Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 102 (2004) 77–97
www.elsevier.com/locate/entcs

Formal Semantics of OCL Messages

Stephan Flake and Wolfgang Mueller[1]

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

Abstract

The latest OCL 2.0 proposal provides two semantic descriptions, i.e., a metamodel-based semantics that uses UML itself to associate the semantic domain with the language concepts and a *formal semantics* based on a set-theoretic approach called *object model*. Unfortunately, these two semantics are currently neither consistent nor complete, as (a) the formal semantics does not consider the newly introduced concept of OCL messages and (b) both semantics lack an integration of Statecharts and a semantic definition of state-related operations.
This article focuses on a formal semantics for OCL messages as a foundation for consistency among the two OCL semantics. We extend object models and present an extended definition of a *system state* that comprises all relevant information to be able to evaluate OCL expressions also w.r.t. OCL messages.

Keywords: OCL 2.0, Extended Object Model

1 Introduction

The adopted OCL 2.0 proposal follows two approaches to define the semantics of OCL. First, a semantics is described using UML itself by a metamodel-based approach [6, Chapter 5]. Different packages are defined that represent the abstract syntax on the metamodel layer M2 and the semantic domain on UML modeling layer M1. A separate package then relates these packages by associations between elements of the semantic domain and elements of the abstract syntax. For example, each value of the semantic domain is associated with a type of the abstract syntax. Similarly, evaluations of OCL expressions are associated with corresponding expressions in the abstract syntax. A particular

[1] Email: {flake,wolfgang}@c-lab.de

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2004.09.009

evaluation of an OCL expression is performed over a given system snapshot, such that a unique value is yielded as a result.

Additionally, a formal semantics is defined by a set-theoretic mathematical approach called *object model* [6, App. A] based on work by M. Richters [10]. An object model is a tuple

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

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 association $as \in ASSOC$ its dedicated classes, their role names, and multiplicities, respectively.

In the remainder of this article, a particular instantiation of an object model is called a *system*. A system is in different states as it changes over time, i.e., the (number of) objects, their attribute values, and other characteristics change during execution of the system. In the OCL 2.0 proposal, a *system state* $\sigma(\mathcal{M}) \stackrel{def}{=} \langle \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{ASSOC} \rangle$ is formally defined as a triple consisting of a set Σ_{CLASS} of currently existing objects, a set Σ_{ATT} of attribute values for the objects, and a set Σ_{ASSOC} of currently established links that connect the objects.

However, the formal semantics provided in the OCL 2.0 proposal is not complete, as it is not possible with the information given by a system state to reason about currently activated Statechart states or messages that have been sent. Thus, it is not possible to provide a formal semantics for state-related operations and operations on OCL messages.

In our work, we focus on the completion of the formal semantics based on object models. In a previous article, we already integrated Statecharts to OCL by a notion of state configurations and gave a formal semantics for state-related operation `oclInState(statename:OclState)` [5]. This article now further extends that work and focuses on the formalization of OCL messages.

The remainder of this article is structured as follows. In Section 2, we briefly explain the concept of OCL messages. Section 3 extends the formal definition of object models by providing additional components to also capture OCL messages. Section 4 then provides a corresponding semantics by introducing interpretation functions for OCL message-related operators and operations. Section 5 concludes this article.

2 OCL Messages

The concept of OCL messages has been newly introduced in the OCL 2.0 proposal to specify behavioral constraints over messages sent by objects. It is based on work presented in [7,8]. Basically, an OCL message refers to a signal sent or a (synchronous or asynchronous) operation called. While signals sent are asynchronous by nature and the calling object simply continues its execution, synchronous operation calls make the invoking operation wait for a return value. In contrast, an asynchronous operation call is like sending a signal, such that a potential return value is simply discarded. For more details about messaging actions, see the action semantics of UML 1.5 [9, Section 2.24]. Note here that the UML action semantics also define *broadcast signal actions*, while a corresponding kind of OCL message is not yet defined.

The concept of OCL messages enables modelers to specify postconditions that require that specific signals must have been sent, operations must have been called, or operations must have been completely executed and returned.

2.1 Syntax

A predefined parameterized type `OclMessage(T)` is now part of the OCL type system within the OCL Standard Library, where the template parameter `T` denotes 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. The operations defined for type `OclMessage(T)` are listed in Figure 1. Note that it is only allowed to obtain and make use of OCL messages in operation postconditions.

OCL messages are obtained by the message operator `^^` that is attached to a *target object*. For example, the OCL expression `targetObj^^setValue(17)` results in the sequence of messages `setValue(17)` that have been sent to the object determined by `targetObj` 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 type of OCL expression `targetObj^^setValue(17)` is

```
Sequence(OclMessage(setValue(i:Integer))) .
```

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., `targetObj^^setValue(?:Integer)`. Parameter types can be omitted in OCL message expressions, but note that

```
 1: hasReturned() : Boolean
 2:   -- Returns true iff the template parameter denotes an operation
 3:   -- and the invoked operation has already returned.
 4:
 5: result() : <<The return type of the invoked operation>>
 6:   -- Returns the result of the invoked operation iff the template
 7:   -- parameter denotes an operation and the invoked operation
 8:   -- has already returned. Otherwise OclUndefined is returned.
 9:
10: isSignalSent() : Boolean
11:   -- Returns true iff the template parameter represents a signal.
12:
13: isOperationCall() : Boolean
14:   -- Returns true iff the template parameter represents an
15:   -- operation call.
```

Fig. 1. Operations for OCL Messages

they might be necessary in order to refer to the correct operation when the operation is specified more than once with different parameter types.

To check whether a message has been sent, the *hasSent operator* `^` can be used, e.g., the expression `targetObj^setValue(17)` results in true iff a message `setValue(17)` has been sent to `targetObj` during execution of the considered operation. More examples can be found in [6, Section 2.7.3].

2.2 Semantics

The semantics of OCL messages is currently only defined in the metamodel-based semantics [6, Section 5.2]. In this context, the so-called `Values` package that represents the semantic domain has a class for *local snapshots*. A local snapshot is an element of the semantic domain that stores the values that are necessary for later reference. Local snapshots are kept as an ordered list that 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 and the sequence of messages that the object has received during execution of an operation.

