Towards the Completion of the Formal Semantics of OCL 2.0

Stephan Flake

C-LAB, Paderborn University
Fuerstenallee 11
33102 Paderborn, Germany
Email: `flake@c-lab.de`

Abstract

The Object Constraint Language (OCL) is part of the Unified Modeling Language (UML) to specify restrictions on values of a given UML model. As part of the UML 2.0 standardization process, a proposal for the new version OCL 2.0 has recently been adopted by the Object Management Group. This proposal provides extensive semantic descriptions by both a metamodel-based as well as a formal mathematical approach, but these two semantics are currently neither consistent nor complete. In particular, the formal semantics of the OCL 2.0 proposal currently lacks descriptions of ordered sets, global OCL variable definitions, UML Statechart states, and OCL messages. This article provides corresponding definitions to overcome these deficiencies. We also define a notion of execution traces that capture all system changes of a running system that are necessary to be able to evaluate OCL constraints.

Keywords: Object-Oriented Modeling, Object Constraint Language (OCL), Formal Semantics

1 Introduction

The Object Constraint Language (OCL) has been part of the Unified Modeling Language (UML) since UML version 1.3. OCL is an expression language that enables modelers to formulate constraints in the context of a given UML model. It is used to specify invariants of classes, pre- and postconditions of operations, and conditions of state transitions (Warmer & Kleppe 1999).

A proposal for a new version called *OCL 2.0* has recently been adopted by the Object Management Group (Ivner, Högström, Johnston, Knox & Rivett 2003). Compared to the previous version, some new concepts are introduced in OCL 2.0, i.e.,

- a *metamodel* to better integrate into UML,
- mathematical *tuples* and operations on tuples,
- *ordered sets* in addition to the already existing kinds of collections (sets, sequences, and bags),
- *nesting* of collections (so far, all collections were automatically flattened), and
- access to *messages sent* during operation execution (referred to as *OCL messages*).

The OCL 2.0 proposal provides two semantic descriptions. The first semantics is described by a metamodel-based approach, i.e., the semantics of an OCL expression is given by associating each value defined in the semantic domain with a type defined in the abstract syntax (i.e., the metamodel), and by associating each evaluation with an expression of the abstract syntax. Given a snapshot of a running system, these associations allow to yield a unique value for an OCL expression, which determines the result value of expression evaluation.

Secondly, a *formal* semantics is defined by a set-theoretic mathematical approach called *object model* (Ivner et al. 2003, Appendix A) based on work by Richters (2001). An object model is a tuple

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

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 $as \in ASSOC$ its dedicated classes, the classes' role names, and multiplicities, respectively.

In this article, we call a particular instantiation of an object model a *system*. A system changes over time, i.e., the (number of) objects, their attribute values, and other characteristics change during system execution. The information that is needed to evaluate OCL expressions is stored in *system states* which represent snapshots of the running system. In the OCL 2.0 proposal, a system state $\sigma(\mathcal{M})$ is formally defined as a triple $\sigma(\mathcal{M}) = \langle \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{ASSOC} \rangle$, consisting of a set Σ_{CLASS} of currently existing objects, a set Σ_{ATT} of attribute values of the objects, and a set Σ_{ASSOC} of currently established links that connect the objects.

The formal semantics of OCL expressions is defined over system states. However, this information is not sufficient to evaluate expressions that reason about currently activated Statechart states or messages that have been sent. We already integrated Statecharts to OCL by a formal notion of active state configurations, such that a semantics for state-related operations is already available (Flake & Mueller 2003). This article now further extends that work w.r.t. OCL messages.

The remainder of this article is structured as follows. In Section 2, we explain the concepts of OCL. Section 3 extends the formal definition of object models by additional components that capture Statechart states and OCL messages. In Section 4, we then extend the formal semantics of the OCL 2.0 proposal. Section 5 introduces *traces* for OCL, i.e., sequences of system states that capture all changes of a running system relevant for the evaluation of OCL constraints. Section 6 concludes this article.

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

result() : <<The return type of the invoked operation>>
  -- Returns the result of the invoked operation if the template parameter denotes an operation and the invoked
  -- operation has already returned. Otherwise OclUndefined is returned.

isSignalSent() : Boolean
  -- Returns true if the template parameter represents a signal.

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

Figure 1: Operations on OCL Messages

2 OCL

OCL is a declarative expression-based language, i.e., evaluation of OCL expressions does not have side effects on the corresponding UML model. In the remainder, we will call this UML model the *referred user model*.

