Electronic Notes in Theoretical Computer Science 102 (2004) 77–97 

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

Formal Semantics of OCL Messages

Stephan Flake and Wolfgang Mueller[1](#_bookmark0)

*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 seman- tics 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

# 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](#_bookmark15), Chapter 5]. Different packages are defined that represent the ab- stract syntax on the metamodel layer M2 and the semantic domain on UML modeling layer M1. A separate package then relates these packages by associ- ations 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 as- sociated with corresponding expressions in the abstract syntax. A particular

1 Email: *{*flake,wolfgang*}*@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.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](#_bookmark15), App. A] based on work by M. Richters [[10](#_bookmark20)]. An object model is a tuple

*M def* *CLASS, ATT, OP, ASSOC, ≺, associates, roles, multiplicities*

=

with a set *CLASS* of classes, a set *ATT* of attributes, a set *OP* of opera- tions, a set *ASSOC* of associations, a generalization hierarchy *≺* over classes, and functions *associates*, *roles*, and *multiplicities* that give for each associ- ation *as ∈ 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*

*def*

*σ*(*M*) = *⟨*Σ*CLASS,* Σ*ATT ,* Σ*ASSOC⟩* 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](#_bookmark16)]. 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](#_bookmark1), we briefly explain the concept of OCL messages. Section [3](#_bookmark3) extends the formal definition of object models by providing additional components to also cap- ture OCL messages. Section [4](#_bookmark8) then provides a corresponding semantics by introducing interpretation functions for OCL message-related operators and

operations. Section [5](#_bookmark10) concludes this article.

# 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](#_bookmark17),[8](#_bookmark18)]. 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](#_bookmark19), 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.

* 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](#_bookmark2). 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 op- eration – 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 ac- tual parameter does not need to have a specific value. Unspecified values are denoted by question marks, e.g., targetObj^^setValue(?:Integer). Pa- rameter 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](#_bookmark15), Section 2.7.3].

* 1. *Semantics*

The semantics of OCL messages is currently only defined in the metamodel- based semantics [[6](#_bookmark15), 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.

* 1. *Example*

As an application example, we review a postcondition found in the OCL 2.0 proposal [[6](#_bookmark15), 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 ex- pression company^getMoney(amount) does not return an OCL message, but rather a boolean value, as the hasSent operator is applied. Instead, the mes- sage 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

# 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](#_bookmark4) explains the syntactical elements of extended object models. In Subsection [3.2](#_bookmark6), 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.

* 1. *Syntax*

In the remainder of this article, let *A* be an alphabet, *N* be a set of names over *A*+, and *T* a set of types. In particular, *T* = *TB ∪ TE ∪ TC ∪ TS* comprises

* a set of basic standard library types *TB*, i.e., *Integer*, *Real*, *Boolean*, and

*String*,

* a set *TE* of user-defined enumeration types,
* a set *TC* of user-defined classes, *c ∈ CLASS*, and
* a set of special types *TS*

*def*

= *OclV oid, OclState, OclAny* .

*{ }*

We call the value set *ITY PE*(*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 *⊥*) is included in each type

domain, such that we have, e.g., *I*(*OclV oid*) = *{⊥}* and

*def*

*I*(*OclAny*) =

*t∈TB∪TE∪TC*

*I*(*t*) *∪ {⊥}.*

Furthermore, let *c ∈ CLASS* be a class and *tc ∈ TC* be the type of class

*c*. [2](#_bookmark5) Each class *c* is associated with a set *ATTc* of attributes that describe characteristics of their objects. An attribute has a name *a ∈ N* and a type *t ∈ T* that specifies the domain of attribute values. A class *c* is also associated with a set *OPc* of operations and a set *SIGc* of signals (in UML, signals handled by a class are specified by so-called *receptions* [[9](#_bookmark19), Section 3.26.6]).