A formal semantics of OCL messages has not yet been defined, i.e., the two semantics for OCL are currently inconsistent. To overcome this deficiency, we therefore extend the formal approach of object models in the next section.

2.3 Example

As an application example, we review a postcondition found in the OCL 2.0 proposal [6, Section 2.7.2]:

```
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 call returned true
```

Unfortunately, this postcondition is not quite correctly specified; the expression `company^getMoney(amount)` does not return an OCL message, but rather a boolean value, as the hasSent operator is applied. Instead, the message operator `^^` must be used to extract the corresponding message(s) sent:

```
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)
```

Note that we now have to reason about a *sequence of OCL messages*. The postcondition above requires that all messages `getMoney(amount)` sent to object `company` have already returned with result value `true`. If we want to restrict that the message `getMoney(amount)` is sent exactly once, we have to add an additional condition as follows:

```
context Person::giveSalary(amount : Integer)
post: let messages : Sequence(OclMessage) = company^^getMoney(amount)
          in
          messages->size() = 1                -- getMoney was sent once
          and
          messages->first().hasReturned()     -- getMoney returned
          and
          messages->first().result() = true   -- the call returned true
```

3 Extended Object Models

In the OCL 2.0 proposal, the formal definition of object models currently lacks of components for Statechart states and OCL messages and we therefore define an extension of object models called *extended object models*. In particular, the following concepts have to be newly introduced:

- signal receptions for classes with corresponding well-formedness rules,
- Statecharts and their association with active classes,
- a formal definition of state configurations, and
- the extension of the formal descriptor of a class.

Additionally, the following information has to be added to system states to be able to evaluate OCL expressions that make use of state-related and OCL message-related operations:

- state configurations of all currently existing active objects,
- currently executed operations, and
- for each currently executed operation, all messages sent so far.

Subsection 3.1 explains the syntactical elements of extended object models. In Subsection 3.2, we then present an extended version of system states. This extension enables us to give a semantics to message-related operations that could so far not formally be defined.

3.1 Syntax

In the remainder of this article, let $\mathcal{A}$ be an alphabet, $\mathcal{N}$ be a set of names over $\mathcal{A}^+$, and T a set of types. In particular, $T = T_B \cup T_E \cup T_C \cup T_S$ comprises

- a set of basic standard library types T_B, i.e., *Integer*, *Real*, *Boolean*, and *String*,
- a set T_E of user-defined enumeration types,
- a set T_C of user-defined classes, $c \in CLASS$, and
- a set of special types $T_S \stackrel{def}{=} \{OclVoid, OclState, OclAny\}$.

We call the value set $I_{TYPE}(t)$ (or simply $I(t)$ when the context is clear) represented by a type t the *type domain*. For convenience, we presume that `OclUndefined` (in the following denoted by symbol $\perp$) is included in each type domain, such that we have, e.g., $I(OclVoid) \stackrel{def}{=} \{\perp\}$ and

$$I(OclAny) = \Big(\bigcup_{t \in T_B \cup T_E \cup T_C} I(t) \Big) \cup \{\perp\}.$$

Furthermore, let $c \in CLASS$ be a class and $t_c \in T_C$ be the type of class c.[2] Each class c is associated with a set ATT_c of attributes that describe characteristics of their objects. An attribute has a name $a \in \mathcal{N}$ and a type $t \in T$ that specifies the domain of attribute values. A class c is also associated with a set OP_c of operations and a set SIG_c of signals (in UML, signals handled by a class are specified by so-called *receptions* [9, Section 3.26.6]).

We define *Extended Object Models* by the tuple

$$\mathcal{M} \stackrel{def}{=} \langle\ CLASS, ATT, OP, paramKind, isQuery, SIG, SC, \\ ASSOC, \prec, \prec_{sig}, associates, roles, multiplicities\ \rangle$$

[2] Each class $c \in CLASS$ induces an object type $t_c \in T$ that has the same name as the class. The difference between c and t_c is that we have the special value $\perp \in I(t_c)$ for all $c \in CLASS$.

with

- a set $CLASS = ACTIVE \cup PASSIVE$ of active and passive classes,
- a set ATT of attributes, $ATT = \bigcup_{c \in CLASS} ATT_c$,
- a set OP of operations, $OP = \bigcup_{c \in CLASS} OP_c$,
- a function $paramKind : CLASS \times OP \times \mathbb{N} \to \{in, inout, out\}$ that gives for each operation parameter its parameter kind (cf. [9, Section 2.5.2.31]),
- a function $isQuery : CLASS \times OP \to Boolean$ that determines whether an operation is a query operation without side effects or not (cf. [9, Section 2.5.2.7]),
- a set SIG of signals, $SIG \supseteq \bigcup_{c \in CLASS} SIG_c$,
- a set SC of Statecharts, $SC = \bigcup_{c \in ACTIVE} SC_c$,
- a set $ASSOC$ of associations between classes,
- generalization hierarchies $\prec$ for classes and $\prec_{sig}$ for signals, and
- functions *associates*, *roles*, and *multiplicities* that define a mapping for each element in $ASSOC$ to the participating classes, their corresponding role names, and multiplicities, respectively.

Note that we do not further describe the tuple components of extended object models here. For more details on sets $CLASS$, ATT, OP, and $ASSOC$, readers are referred to the corresponding sources [6,10]. We also omit the formal syntax definitions for signals and Statecharts and refer to [5] for further details.

The set of characteristics defined in a class together with its inherited characteristics is called the *full descriptor of a class*. More formally, the full descriptor of a class $c \in CLASS$ is a tuple

$$FD_c \stackrel{def}{=} \langle\, ATT_c^*, OP_c^*, SIG_c^*, SC_c, navEnds^*(c) \,\rangle$$

containing the complete sets of attributes, operations, signals, navigable role names, and – in the case of an active class – the associated Statechart. For example, the complete set of attributes of a class c is defined by

$$ATT_c^* \stackrel{def}{=} ATT_c \cup \bigcup_{c' \in parents(c)} ATT_{c'},$$

where $parents(c)$ denotes the set of (transitive) superclasses of c. The complete sets OP_c^*, SIG_c^*, and $navEnds^*(c)$ of operations, signals, and navigable role names are defined correspondingly.

