume 26 of *Australian Computer Science Communications*, pages 73–82, January 2004.

[5] S. Flake and W. Mueller. Semantics of State-Oriented Expressions in the Object Constraint Language. In *15th International Conference on Software Engineering and Knowledge Engineering (SEKE 2003), July 2003, San Francisco Bay, CA, USA*, pages 142–149. Knowledge Systems Institute, 2003.

[6] A. Kleppe and J. Warmer. Extending OCL to Include Actions. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. York, UK*, volume 1939 of *LNCS*, pages 440–450. Springer, 2000.

[7] A. Kleppe and J. Warmer. The Semantics of the OCL Action Clause. In T. Clark and J. Warmer, editors, *Object Modeling with the OCL: The Rationale behind the Object Constraint Language*, pages 213–227. Springer, 2002.

[8] M. Kyas and F. de Boer. On Message Specifications in OCL. In *UML 2003 Workshop on Compositional Verification of UML Models, October 2003, San Francisco, CA, USA*, Electronic Notes in Theoretical Computer Science. Elsevier, Amsterdam, The Netherlands, 2003.

[9] OMG, Object Management Group. UML 2.0 OCL Final Adopted Specification. OMG Document ptc/03-10-14, October 2003. ftp://ftp.omg.org/pub/docs/ptc/03-10-14.pdf.

[10] OMG, Object Management Group. Unified Modeling Language 1.5 Specification. OMG Document formal/03-03-01, March 2003.

[11] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universität Bremen, Bremen, Germany, 2001.

[12] J. Warmer and A. Kleppe. *The Object Constraint Language – Getting Your Models Ready for MDA*. Addison-Wesley, 2003.