We define *Extended Object Models* by the tuple

*M def* *CLASS, ATT, OP, paramKind, isQuery, SIG, SC,*

=

*ASSOC, ≺, ≺sig, associates, roles, multiplicities*

2 Each class *c ∈ CLASS* induces an object type *tc ∈ T* that has the same name as the class. The difference between *c* and *tc* is that we have the special value *⊥ ∈ I*(*tc*) for all *c ∈ CLASS*.

with

* a set *CLASS* = *ACTIV E ∪ P ASSIV E* of active and passive classes,
* a set *ATT* of attributes, *ATT* = *c∈CLASS ATTc*,
* a set *OP* of operations, *OP* = *c∈CLASS OPc*,
* a function *paramKind* : *CLASS × OP ×* N *→ {in, inout, out}* that gives for each operation parameter its parameter kind (cf. [[9](#_bookmark19), Section 2.5.2.31]),
* a function *isQuery* : *CLASS × OP → Boolean* that determines whether an operation is a query operation without side effects or not (cf. [[9](#_bookmark19), Section 2.5.2.7]),
* a set *SIG* of signals, *SIG ⊇* *c∈CLASS SIGc*,
* a set *SC* of Statecharts, *SC* = *c∈ACT IV E SCc*,
* a set *ASSOC* of associations between classes,
* generalization hierarchies *≺* for classes and *≺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](#_bookmark15),[10](#_bookmark20)]. We also omit the formal syntax definitions for signals and Statecharts and refer to [[5](#_bookmark16)] 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 ∈ CLASS* is a tuple

*FD def* *ATT∗, OP∗, SIG∗, SC , navEnds∗*(*c*)

*c*

=

*c*

*c*

*c*

*c*

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

*∗ def*

*ATTc* = *ATTc ∪*

*c'∈parents*(*c*)

*ATTc',*

where *parents*(*c*) denotes the set of (transitive) superclasses of *c*. The com- plete sets *OP∗*, *SIG∗*, and *navEnds∗*(*c*) of operations, signals, and navigable

*c c*

role names are defined correspondingly.

* 1. *System State*

The domain of a class *c ∈ 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 ∈ CLASS* is defined by an infinite

*def*

set *oid*(*c*) = *{objId*1*, objId*2*,.. .}*. The domain of a class *c ∈ CLASS* is

defined as

*ICLASS*

*def*

(*c*) =

*c'∈CLASS* with *c'≺c ∨ c'*=*c*

*oid*(*c'*)*.*

For technical purposes, we also define *ICLASS*

*def*

=

*c∈CLASS*

*oid*(*c*).

When a particular instantiation of an extended object model (i.e., a sys-

tem) 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 com- ponents is not sufficient to be able to evaluate OCL expressions that make use of state-related operations and OCL messages. Additionally, we need in- formation about currently activated states, operations that have been called and signals sent, currently executed operations, etc. In this context, we adopt ideas of [[11](#_bookmark21)] to formalize currently executed operations and define functions to capture the required additional information.

Formally, a *system state* for an extended object model *M* is a tuple

*σ*(*M*) *def* Σ

=

*CLASS*

*,* Σ*ATT*

*,* Σ*ASSOC*

*,* Σ*CONF ,*

Σ*currentOp,* Σ*currentOpP aram,* Σ*sentMsg,* Σ*sentMsgParam* *.*

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](#_bookmark15),[10](#_bookmark20)].

1. Σ*CLASS*

*def*

=

*c∈CLASS*

Σ*CLASS,c*

. The finite sets Σ

*CLASS,c*

contain all

objects of a class *c ∈ CLASS* existing in the system state, i.e.,

Σ*CLASS,c ⊆ oid*(*c*) *⊆ ICLASS*(*c*)*.*

Furthermore, we define sets Σ*ACT IV E,c* for active and Σ*PASSIV E,c* for pas- sive classes correspondingly.

Note that – in contrast to the current formal OCL semantics – we notationally distinguish between object identifiers *objId ∈ oid*(*c*) and currently existing objects *objId ∈* Σ*CLASS,c*, i.e., we use additional un- derlines 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](#_bookmark8).

1. The current attribute values are kept in set Σ*ATT* . It is the union of functions *σATT,a* : Σ*CLASS,c → I*(*t*), where *a ∈ ATT∗* and *t* is the type specified for *a*. Each function *σATT,a* assigns a value to a certain attribute of each object of a given class *c ∈ CLASS*.

*c*

1. Σ

*ASSOC*

*def*

=

*as∈ASSOC*

Σ*ASSOC,as*

comprises the finite sets Σ*ASSOC,as*

that

contain links that connect objects. We refer to [[6](#_bookmark15),[10](#_bookmark20)] for detailed infor-

mation about links, i.e., elements of *IASSOC*(*as*), and formalization of multiplicity specifications.