3.2 System State

The domain of a class $c \in CLASS$ is the set of objects of this class and all of its child classes. Objects are referred to by object identifiers that are unique in the context of the whole system.

The set of object identifiers of a class $c \in CLASS$ is defined by an infinite set $oid(c) \stackrel{def}{=} \{objId_1, objId_2, \ldots\}$. The domain of a class $c \in CLASS$ is defined as

$$I_{CLASS}(c) \stackrel{def}{=} \bigcup_{c' \in CLASS \text{ with } c' \prec c \vee c'=c} oid(c').$$

For technical purposes, we also define $I_{CLASS} \stackrel{def}{=} \bigcup_{c \in CLASS} oid(c)$.

When a particular instantiation of an extended object model (i.e., a system) is executed, the number of instantiated objects, their attribute values, Statechart configurations, and other characteristics will change over time. As pointed out earlier, the current notion of a system state with only three components is not sufficient to be able to evaluate OCL expressions that make use of state-related operations and OCL messages. Additionally, we need information about currently activated states, operations that have been called and signals sent, currently executed operations, etc. In this context, we adopt ideas of [11] to formalize currently executed operations and define functions to capture the required additional information.

Formally, a *system state* for an extended object model $\mathcal{M}$ is a tuple

$$\sigma(\mathcal{M}) \stackrel{def}{=} \langle\ \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{ASSOC}, \Sigma_{CONF}, \Sigma_{currentOp}, \Sigma_{currentOpParam}, \Sigma_{sentMsg}, \Sigma_{sentMsgParam}\ \rangle .$$

In the remainder of this subsection, we explain the components of system states in more detail, but note that Σ_{CLASS}, Σ_{ATT}, and Σ_{ASSOC} are already defined in [6,10].

(1) $\Sigma_{CLASS} \stackrel{def}{=} \bigcup_{c \in CLASS} \Sigma_{CLASS,c}$. The finite sets $\Sigma_{CLASS,c}$ contain all objects of a class $c \in CLASS$ existing in the system state, i.e.,

$$\Sigma_{CLASS,c} \subseteq oid(c) \subseteq I_{CLASS}(c).$$

Furthermore, we define sets $\Sigma_{ACTIVE,c}$ for active and $\Sigma_{PASSIVE,c}$ for passive classes correspondingly.

Note that – in contrast to the current formal OCL semantics – we notationally distinguish between object identifiers $objId \in oid(c)$ and currently existing objects $\underline{objId} \in \Sigma_{CLASS,c}$, i.e., we use additional underlines to emphasize the fact that we refer to a currently existing object.

This differentiation is of help for the definition of the semantics of OCL messages in Section 4.

(2) The current attribute values are kept in set Σ_{ATT}. It is the union of functions $\sigma_{ATT,a} : \Sigma_{CLASS,c} \rightarrow I(t)$, where $a \in ATT_c^*$ and t is the type specified for a. Each function $\sigma_{ATT,a}$ assigns a value to a certain attribute of each object of a given class $c \in CLASS$.

(3) $\Sigma_{ASSOC} \stackrel{def}{=} \bigcup_{as \in ASSOC} \Sigma_{ASSOC,as}$ comprises the finite sets $\Sigma_{ASSOC,as}$ that contain links that connect objects. We refer to [6,10] for detailed information about links, i.e., elements of $I_{ASSOC}(as)$, and formalization of multiplicity specifications.

(4) The current Statechart configurations are kept by

$$\sigma_{CONF} \stackrel{def}{=} \bigcup_{c \in ACTIVE} \left\{ \sigma_{CONF,c} : \Sigma_{ACTIVE,c} \rightarrow I_{SC}(c) \right\} .$$

Each function $\sigma_{CONF,c}$ assigns a complete state configuration comprising all activated (sub)states to each object of a given class $c \in ACTIVE$. Set $I_{SC}(c)$ denotes the possible state configurations of the Statechart SC_c associated with active class c. A formal definition of state configurations can be found in [5].

Additional runtime information has to be taken into account to be able to evaluate expressions that access OCL messages. This mainly concerns the currently executed operations and the histories of signals and messages sent. This relates to the *local snapshots* defined in the metamodel-based semantics for OCL 2.0 [6, Section 5.2].

3.2.1 Currently Executed Operations

Let $\mathcal{ID}$ be an infinite enumerable set, e.g., $\mathcal{ID} = \mathbb{N}$, and let $OpStatus \stackrel{def}{=} \{executing, returning\}$. At the starting point of an operation execution, a unique identifier $opId \in \mathcal{ID}$ is associated with the current operation execution. Thus, an operation execution can uniquely be identified by a given object $\underline{objId} \in \Sigma_{CLASS,c}$, an operation signature $op \in OP_c^*$, and an operation identifier $opId \in \mathcal{ID}$. The set of currently executed operations is defined by

$$\Sigma_{currentOp} \stackrel{def}{=} \bigcup_{c \in CLASS} \left\{ \sigma_{currentOp,c} : \Sigma_{CLASS,c} \times OP_c^* \rightarrow \mathcal{P}(I_{CLASS} \times OP \times \mathcal{ID} \times \mathcal{ID} \times OpStatus) \right\} .$$

Each function $\sigma_{currentOp,c}$ gives a set of tuples of the form $\langle sourceId, sourceOp, sourceOpId, opId, status \rangle$ that uniquely identify all currently executed operations for a given object and operation name. Elements $sourceId$, $sourceOp$,

and $sourceOpId$ refer to the operation execution that originally invoked the considered operation op with identifier $opId$. These elements are necessary to have a reference for returning a potential result value after termination of an operation execution. We require that the associated operation identifier $opId$ must not change until the execution of that operation terminates.