Each OCL expression has a type. Besides user-defined model types (e.g., classes or interfaces) and some predefined basic types (e.g., `Integer`, `Real`, or `Boolean`), OCL also has a notion of object collection types (i.e., sets, ordered sets, sequences, and bags). Collection types are homogeneous in the sense that all elements of a collection have a common type. A new feature of the OCL 2.0 proposal is a built-in *tuple* type. Tuples are sequences of a fixed number of elements that can be of different types. Moreover, a standard library is available that provides operations to access and manipulate values and objects.

For example, assume a UML model with classes `Person` and `Company` and an association that connects those classes. We can navigate from class `Person` to class `Company` via that association and make use of the role name `employers` that is attached to the association-end at the `Company` side. The following invariant ensures that each object that is an instance of class `Person` has at least one associated employer:

```
context Person
inv: self.employers->notEmpty()
```

Let us briefly outline how to read this OCL invariant. The class identifier that follows the `context` keyword specifies the class for which the following expression should hold. The keyword `inv` specifies that this is an invariant, i.e., for each object of the context class the following expression must evaluate to true at any time. Note that an invariant may be violated during execution of an operation. In Section 5, we will therefore give a more precise definition of the meaning of *at any time* in this context. The optional keyword `self` refers to the object for which the constraint is evaluated. Attributes, operations, and associations can be accessed by dot notation, e.g., `self.employers` results in a (possibly empty) set of instances of `Company`. The arrow operator indicates that a collection of objects is manipulated by one of the predefined OCL collection operations. For example, operation `notEmpty()` returns true if the accessed set is not empty.

2.1 OCL messages

OCL messages have been newly introduced in the OCL 2.0 proposal to specify constraints over messages sent by objects. It is based on work presented by Kleppe & Warmer (2000, 2002). Basically, an OCL message refers to a signal sent or a (synchronous or asynchronous) operation called. While signals sent are asynchronous and the calling object simply continues its execution, synchronous operation calls make the invoking operation wait for a return value. 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, we refer to the action semantics of UML 1.5 (OMG 2003, Section 2.24).

The concept of OCL messages enables modelers to specify constraints that require that specific signals must have been sent, operations must have been called, or operations must have been completely executed and returned. The corresponding operations are listed in Figure 1. Note that OCL messages can only be accessed in operation postconditions.

Syntax. A parameterized type `OclMessage(T)` is now part of the 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.

OCL messages are obtained by the *message operator* `^^` that is attached to a destination object. For example, the OCL expression `destObj^^setValue(17)` results in the sequence of messages `setValue(17)` that have been sent to the object determined by `destObj` 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 `destObj^^setValue(17)` is `Sequence(OclMessage(setValue(i:Integer)))`.

One can make use of so-called *unspecified values* to indicate that an actual parameter is not restricted to a specific value. Unspecified values are denoted by question marks, e.g., we may specify `destObj^^setValue(?:Integer)`. Specification of parameter types is optional, but 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 `destObj^setValue(17)` results in true if a message `setValue(17)` has been sent to `destObj` during execution of the operation under investigation. Note that the hasSent operator is a shortcut and can be derived from the message operator `^^`. For example, the expression `destObj^setValue(17)` can be replaced by `destObj^^setValue(17)->notEmpty()`.

Semantics. The semantics of OCL messages is currently only defined in the metamodel-based semantics (Ivner et al. 2003, 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.

As a formal semantics of OCL messages has not yet been defined, 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.

Example. We here present a slightly modified version of a postcondition found in the OCL 2.0 proposal (Ivner et al. 2003, Section 2.7.2):

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

The expression `company^^getMoney(amount)` returns the sequence of messages named `getMoney` with actual parameter value `amount`. We require that these messages must all have been returned at the time of termination of operation execution of `giveSalary()` and that exactly one of these messages returned with the result value `true`.

3 Extended object models

We define an extension of object models called *extended object models*, in which the following concepts are newly introduced (cf. Section 3.1):

- operation parameter kinds *in*, *inout*, and *out*,
- a flag that indicates whether an operation is a query operation without side-effects or not,
- signal receptions for classes with corresponding well-formedness rules,
- Statecharts and their association with classes,
- a formal definition of state configurations, and
- an 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 (cf. Section 3.2):

- for each object, the input queue of received signals and operation calls that are waiting to be dispatched,
- the state configurations of all currently existing active objects,
- the currently executed operations, and
- for each currently executed operation, the messages sent so far.