1. The current Statechart configurations are kept by

*σCONF*

*def*

=

*c∈ACT IV E*

*σCONF,c*

: Σ*ACT IV E,c*

*→ ISC*

(*c*) } *.*

Each function *σCONF,c* assigns a complete state configuration comprising all activated (sub)states to each object of a given class *c ∈ ACTIV E*. Set *ISC*(*c*) denotes the possible state configurations of the Statechart *SCc* associated with active class *c*. A formal definition of state configurations can be found in [[5](#_bookmark16)].

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](#_bookmark15), Section 5.2].

* + 1. *Currently Executed Operations*

Let *ID* be an infinite enumerable set, e.g., *ID* = N, and let *OpStatus*

*def*

=

*{executing, returning}*. At the starting point of an operation execution, a unique identifier *opId ∈ ID* is associated with the current operation exe- cution. Thus, an operation execution can uniquely be identified by a given object *objId ∈* Σ*CLASS,c*, an operation signature *op ∈ OP∗*, and an operation identifier *opId ∈ ID*. The set of currently executed operations is defined by

*c*

Σ*currentOp*

*def*

=

*σcurrentOp,c*

: Σ*CLASS,c*

*× OP∗ →*

*c*

*c∈CLASS*

*P*(*ICLASS × OP × ID × ID × OpStatus*) } *.*

Each function *σcurrentOp,c* gives a set of tuples of the form *⟨sourceId*, *sourceOp*, *sourceOpId*, *opId*, *status⟩* that uniquely identify all currently executed oper- ations 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 ∈ OpStatus* indicates the current status of operation execu- tion. Compared to the messaging actions specified in UML 1.5, we here omit statuses *ready* and *complete* [[9](#_bookmark19), 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 Σ*currentOpP aram*.

Σ*currentOpP aram*

*def*

=

*σcurrentOpP aram,c*

: Σ*CLASS,c*

*× OP∗ × ID*

*c*

*c∈CLASS*

For all *t ∈ T* , we define *I*?(*t*

*→ I*?(*t*1) *× ... × I*?(*tn*) *× I*?(*t*) } *.*

*I*(*t*) *∪ {*?*}*. Symbol ? denotes the *unassigned*

*def*

) =

*status* of a value. This symbol must not be mixed up with the *undeﬁned value*

denoted by *⊥* and is also *different from the String literal ’?’*.

A function *σcurrentOpP aram,c* gives the actual parameter values of the cur- rently executed operations for a given object, operation signature, and opera- tion execution identifier. For each *c ∈ CLASS*, we define *σcurrentOpP aram,c* as follows, where *op* = (*ω* : *tc × t*1 *× ... × tn → t*) *∈ OP∗*:

*c*

*σcurrentOpP aram,c*(*objId, op, opId*) *'→*

⎧

⎪⎨

⎪ *⟨val*1*, ..., valn, returnV al⟩,* if *⟨srcId, srcOp, srcOpId, opId, executing⟩*

⎪⎪⎩

*∈ σcurrentOp,c*(*objId, op*)

*∨ ⟨srcId, srcOp, srcOpId, opId, returning⟩*

*∈ σcurrentOp,c*(*objId, op*)

∅*,* otherwise*.*

In the definition above, *vali ∈ I*?(*ti*) denotes a value defined for type *ti ∈*

*T* , 1 *≤ i ≤ n*. For a parameter at position *i* with *paramKind*(*c, op, i*) *∈*

*{in, inout}*, the corresponding value *vali* 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 *returnV al ∈ 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 *return*- *ing*, the parameters of kind *inout* and *out* as well as the return value *returnV al*

are updated and get a value */*= ?. [3](#_bookmark7) If an operation is not returning a result, the result type *t* of operation *op* is *OclV oid*. In that case, we set *returnV al* = *⊥* 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.

* + 1. *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 *objId ∈* Σ*CLASS,c* and each of its currently executed operations *op* with identifier *opId*, we define a function *σsentMsg,c*(*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 ac- tually 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 *σsentMsg,c* has to use the general set *ICLASS* of *target object identiﬁers*.

We define the set of messages sent by

Σ*sentMsg*

*def*

=

*σsentMsg,c*

: Σ*CLASS,c*

*× OP∗ × ID →*

*c*

*c∈CLASS*

*P*(*ICLASS ×* (*SIG ∪ OP* ) *× ID*) } *.*

The set *ID* in the value set *P*(*ICLASS ×* (*SIG ∪ OP* ) *× 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 *IDsourceId*, but we here simply reuse set *ID* for the sake of concision.