A flag $status \in OpStatus$ indicates the current status of operation execution. Compared to the messaging actions specified in UML 1.5, we here omit statuses *ready* and *complete* [9, Section 2.19.2.3], as they are currently not necessary in the context of OCL.

Actual parameter values of executed operations are kept in $\Sigma_{currentOpParam}$.

$$\Sigma_{currentOpParam} \stackrel{def}{=} \bigcup_{c \in CLASS} \left\{ \begin{array}{l} \sigma_{currentOpParam,c} : \Sigma_{CLASS,c} \times OP_c^* \times \mathcal{ID} \\ \qquad \rightarrow I^?(t_1) \times \ldots \times I^?(t_n) \times I^?(t) \end{array} \right\}.$$

For all $t \in T$, we define $I^?(t) \stackrel{def}{=} I(t) \cup \{?\}$. Symbol ? denotes the *unassigned status* of a value. This symbol must not be mixed up with the *undefined value* denoted by $\perp$ and is also *different from the String literal '?'*.

A function $\sigma_{currentOpParam,c}$ gives the actual parameter values of the currently executed operations for a given object, operation signature, and operation execution identifier. For each $c \in CLASS$, we define $\sigma_{currentOpParam,c}$ as follows, where $op = (\omega : t_c \times t_1 \times \ldots \times t_n \rightarrow t) \in OP_c^*$:

$$\sigma_{currentOpParam,c}(\underline{objId}, op, opId) \mapsto$$
$$\begin{cases} \langle val_1, ..., val_n, returnVal \rangle, & \text{if } \langle srcId, srcOp, srcOpId, opId, executing \rangle \\ & \quad \in \sigma_{currentOp,c}(\underline{objId}, op) \\ & \quad \vee \langle srcId, srcOp, srcOpId, opId, returning \rangle \\ & \quad \in \sigma_{currentOp,c}(\underline{objId}, op) \\ \varnothing, & \text{otherwise.} \end{cases}$$

In the definition above, $val_i \in I^?(t_i)$ denotes a value defined for type $t_i \in T$, $1 \leq i \leq n$. For a parameter at position i with $paramKind(c, op, i) \in \{in, inout\}$, the corresponding value val_i is predetermined by the operation call. Parameters with $paramKind(c, op, i) = out$ carry the unassigned value ? until the end of operation execution. The return value $returnVal \in I^?(t)$ is also kept unassigned until the operation terminates. We require that all parameter values do not change until operation termination.

When the status of operation execution changes from *executing* to *returning*, the parameters of kind *inout* and *out* as well as the return value $returnVal$

are updated and get a value $\neq ?$.[3] If an operation is not returning a result, the result type t of operation op is $OclVoid$. In that case, we set $returnVal = \bot$ when the operation terminates. Note that these updates only have an effect for *synchronous* operation calls, as result values of asynchronous operation calls are discarded according to the UML specification.

3.2.2 Messages Sent

To be able to evaluate OCL expressions that use the message operator ^^, we have to store the *history of messages sent* for each executed operation. For each object $\underline{objId} \in \Sigma_{CLASS,c}$ and each of its currently executed operations op with identifier $opId$, we define a function $\sigma_{sentMsg,c}(\underline{objId}, op, opId)$ that gives the set of messages sent together with their corresponding target objects.

When a message is sent from an execution of operation op with identifier $opId$ to a target object with identifier $targetId$, that target object must actually exist (otherwise we could not refer to it), but it may already have been destroyed when the execution of operation op terminates. This is the reason why we cannot use the set Σ_{CLASS} as the base set for target objects, as that set only keeps currently existing objects. Instead, the signature of function $\sigma_{sentMsg,c}$ has to use the general set I_{CLASS} of *target object identifiers*.

We define the set of messages sent by

$$\Sigma_{sentMsg} \stackrel{def}{=} \bigcup_{c \in CLASS} \left\{ \begin{array}{l} \sigma_{sentMsg,c} : \Sigma_{CLASS,c} \times OP_c^* \times \mathcal{ID} \rightarrow \\ \qquad \mathcal{P}(I_{CLASS} \times (SIG \cup OP) \times \mathcal{ID}) \end{array} \right\}.$$

The set $\mathcal{ID}$ in the value set $\mathcal{P}(I_{CLASS} \times (SIG \cup OP) \times \mathcal{ID})$ is used to refer to the correct message identifier when returning a value for synchronous operation calls. It would be sufficient to have an identifier that is unique in the context of the source object, e.g., named $\mathcal{ID}_{sourceId}$, but we here simply reuse set $\mathcal{ID}$ for the sake of concision.

An element $\langle targetId, msg, callId \rangle \in \sigma_{sentMsg,c}(\underline{objId}, op, opId)$ denotes that a message with signature msg and call identifier $callId$ has been sent from the object $\underline{objId}$ to an object with identifier $targetId$ as part of the operation execution with signature op and identifier $opId$.

We only have to keep information about messages sent during the corresponding invoking operation execution. Before and after operation execution,

[3] Note that rules for such updates are part of the dynamic semantics. We here have to assume that updates are correctly performed and have the desired effect.

we simply set $\sigma_{sentMsg,c}(\underline{objId}, op, opId) = \varnothing$. More formally, we have

$$\sigma_{sentMsg,c}(\underline{objId}, op, opId) \mapsto \begin{cases} \{\ \langle targetId, msg, callId\rangle\ \}, & \text{if } \langle srcId, srcOp, srcOpId, opId, executing\rangle \\ & \quad \in \sigma_{currentOp,c}(\underline{objId}, op) \\ & \vee \langle srcId, srcOp, srcOpId, opId, returning\rangle \\ & \quad \in \sigma_{currentOp,c}(\underline{objId}, op) \\ \varnothing, & \text{otherwise.} \end{cases}$$

Additionally, we have to store the actual parameter values of each message sent. We therefore define $\Sigma_{sentMsgParam} \stackrel{def}{=}$

$$\bigcup_{c \in CLASS} \{\ \sigma_{sentMsgParam,c} : \Sigma_{CLASS,c} \times OP_c^* \times \mathcal{ID} \times I_{CLASS} \times (SIG \cup OP) \times \mathcal{ID} \to I^?(t_1) \times \ldots \times I^?(t_n) \times I^?(t) \ \}.$$

The number n and the types t_i, $1 \leq i \leq n$, are determined by the formal parameters of the corresponding message signature, i.e., either a signal $sig = (\omega : t_c \times t_1 \times \ldots \times t_n) \in SIG_c^*$ or an operation $op = (\omega : t_c \times t_1 \times \ldots \times t_n \to t) \in OP_c^*$.