These system state components allow to give a semantics to message-related operators and operations that could so far not formally be defined (cf. Section 4). Moreover, these components are used to define a high-level dynamic semantics for OCL by means of *execution traces* (cf. Section 5).

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

$$T_S = \{OclVoid, OclState, OclAny\}.$$

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., $I(OclVoid) = \{\perp\}$ and

$$I(OclAny) = \bigcup_{t \in T_B \cup T_E \cup T_C \cup \{OclVoid, OclState\}} I(t).$$

Furthermore, let $c \in CLASS$ be a class and $t_c \in T_C$ be the type of class c.[1] 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* (OMG 2003, Section 3.26.6)).

We define *Extended Object Models* $\mathcal{M}$ by the tuple

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

with

- a set $CLASS = ACTIVE \cup PASSIVE$ of active and passive classes,
- a set 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,
- a function $isQuery : CLASS \times OP \to Boolean$ that determines whether an operation is a query operation or not,
- 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.

[1] 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 $c \in CLASS$.

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 (Ivner et al. 2003, Richters 2001). We also omit the formal syntax definitions for signals and Statecharts and refer to (Flake & Mueller 2003) for further details. Concerning Statecharts and their inheritance among classes, we assume that there is exactly one Statechart for each active class that complies to some *inheritance policy*. Though the UML 1.5 standard suggests some informal policies (subtyping, strict inheritance, and general refinement) (OMG 2003, Section 2.12.5), different other formal notions for behavioral consistency have been identified in the literature, e.g., by Stumptner & Schrefl (2000).

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

$$FD_c = \langle \; ATT^*_c, OP^*_c, paramKind^*_c, isQuery^*_c, SIG^*_c, SC_c, navEnds^*(c) \; \rangle$$

containing the complete sets of attributes, operations (with corresponding functions that determine parameter kinds and query 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

$$ATT^*_c = ATT_c \;\cup \bigcup_{c' \in parents(c)} ATT_{c'},$$

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. Functions $isQuery^*_c : OP^*_c \rightarrow Boolean$ and $paramKind^*_c : OP^*_c \times \mathbb{N} \rightarrow \{in, inout, out\}$ are derived from functions $isQuery$ and $paramKind$, respectively.

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 $oid(c) = \{oid_1, oid_2, \dots\}$. The domain of a class $c \in CLASS$ is defined as

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

For technical purposes, we also define $I_{CLASS} = \bigcup_{c' \in CLASS} I_{CLASS}(c')$. Note that – in contrast to the current OCL semantics – we distinguish between "real" objects $\underline{oid}$ and their identifiers oid in the remainder of this article, simply by using underlines.

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 called, signals sent, currently executed operations, etc. In this context, we adopt ideas of Ziemann & Gogolla (2002) to formalize currently executed operations and define further functions to capture the required additional information. Formally, a *system state* for an extended object model $\mathcal{M}$ is a tuple

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

We explain the components of system states in more detail, but note that Σ_{CLASS}, Σ_{ATT}, and Σ_{ASSOC} are already defined by Richters (2001) and Ivner *et al.* (2003).

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

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

For further application, we define $\Sigma_{ACTIVE,c}$ for active and $\Sigma_{PASSIVE,c}$ for passive classes correspondingly.

(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) $\Sigma_{ASSOC} = \bigcup_{as \in ASSOC} \Sigma_{ASSOC,as}$ comprises the finite sets $\Sigma_{ASSOC,as}$ that contain links that connect objects. We refer to the sources mentioned above for detailed information about links, i.e., elements of $I_{ASSOC}(as)$, and the formalization of multiplicity specifications.

(4) The current Statechart configurations are kept by $\sigma_{CONF} =$

$$\bigcup_{c \in ACTIVE} \{ \; \sigma_{CONF,c} : \Sigma_{ACTIVE,c} \rightarrow I_{SC}(c) \; \} \; .$$

Each function $\sigma_{CONF,c}$ assigns a complete active state configuration 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. Note that the UML 1.5 only informally specifies state configurations. In particular, we identified deficiencies concerning final states and provided a concise definition of state configurations (Flake & Mueller 2003).

While the definition of a system state so far is sufficient to reason about currently activated states, additional runtime information must be taken into account to be able to evaluate expressions that access OCL messages. This mainly concerns the histories of sending, receiving, and consuming signals and sending, receiving, dispatching, execution, and return of operation calls. Both, the source and the destination object of an OCL message must keep corresponding information. The following subsections describe corresponding system state components that relate to the *local snapshots* defined in the metamodel-based semantics of OCL 2.0.