An element *⟨targetId, msg, callId⟩ ∈ σsentMsg,c*(*objId, op, opId*) denotes that a message with signature *msg* and call identifier *callId* has been sent from the object *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 corre- sponding 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 *σsentMsg,c*(*objId, op, opId*)= ∅. More formally, we have

*σsentMsg,c*(*objId, op, opId*) *'→*

⎧

⎪ *⟨targetId, msg, callId⟩* }*,* if *⟨srcId, srcOp, srcOpId, opId, executing⟩*

⎪⎨

⎪⎪⎩

*∈ σcurrentOp,c*(*objId, op*)

*∨ ⟨srcId, srcOp, srcOpId, opId, returning⟩*

*∈ σcurrentOp,c*(*objId, op*)

∅*,* otherwise*.*

Additionally, we have to store the actual parameter values of each message

*def*

sent. We therefore define Σ*sentMsgParam* =

*c∈CLASS*

*σsentMsgParam,c* :

Σ*CLASS,c × OP∗ × ID × ICLASS ×* (*SIG ∪ OP* ) *× ID*

*c*

*→ I*?(*t*1) *× ... × I*?(*tn*) *× I*?(*t*) } *.*

The number *n* and the types *ti*, 1 *≤ i ≤ n*, are determined by the formal parameters of the corresponding message signature, i.e., either a signal *sig* = (*ω* : *tc × t*1 *× ... × tn*) *∈ SIG∗* or an operation *op* = (*ω* : *tc × t*1 *× ... × tn →*

*c*

*t*) *∈ OP∗*.

*c*

Each function *σsentMsgParam,c* is defined by

*σsentMsgParam,c*(*objId, op, opId, targetId, msg, callId*) *'→*

⎧

⎨ *σsentMsg,c*(*objId, op, opId*)

⎪ *⟨val*1*,..., valn, returnV al⟩,* if *⟨targetId, msg, callId⟩∈*

⎪⎩ ∅*,* otherwise*.*

The values *vali ∈ I*?(*ti*) document the parameter values of the message sent. We set all parameters of kind *out* and the return variable *returnV al* to ? by default, i.e., these parameter values are left unassigned until they are calcu- lated. Basically, they are only relevant for synchronous operation calls, where values */*= ? 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*?(*OclV oid*) 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 Σ*sentMsg* and Σ*sentMsgParam* and sub-functions of *σsentMsg,c* and *σsentMsgParam,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 Σ*sentSig*. Within this set, functions *σsentSig,c* return the history of signals sent. Actual parameter values are kept in set Σ*sentSigParam* with functions *σsentSigParam,c*.

Operations called are kept in set Σ*calledOp*. We make use of functions *σcalledOp,c* that return the history of operations called. Set Σ*calledOpP aram* keeps the actual parameter values of called operations, and functions *σcalledOpP aram,c* are used to access the actual parameter values of operations called.

To further distinguish synchronous and asynchronous operation calls, sets Σ*calledSynchOp* and Σ*calledAsynchOp* are employed. Within each set, we have func- tions *σcalledSynchOp,c* and *σcalledAsynchOp,c* that return the history of called syn- chronous and asynchronous operations for a given operation execution. Actual parameter values are kept in sets Σ*calledSynchOpP aram* and Σ*calledAsynchOpP aram* and are accessed by functions *σcalledSynchOpP aram,c* and *σcalledAsynchOpP aram,c*.

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

# Semantics of OCL Messages

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

*I*(*OclMessage*

*def*

) =

*c∈CLASS,op∈OP∗*

*I*(*OclMessage*(*op*))

*c*

*∪ c∈CLASS,sig∈SIG∗ I*(*OclMessage*(*sig*))*,*

*c*

where the set *I*(*OclMessage*(*op*)) for a given operation *op* = (*ω* : *tc × t*1 *×*

*... × tn → t*) *∈ OP∗* is defined as follows:

*c*

*I*(*OclMessage*(*op*)) = *ID × ICLASS × I*(*t*1) *× ... × I*(*tn*)*.*

Set *ID* refers to the unique call identifiers (*callId*) of sent messages. Set *ICLASS* 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* = (*ω* :

*tc × t*1 *× ... × tn*) *∈ SIG∗* is very similar, i.e.,

*c*

*I*(*OclMessage*(*sig*)) = *ID × ICLASS × I*(*t*1) *× ... × I*(*tn*)*.*

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](#_bookmark9).