Each function $\sigma_{sentMsgParam,c}$ is defined by

$$\sigma_{sentMsgParam,c}(\underline{objId}, op, opId, targetId, msg, callId) \mapsto \begin{cases} \langle val_1, \ldots, val_n, returnVal\rangle, & \text{if } \langle targetId, msg, callId\rangle \in \\ & \quad \sigma_{sentMsg,c}(\underline{objId}, op, opId) \\ \varnothing, & \text{otherwise.} \end{cases}$$

The values $val_i \in I^?(t_i)$ document the parameter values of the message sent. We set all parameters of kind *out* and the return variable $returnVal$ to ? by default, i.e., these parameter values are left unassigned until they are calculated. Basically, they are only relevant for synchronous operation calls, where values $\neq$? are assigned after termination of the called operation. Again, note that potential results are discarded anyway for asynchronous operation calls. For signals sent, the domain of return type t is set to $I^?(OclVoid)$ by default and the return value simply remains unassigned.

Help Sets and Functions.

In the remainder of this article, we need some help sets and functions. These are basically subsets of $\Sigma_{sentMsg}$ and $\Sigma_{sentMsgParam}$ and sub-functions of $\sigma_{sentMsg,c}$ and $\sigma_{sentMsgParam,c}$, respectively. As their formal definitions are straight-forward, we omit them here for the sake of brevity.

Signals sent during execution of an operation are kept in set $\Sigma_{sentSig}$. Within this set, functions $\sigma_{sentSig,c}$ return the history of signals sent. Actual parameter values are kept in set $\Sigma_{sentSigParam}$ with functions $\sigma_{sentSigParam,c}$.

Operations called are kept in set $\Sigma_{calledOp}$. We make use of functions $\sigma_{calledOp,c}$ that return the history of operations called. Set $\Sigma_{calledOpParam}$ keeps the actual parameter values of called operations, and functions $\sigma_{calledOpParam,c}$ are used to access the actual parameter values of operations called.

To further distinguish synchronous and asynchronous operation calls, sets $\Sigma_{calledSynchOp}$ and $\Sigma_{calledAsynchOp}$ are employed. Within each set, we have functions $\sigma_{calledSynchOp,c}$ and $\sigma_{calledAsynchOp,c}$ that return the history of called synchronous and asynchronous operations for a given operation execution. Actual parameter values are kept in sets $\Sigma_{calledSynchOpParam}$ and $\Sigma_{calledAsynchOpParam}$ and are accessed by functions $\sigma_{calledSynchOpParam,c}$ and $\sigma_{calledAsynchOpParam,c}$.

We now have all necessary components to be able to evaluate general OCL expressions, i.e., also those that access OCL messages.

4 Semantics of OCL Messages

First, we formally define the domain of type $OclMessage$ by

$$I(OclMessage) \stackrel{def}{=} \bigcup_{c\in CLASS, op\in OP_c^*} I(OclMessage(op)) \cup \bigcup_{c\in CLASS, sig\in SIG_c^*} I(OclMessage(sig)),$$

where the set $I(OclMessage(op))$ for a given operation $op = (\omega : t_c \times t_1 \times \ldots \times t_n \rightarrow t) \in OP_c^*$ is defined as follows:

$$I(OclMessage(op)) = \mathcal{ID} \times I_{CLASS} \times I(t_1) \times \ldots \times I(t_n).$$

Set $\mathcal{ID}$ refers to the unique call identifiers ($callId$) of sent messages. Set I_{CLASS} is used to keep the object identifier of the target object to which the message is sent.

The formal definition of set $I(OclMessage(sig))$ for a signal $sig = (\omega : t_c \times t_1 \times \ldots \times t_n) \in SIG_c^*$ is very similar, i.e.,

$$I(OclMessage(sig)) = \mathcal{ID} \times I_{CLASS} \times I(t_1) \times \ldots \times I(t_n).$$

We are now able to give a syntax for postcondition expressions w.r.t. OCL message operators and a corresponding semantics in the next subsection. A semantics of operations on OCL messages is then given in Subsection 4.2.

4.1 OCL Message Operators

We here focus on the formalization of the more general *message operator* `^^`, as the *hasSent operator* `^` can easily be derived as follows. Given a target object expression `targetExpr` and an OCL message declaration `msg(msgArgs)`, we can substitute the OCL expression `targetExpr^msg(msgArgs)` by

```
targetExpr^^msg(msgArgs)->size() > 0 .
```

Note that this identity is not quite correctly treated in the OCL 2.0 proposal, as the definition of `OclMessageExpCS` says that the number of messages sent to the target object is exactly $= 1$ (instead of > 0) [6, Section 4.3].

Syntax. The basic syntactical elements of OCL expressions are defined by a so-called *data signature* $\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$ [6, Section A.2.8], where

- $T_{\mathcal{M}}$ is the set of *type expressions* $T_{\text{Expr}}(t)$ for types $t \in T_B \cup T_E \cup T_C \cup T_S$ [6, Section A.2.5],
- $\leq$ is a type hierarchy over $T_{\mathcal{M}}$ [6, Section A.2.7], and
- $\Omega_{\mathcal{M}}$ is the set of operation signatures, $\Omega_{\mathcal{M}} = \Omega_{T_{\mathcal{M}}} \cup \Omega_B \cup \Omega_E \cup \Omega_C \cup \Omega_S$.

The formal syntax of general valid OCL expressions is then inductively defined, such that more complex expressions are recursively built from simpler ones. The syntax of OCL expressions is given by the set

$$\text{Expr} \stackrel{def}{=} \bigcup_{t \in T_{\mathcal{M}}} \text{Expr}_t$$

and an additional function to capture free variables. The set Post-Expr of valid OCL postcondition expressions is defined in the same way as Expr, but with additional rules for allowing operation $oclIsNew()$, operator $@pre$, and a predefined result variable named `result` [6, Section A.3.2.2].

Additionally, the following rule viii. introduces a new kind of postcondition expression w.r.t. OCL messages. Note here that we also have to consider signals for message expressions. We therefore make use of set $\Psi_{\mathcal{M}}$ to refer to the set of signals defined in an instantiation of an object model $\mathcal{M}$.

viii. if (a) $e_{target} \in \text{Post-Expr}_t$, and
(b) either $(\omega : t_c \times t_1 \times \ldots \times t_n \to t) \in \Omega_{\mathcal{M}}$
or $(\omega : t_c \times t_1 \times \ldots \times t_n) \in \Psi_{\mathcal{M}}$, and
(c) $e_i \in \text{Post-Expr}_{t_i}$ for all $i \in \{1, \ldots, n\}$,
then $e_{target}\hat{}\hat{}\omega(e_1, \ldots, e_n) \in \text{Post-Expr}_t$
as well as $e_{target}\hat{}\omega(e_1, \ldots, e_n) \in \text{Post-Expr}_t$.
This maps into `OclMessageExp` in the abstract syntax of the OCL 2.0 proposal.

Semantics. Generally, the semantics of expressions is defined in the context of a given environment $\tau = \langle \sigma(\mathcal{M}), \beta \rangle$ with a system state $\sigma(\mathcal{M})$ and a variable assignment $\beta : Var_t \to I(t)$. A variable assignment β maps variable names to values [6, Section A.3.1.1]. In the following, let Env be the set of environments $\tau = \langle \sigma(\mathcal{M}), \beta \rangle$.

While the semantics of an OCL expression e is usually defined by a function $I[[e]] : Env \to I(t)$, we have to consider two environments in the case of operation postconditions, i.e., the environments τ_{pre} (at the beginning of operation execution) and τ_{post} (at time of termination). Thus, the interpretation function for expressions e specified in postconditions becomes $I[[e]] : Env \times Env \to I(t)$.

We now define the semantics of OCL message operators over environments $(\tau_{pre}, \tau_{post})$ in the context of a given object $\underline{objId} \in \Sigma_{CLASS,c}$ and an executed operation with signature $op \in OP_c^*$ and identifier $opId$ (implicitly, we assume that the operation execution has just terminated).

First, we define a help set $MSG_{e_{target}\hat{}\hat{}\omega(e_1,\ldots,e_n)}$ that keeps all relevant messages sent.

$$
\begin{aligned}
&MSG_{e_{target}\hat{}\hat{}\omega(e_1,\ldots,e_n)} \stackrel{def}{=} \\
&\quad \Big\{ \langle callId, eVal_{target}, v_1, \ldots, v_n \rangle \;\mid\; \exists c' \in CLASS : \\
&\qquad eVal_{target} = I[[e_{target}]](\tau_{pre}, \tau_{post}) \in I_{CLASS}(c') \setminus \{\bot\} \\
&\qquad \wedge\ \forall i \in \{1, \ldots, n\} : eVal_i = I[[e_i]](\tau_{pre}, \tau_{post}) \in I^?(t_i) \setminus \{\bot\} \\
&\qquad \wedge\ \forall i \in \{1, \ldots, n\} : (eVal_i \neq ?\ \Rightarrow\ eVal_i = v_i) \\
&\qquad \wedge\ \Big(\exists msg = (\omega : t_{c'} \times t_1 \times \ldots \times t_n \to t) \in OP_{c'}^* \\
&\qquad\quad \vee\ \exists msg = (\omega : t_{c'} \times t_1 \times \ldots \times t_n) \in SIG_{c'}^* \Big), \text{ such that} \\
&\qquad\quad \langle eVal_{target}, msg, callId \rangle \in \sigma_{sentMsg,c}(\underline{objId}, op, opId) \\
&\qquad\quad \wedge\ \exists anyVal \in I^?(OclAny) : \langle v_1, \ldots, v_n, anyVal \rangle \in \\
&\qquad\qquad \sigma_{sentMsgParam,c}(\underline{objId}, op, opId, eVal_{target}, msg, callId) \Big\} .
\end{aligned}
$$

Informally, the elements $\langle callId, eVal_{target}, v_1, \ldots, v_n\rangle$ of this set are determined as follows. The target object identifier $eVal_{target}$ is evaluated from e_{target} and must be well-defined in the sense that it is different from $\bot$. Similarly, all evaluations $eVal_i$ of the parameter expressions e_i must be well-defined. For consistency reasons, those $eVal_i$ that evaluate to an actual value (i.e., a value $\neq ?$) must be equal to v_i.

Furthermore, there must be a message signature msg, such that the triple $\langle eVal_{target}, msg, callId\rangle$ represents a message sent from object $\underline{objId}$ within the regarded operation execution. And finally, the values v_i must be equal to the actual parameter values that are stored (and potentially updated) for the investigated messages sent. Variable $anyVal$ is only introduced for technical reasons to allow for an arbitrary value of the return value.

In the following, let m be the number of elements in $MSG_{e_{target}\hat{}\hat{}\omega(e_1,\ldots,e_n)}$. For each $i \in \{1,\ldots,m\}$, let $x_i = \langle callId_i, eVal_{target}, v_{1,i}, \ldots, v_{n,i}\rangle$ be a distinct element of set $MSG_{e_{target}\hat{}\hat{}\omega(e_1,\ldots,e_n)}$ with $callId_j < callId_{j+1}$ for all $j \in \{1,\ldots,m-1\}$.