* 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 pro- posal, as the definition of OclMessageExpCS says that the number of messages sent to the target object is exactly = 1 (instead of *>* 0) [[6](#_bookmark15), Section 4.3].

**Syntax.** The basic syntactical elements of OCL expressions are defined by a so-called *data signature* Σ*M* = (*TM, ≤,* Ω*M*) [[6](#_bookmark15), Section A.2.8], where

* *TM* is the set of *type expressions T*Expr(*t*) for types *t ∈ TB ∪ TE ∪ TC ∪ TS*

[[6](#_bookmark15), Section A.2.5],

* *≤* is a type hierarchy over *TM* [[6](#_bookmark15), Section A.2.7], and
* Ω*M* is the set of operation signatures, Ω*M* = Ω*TM ∪* Ω*B ∪* Ω*E ∪* Ω*C ∪* Ω*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

*def*

Expr =

*t∈TM*

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 *oclIsN ew*(), operator @*pre*, and a predefined result variable named result [[6](#_bookmark15), 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 Ψ*M* to refer to the set of signals defined in an instantiation of an object model *M*.

1. if (a) *etarget ∈* Post-Expr*t*, and
   1. either (*ω* : *tc × t*1 *× ... × tn → t*) *∈* Ω*M* or (*ω* : *tc × t*1 *× ... × tn*) *∈* Ψ*M*, and
   2. *ei ∈* Post-Expr*ti* for all *i ∈ {*1*,..., n}*, then *etarget*^^*ω*(*e*1*,..., en*) *∈* Post-Expr*t*

as well as *etarget*^*ω*(*e*1*,..., en*) *∈* 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 *τ* = *⟨σ*(*M*)*, β⟩* with a system state *σ*(*M*) and a variable assignment *β* : *V art → I*(*t*). A variable assignment *β* maps variable names to values [[6](#_bookmark15), Section A.3.1.1]. In the following, let *Env* be the set of environments *τ* = *⟨σ*(*M*)*, β⟩*.

While the semantics of an OCL expression *e* is usually defined by a function *I*[[*e*]] : *Env → I*(*t*), we have to consider two environments in the case of oper- ation postconditions, i.e., the environments *τpre* (at the beginning of operation execution) and *τpost* (at time of termination). Thus, the interpretation function for expressions *e* specified in postconditions becomes *I*[[*e*]] : *Env×Env → I*(*t*). We now define the semantics of OCL message operators over environments (*τpre, τpost*) in the context of a given object *objId ∈* Σ*CLASS,c* and an executed operation with signature *op ∈ OP∗* and identifier *opId* (implicitly, we assume

*c*

that the operation execution has just terminated).

First, we define a help set *MSGetarget*ˆˆ*ω*(*e*1*,...,en*) that keeps all relevant messages sent.

*MSGetarget*ˆˆ*ω*(*e*1*,...,en*)

*def*

=

*⟨callId, eV altarget, v*1*,..., vn⟩ | ∃c' ∈ CLASS* :

*eV altarget* = *I*[[*etarget*]](*τpre, τpost*) *∈ ICLASS* (*c'*) *\ {⊥}*

*∧ ∀i ∈ {*1*,..., n}* : *eV ali* = *I*[[*ei*]](*τpre, τpost*) *∈ I*?(*ti*) *\ {⊥}*

*∧ ∀i ∈ {*1*,..., n}* : (*eV ali /*= ? *⇒ eV ali* = *vi*)

*∧* *∃msg* = (*ω* : *tc' × t*1 *× ... × tn → t*) *∈ OP∗*

*c'*

*∨ ∃msg* = (*ω* : *tc' × t*1 *× ... × tn*) *∈ SIGc'* *,* such that

*∗*

*⟨eV altarget, msg, callId⟩∈ σsentMsg,c*(*objId, op, opId*)

*∧ ∃anyV al ∈ I*?(*OclAny*): *⟨v*1*,..., vn, anyV al⟩∈*

*σsentMsgParam,c*(*objId, op, opId, eV altarget, msg, callId*) , *.*

Informally, the elements *⟨callId, eV altarget, v*1*,..., vn⟩* of this set are deter- mined as follows. The target object identifier *eV altarget* is evaluated from *etarget* and must be well-defined in the sense that it is different from *⊥*. Similarly, all evaluations *eV ali* of the parameter expressions *ei* must be well-defined. For consistency reasons, those *eV ali* that evaluate to an actual value (i.e., a value

*/*= ?) must be equal to *vi*.

Furthermore, there must be a message signature *msg*, such that the triple

*⟨eV altarget, msg, callId⟩* represents a message sent from object *objId* within the regarded operation execution. And finally, the values *vi* must be equal to the actual parameter values that are stored (and potentially updated) for the investigated messages sent. Variable *anyV al* 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 *MSGetarget*ˆˆ*ω*(*e*1*,...,en*). For each *i ∈ {*1*,..., m}*, let *xi* = *⟨callIdi, eV altarget, v*1*,i,..., vn,i⟩* be a dis- tinct element of set *MSGetarget*ˆˆ*ω*(*e*1*,...,en*) with *callIdj < callIdj*+1 for all *j ∈ {*1*,...,m −* 1*}*.

Because of the unique call identifiers of messages sent, the latter condition induces an order on the elements *xi ∈ MSGetarget*ˆˆ*ω*(*e*1*,...,en*), such that we can define the corresponding *sequence of messages sent* as follows, using double angle brackets to denote a sequence of elements.

*I*[[*etarget*^^*ω*(*e*1*,..., en*)]](*τpre, τpost*) = *⟨⟨x*1*, x*2*,..., xm⟩⟩*

*def*

If at least one *I*[[*ei*]](*τpre, τpost*), 1 *≤ i ≤ n*, evaluates to *⊥*, the whole expression evaluates to the empty sequence, as we have explicitly required *I*[[*ei*]](*τpre, τpost*) *∈ I*?(*t*) *\{⊥}* in the definition of *MSGetarget*ˆˆ*ω*(*e*1*,...,en*). Alter- natively, as the OCL 2.0 proposal does not consider this issue, we can define a semantics that evaluates to *⊥* 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*[[*etarget*]](*τpre, τpost*) evaluates to an object *eV altarget* that still exists at the time of postcondition evaluation. Instead, we interpret *I*[[*etarget*]](*τpre, τpost*) as an *object identiﬁer ∈ ICLASS*(*c'*) only. To further indi- cate that we are only referring to an object identifier here, we do not underline *eV altarget*.

The meaning of *eV altarget* = *⊥* is now that the object identifier *eV altarget* is not defined w.r.t. the complete execution of the operation under consideration.

In this case, *I*[[*etarget*^^*ω*(*e*1*,..., en*)]](*τpre, τpost*) results in the empty sequence.

* 1. *OCL Message Operations*

The signatures of the four predefined OCL message operations are

*IhasReturned*:*OclMessage→Boolean* : *I*(*OclMessage*) *→ I*(*Boolean*)*, Iresult*:*OclMessage→OclAny* : *I*(*OclMessage*) *→ I*(*OclAny*)*, IisOperationCall*:*OclMessage→Boolean* : *I*(*OclMessage*) *→ I*(*Boolean*)*,* and *IisSignalSent*:*OclMessage→Boolean* : *I*(*OclMessage*) *→ I*(*Boolean*)*.*

As existing OCL syntax does not need to be adjusted for message opera- tions, we here only have to define a semantics for message operations. Gen- erally, the semantics of an operation (*ω* : *tc × t*1 *× ... × tn → t*) *∈ OP∗* is

*c*

recursively defined by

*def*

*I*[[*ω*(*e*1*,..., en*)]](*τpre, τpost*) =

*I*(*ω*)(*τpost*) *I*[[*e*1]](*τpre, τpost*)*,...,I*[[*en*]](*τpre, τpost*) *.*

We define the semantics of OCL message operations over environments (*τpre, τpost*) in the context of a given object *objId ∈* Σ*CLASS,c* and an executed operation with signature *op* = (*ω* : *tc × t*1 *× ... × tn → t*) *∈ OP∗* and identifier *opId* (implicitly, we assume that the operation execution has just terminated).

*c*

* + 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 *σcalledSynchOp,c* to check whether an OCL message *⟨callId, targetId, v*1*,..., vn⟩∈*

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

*I*(*hasReturned*)(*τ* )() *⟨callId, targetId, v ,...,v ⟩* *def*

*post*

⎧

⎪ *true,* if *Emsg ∈ OP* :

1 *n* =

*⟨targetId, msg, callId⟩∈ σcalledSynchOp,c*(*objId, op, opId*)

⎪

⎪⎨

*Λ σcalledSynchOpP aram,c*(*objId, op, opId, targetId, msg, callId*)

= *⟨val*1*,..., valn, returnV al⟩,* such that

*returnV al /*= ?