Because of the unique call identifiers of messages sent, the latter condition induces an order on the elements $x_i \in MSG_{e_{target}\hat{}\hat{}\omega(e_1,\ldots,e_n)}$, such that we can define the corresponding *sequence of messages sent* as follows, using double angle brackets to denote a sequence of elements.

$$I[[e_{target}\hat{}\hat{}\omega(e_1,\ldots,e_n)]](\tau_{pre},\tau_{post}) \stackrel{def}{=} \langle\langle x_1, x_2, \ldots, x_m\rangle\rangle$$

If at least one $I[[e_i]](\tau_{pre},\tau_{post})$, $1 \leq i \leq n$, evaluates to $\bot$, the whole expression evaluates to the empty sequence, as we have explicitly required $I[[e_i]](\tau_{pre},\tau_{post}) \in I^?(t) \setminus \{\bot\}$ in the definition of $MSG_{e_{target}\hat{}\hat{}\omega(e_1,\ldots,e_n)}$. Alternatively, as the OCL 2.0 proposal does not consider this issue, we can define a semantics that evaluates to $\bot$ in this case.

Furthermore, it is not clearly defined in the OCL 2.0 proposal whether the target object that is specified as part of the message expression must still exist at the time of checking the postcondition. In order not to loose generality, we think it should be allowed to also refer to objects that might have been destroyed while the operation was still executing. Consequently, we cannot assume that $I[[e_{target}]](\tau_{pre},\tau_{post})$ evaluates to an object $\underline{eVal}_{target}$ that still exists at the time of postcondition evaluation. Instead, we interpret $I[[e_{target}]](\tau_{pre},\tau_{post})$ as an *object identifier* $\in I_{CLASS}(c')$ only. To further indicate that we are only referring to an object identifier here, we do not underline $eVal_{target}$.

The meaning of $eVal_{target} = \bot$ is now that the object identifier $eVal_{target}$ is not defined w.r.t. the complete execution of the operation under consideration.

In this case, $I[[e_{target}\hat{}\hat{}\omega(e_1,\ldots,e_n)]](\tau_{pre},\tau_{post})$ results in the empty sequence.

4.2 OCL Message Operations

The signatures of the four predefined OCL message operations are

$$\begin{aligned}
&I_{hasReturned:OclMessage\to Boolean} &&: I(OclMessage) \to I(Boolean),\\
&I_{result:OclMessage\to OclAny} &&: I(OclMessage) \to I(OclAny),\\
&I_{isOperationCall:OclMessage\to Boolean} &&: I(OclMessage) \to I(Boolean), \text{ and}\\
&I_{isSignalSent:OclMessage\to Boolean} &&: I(OclMessage) \to I(Boolean).
\end{aligned}$$

As existing OCL syntax does not need to be adjusted for message operations, we here only have to define a semantics for message operations. Generally, the semantics of an operation $(\omega : t_c \times t_1 \times \ldots \times t_n \to t) \in OP_c^*$ is recursively defined by

$$\begin{aligned}
&I[[\omega(e_1,\ldots,e_n)]](\tau_{pre},\tau_{post}) \stackrel{def}{=}\\
&\quad I(\omega)(\tau_{post})\ \big(\ I[[e_1]](\tau_{pre},\tau_{post}),\ldots,I[[e_n]](\tau_{pre},\tau_{post})\ \big)\ .
\end{aligned}$$

We define the semantics of OCL message operations over environments (τ_{pre},τ_{post}) in the context of a given object $\underline{objId} \in \Sigma_{CLASS,c}$ and an executed operation with signature $op = (\omega : t_c \times t_1 \times \ldots \times t_n \to t) \in OP_c^*$ and identifier $opId$ (implicitly, we assume that the operation execution has just terminated).

4.2.1 Operations hasReturned() and result()

Note that operations `hasReturned()` and `result()` only make sense over synchronous operation calls, as results of asynchronous operation calls are discarded according to UML 1.5. We can therefore directly apply function $\sigma_{calledSynchOp,c}$ to check whether an OCL message $\langle callId, targetId, v_1,\ldots,v_n\rangle \in$

$I(OclMessage)$ has returned, i.e.,

$$I(hasReturned)(\tau_{post})()\ (\ \langle callId, targetId, v_1, \ldots, v_n\rangle\ ) \stackrel{def}{=}$$
$$\begin{cases} true, \text{ if } \exists msg \in OP: \\ \qquad \langle targetId, msg, callId\rangle \in \sigma_{calledSynchOp,c}(\underline{objId}, op, opId) \\ \qquad \wedge\, \sigma_{calledSynchOpParam,c}(\underline{objId}, op, opId, targetId, msg, callId) \\ \qquad\quad = \langle val_1, \ldots, val_n, returnVal\rangle, \text{ such that} \\ \qquad\qquad returnVal \neq\ ? \\ \qquad\qquad \wedge\, \forall i \in \{1, \ldots, n\} : (v_i \neq\ ?\ \Rightarrow\ val_i = v_i) \\ false, \text{ otherwise.} \end{cases}$$

Condition $returnVal \neq\ ?$ guarantees that the operation has returned, as that parameter value is updated to an element of $I(t)$ after the corresponding operation termination.

The semantics of operation `result()` is defined in a very similar way, as function $\sigma_{calledSynchOp,c}$ can also be applied to determine the result of a synchronous message call.

$$I[[result()]](\tau_{pre}, \tau_{post})\ (\ \langle callId, targetId, v_1, \ldots, v_n\rangle\ ) \stackrel{def}{=}$$
$$\begin{cases} returnVal, \text{ if } \exists msg \in OP: \\ \qquad \langle targetId, msg, callId\rangle \in \sigma_{calledSynchOp,c}(\underline{objId}, op, opId) \\ \qquad \wedge\, \sigma_{calledSynchOpParam,c}(\underline{objId}, op, opId, targetId, msg, callId) \\ \qquad\quad = \langle val_1, \ldots, val_n, returnVal\rangle, \text{ such that} \\ \qquad\qquad returnVal \neq\ ? \\ \qquad\qquad \wedge\, \forall i \in \{1, \ldots, n\} : (v_i \neq\ ?\ \Rightarrow\ val_i = v_i) \\ \bot, \qquad \text{otherwise.} \end{cases}$$

4.2.2 Operations isSignalSent() and isOperationCall()

The semantics of operations `isSignalSent()` and `isOperationCall()` are easily obtained based on the formal definition of operation `hasReturned()`.