⎪

⎪ *Λ 6i ∈ {*1*,..., n}* : (*vi /*= ? *⇒ vali* = *vi*)

⎪⎩ *false,* otherwise*.*

Condition *returnV al /*= ? 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 *σcalledSynchOp,c* can also be applied to determine the result of a synchronous message call.

*post*

1

*n*

=

⎧⎪ *returnV al,* if *Emsg ∈ OP* :

*I*[[*result*()]](*τ ,τ*

*pre*

) *⟨callId, targetId, v ,...,v ⟩* *def*

⎪

⎪⎨

⎪

⎪⎪⎩

*⟨targetId, msg, callId⟩∈ σcalledSynchOp,c*(*objId, op, opId*)

*Λ σcalledSynchOpP aram,c*(*objId, op, opId, targetId, msg, callId*)

= *⟨val*1*,..., valn, returnV al⟩,* such that

*returnV al /*= ?

*Λ 6i ∈ {*1*,..., n}* : (*vi /*= ? *⇒ vali* = *vi*)

*⊥,* otherwise*.*

* + 1. *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 *σcalledSynchOp,c* and *σcalledSynchOpP aram,c* are replaced correspondingly, as we now have to consider synchronous and asynchronous operation calls for isOperationCall() and signals sent for op- eration isSignalSent().

Furthermore, condition *returnV al /*= ? is not needed, as we do not inves- tigate whether a message has returned yet. For the sake of brevity, we here

only provide the formal semantics of operation isOperationCall():

*I*[[*isOperationCall*()]](*τ ,τ*

) *⟨callId, targetId, v ,...,v ⟩* *def*

*pre post*

⎧

⎪ *true,* if *Emsg ∈ OP* :

1 *n* =

⎪⎪⎪⎨

⎪

⎪

*⟨targetId, msg, callId⟩∈ σcalledOp,c*(*objId, op, opId*)

*Λ σcalledOpP aram,c*(*objId, op, opId, targetId, msg, callId*)

= *⟨val*1*,..., valn, returnV al⟩,* such that

*6i ∈ {*1*,..., n}* : (*vi /*= ? *⇒ vali* = *vi*)

⎩ *false,* otherwise*.*

# Conclusion and Outlook

Based upon our previous work that already captures Statecharts and state- related operations, we presented further extensions to object models and sys- tem 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 situa- tion 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 asyn- chronous 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 be- longs) might already be destroyed. It is therefore necessary to refer to *object identiﬁers* instead of “real” objects in the semantic definition of OCL mes- sages.

We further had to extend the domain *I*(*t*) of types *t ∈ T* by an unassigned value to allow for symbol ? in message expressions, i.e., *I*?(*t*) = *I*(*t*) *∪ {*?*}*. 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 param- eter values. This is an abstraction from the UML semantics that assumes that a specific reply object is generated and sent [[9](#_bookmark19), 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 oper- ations, 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](#_bookmark22)]. But different publications of recent years indicate that there is a need for *temporal extensions* of OCL, e.g., [[1](#_bookmark11),[2](#_bookmark12),[3](#_bookmark13),[4](#_bookmark14),[11](#_bookmark21)]. 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](#_bookmark16)].

# Acknowledgement

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

# References

1. J. Bradfield, J. Ku¨ster 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.

1. 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.
2. 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.
3. S. Flake and W. Mu¨ller. 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](http://link.springer.de/) under Digital Object Identifier 10.1007/s12270-003- 0026-x.
4. A. Ivner, J. H¨ogstro¨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. 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.

1. 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.

1. OMG, Object Management Group. Unified Modeling Language 1.5 Specification. OMG Document formal/03-03-01, March 2003.
2. M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Universit¨at Bremen, Bremen, Germany, 2001.
3. P. Ziemann and M. Gogolla. An Extension of OCL with Temporal Logic. In J. Ju¨rjens, M. V. Cengarle, E. B. Fernandez, B. Rumpe, and R. Sandner, editors, *Critical Systems Development with UML*, pages 53–62. Technische Universit¨at Mu¨nchen, Institut fu¨r Informatik, 2002.
4. S. Zschaler. Evaluation der Praxistauglichkeit von OCL-Spezifikationen. Master’s thesis, Technical University of Dresden, Faculty of Computer Science, August 2002. (in German).