The main difference is that functions $\sigma_{calledSynchOp,c}$ and $\sigma_{calledSynchOpParam,c}$ are replaced correspondingly, as we now have to consider synchronous and asynchronous operation calls for `isOperationCall()` and signals sent for operation `isSignalSent()`.

Furthermore, condition $returnVal \neq\ ?$ is not needed, as we do not investigate whether a message has returned yet. For the sake of brevity, we here

only provide the formal semantics of operation `isOperationCall()`:

$$I[[isOperationCall()]](\tau_{pre}, \tau_{post})\ (\ \langle callId, targetId, v_1, \ldots, v_n\rangle\ ) \stackrel{def}{=}$$
$$\begin{cases} true, \text{ if } \exists msg \in OP: \\ \qquad \langle targetId, msg, callId\rangle \in \sigma_{calledOp,c}(\underline{objId}, op, opId) \\ \qquad \wedge\, \sigma_{calledOpParam,c}(\underline{objId}, op, opId, targetId, msg, callId) \\ \qquad\quad = \langle val_1, \ldots, val_n, returnVal\rangle, \text{ such that} \\ \qquad\qquad \forall i \in \{1, \ldots, n\} : (v_i \neq\ ?\ \Rightarrow\ val_i = v_i) \\ false, \text{ otherwise.} \end{cases}$$

5 Conclusion and Outlook

Based upon our previous work that already captures Statecharts and state-related operations, we presented further extensions to object models and system states, such that a formal semantics for OCL messages and corresponding operators and operations could be given. This article is therefore to be seen as a direct contribution to the finalization process of OCL 2.0.

One important aspect in the formalization is that we identified the situation that a target object (i.e., an object to which a message was sent) might no longer exist at time of postcondition evaluation. In turn, when an asynchronous operation call is dispatched or a signal sent is consumed in a target object, the source object (i.e., the object to which the invoking operation belongs) might already be destroyed. It is therefore necessary to refer to *object identifiers* instead of "real" objects in the semantic definition of OCL messages.

We further had to extend the domain $I(t)$ of types $t \in T$ by an unassigned value to allow for symbol ? in message expressions, i.e., $I^{?}(t) = I(t) \cup \{?\}$. This maps to the `UnspecifiedValueExp` in the OCL metamodel. But note that we could also make use of that symbol for the unassigned status of return values prior to assigning an actual result value.

For OCL messages, we used explicit call identifiers to distinguish messages sent from source objects to target objects. When returning from a synchronous operation call, this identifier can be used to update the corresponding parameter values. This is an abstraction from the UML semantics that assumes that a specific reply object is generated and sent [9, Section 2.24].

The formal semantics of OCL 2.0 is now almost complete. What is still missing are formal definitions for def-clauses and operations on `OrderedSet`.

However, this is quite easy to achieve; operations defined for ordered sets are basically the same as for sequences, and def-clauses can directly be mapped to so-called `OclHelper` variables and operations. `OclHelper` variables and operations, in turn, are stereotyped attributes and operations of classifiers. Such variables and operations can be used in OCL expressions just like common attributes and operations. Thus, it only has to be ensured that no naming conflicts occur, while additional semantic issues do not occur.

One important remaining task is to complete the metamodel-based OCL semantics. First of all, Statechart states are still not considered at all in the metamodel-based OCL semantics. But also consistency among the two semantics should be reviewed.

Only few reports are currently available about the applicability of OCL in practice, e.g., [12]. But different publications of recent years indicate that there is a need for *temporal extensions* of OCL, e.g., [1,2,3,4,11]. We think that a dynamic semantics based upon system states as presented in this article is a suitable basis for defining a formal semantics of temporal OCL extensions. To demonstrate the applicability of this approach, a state-oriented temporal OCL extension has already been developed [5].

Acknowledgement

This work receives funding by the Deutsche Forschungsgemeinschaft in the course of the project GRASP within the DFG Priority Programme 1064 “Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen” and partial funding by the DFG Special Research Initiative 614 “Selbstoptimierende Systeme des Maschinenbaus”.

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. Cengarle and A. Knapp. Towards OCL/RT. In L.-H. Eriksson and P. Lindsay, editors, *Formal Methods – Getting IT Right*, volume 2391 of *LNCS*, pages 389–408. Springer, July 2002.

[3] S. Conrad and K. Turowski. Temporal OCL: Meeting Specifications Demands for Business Components. In *Unified Modeling Language: Systems Analysis, Design, and Development Issues*. IDEA Group Publishing, 2001.

[4] D. Distefano, J.-P. Katoen, and A. Rensink. On a Temporal Logic for Object-Based Systems. In S. Smith and C. Talcott, editors, *Proc. of FMOODS'2000 – Formal Methods for Open Object-Based Distributed Systems IV*, Stanford, CA, USA, September 2000. Kluwer Academic Publishers.

[5] S. Flake and W. Müller. Formal Semantics of Static and Temporal State-Oriented OCL Constraints. *Software and System Modeling (SoSyM), Springer*, 2(3), 2003. To appear. Online version available at http://link.springer.de under Digital Object Identifier 10.1007/s12270-003-0026-x.

[6] A. Ivner, J. Högström, 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.

[7] 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.

[8] 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.

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

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

[11] P. Ziemann and M. Gogolla. An Extension of OCL with Temporal Logic. In J. Jürjens, M. V. Cengarle, E. B. Fernandez, B. Rumpe, and R. Sandner, editors, *Critical Systems Development with UML*, pages 53–62. Technische Universität München, Institut für Informatik, 2002.

[12] S. Zschaler. Evaluation der Praxistauglichkeit von OCL-Spezifikationen. Master's thesis, Technical University of Dresden, Faculty of Computer Science, August 2002. (in German).