3.2.1 Currently executed operations

Let $\mathcal{ID}$ be an infinite enumerable set, e.g., $\mathcal{ID} = \mathbb{N}$, and let $Status = \{executing, returning\}$. 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 identifier, an operation signature $op \in OP$, and an operation identifier $opId \in \mathcal{ID}$. The set of currently executed operations is defined by $\Sigma_{currentOp} =$

$$\bigcup_{c \in CLASS} \left\{ \begin{array}{l} \sigma_{currentOp,c} : \Sigma_{CLASS,c} \times OP_c^* \rightarrow \\ \mathcal{P}(I_{CLASS} \times OP \times \mathcal{ID} \times \mathcal{ID} \times Status) \end{array} \right\}$$

Each function $\sigma_{currentOp,c}$ gives a set of tuples of the form $\langle srcId, srcOp, srcOpId, opId, status\rangle$ that uniquely identify all currently executed operations for a given object and operation name. Elements *srcId*, *srcOp*, and *srcOpId* 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 *op*. We require that associated operation identifiers must not change until the execution of that operation terminates.

A flag $status \in Status$ indicates the current status of operation execution. Compared to the messaging actions specified in UML 1.5, we here omit statuses *ready* and *complete* (OMG 2003, Section 2.19.2.3), as they are currently not necessary in the context of the OCL semantics.

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

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

Each function $\sigma_{currentOpParam,c}$ gives the actual parameter values of the currently executed operations for a given object, operation signature, and operation execution identifier. In the definition above, we applied sets $I^?(type) = I(type) \cup \{?\}$, where $type \in T$. Symbol ? denotes the *unspecified status* of a value. This symbol must not be mixed up with the *undefined value* denoted by `OclUndefined` (or $\perp$ in this article) and is also different from the String literal `'?'`. Only operation parameters i with $paramKind(c, op, i) = out$ and the return value carry the unspecified value during operation execution.

We require that all parameter values do not change until operation termination. But when the status of operation execution changes from *executing* to *returning*, the parameters of kind *inout*, *out*, and the return value $returnVal$ are updated. However, this is performed by the actual system and is not in the scope of the OCL semantics. If an operation is not returning a result, the result type t of operation *op* is $OclVoid$. In that case, $returnVal$ is set to $\perp$ 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 make use of the message operator `^^`, we have to store the *history of messages sent* for each executed operation. For each object $\underline{oid} \in \Sigma_{CLASS,c}$ and each of its currently executed operations *op* with identifier *opId*, we define a function $\sigma_{sentMsg,c}(\underline{oid}, op, opId)$ that gives the set of messages sent with their corresponding destination objects.

When a message is sent from an execution of operation *op* with identifier *opId* to a destination object $\underline{destId}$, that destination 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 destination 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} to refer to destination object identifiers. Recall that we distinguish between "real" objects $\underline{destId}$ and their identifiers $destId$.

We define $\Sigma_{sentMsg} =$

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

Set $\mathcal{ID}$ in $\mathcal{P}(I_{CLASS} \times (SIG \cup OP) \times \mathcal{ID})$ 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 $\mathcal{ID}_{\underline{oid}}$, but we here simply reuse set ID for the sake of concision. However, note that we here additionally have to require a total order for $\mathcal{ID}$, such that we can uniquely build sequences of messages sent (cf. Section 4.1).

An element

$$\langle destId, msg, callId\rangle \in \sigma_{sentMsg,c}(\underline{oid}, op, opId)$$

denotes that a message with signature msg and call identifier $callId$ has been sent from object $\underline{oid}$ to the (not necessarily still existing) object with identifier $destId$ as part of operation execution *op* with identifier *opId*.

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

$$\bigcup_{c \in CLASS} \left\{ \begin{array}{l} \sigma_{sentMsgParam,c} : \\ \quad \Sigma_{CLASS,c} \times OP_c^* \times \mathcal{ID} \times I_{CLASS} \\ \qquad \times (SIG \cup OP) \times \mathcal{ID} \\ \qquad \rightarrow I^?(t_1) \times \ldots \times I^?(t_n) \times I^?(t) \end{array} \right\},$$

where the number n and the types t_i of message attributes $val_i \in I(t_i)$, $1 \leq i \leq n$, 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 $op = (\omega : t_c \times t_1 \times \ldots \times t_n \rightarrow t) \in OP_c^*$. The values $val_1, \ldots, val_n$ correspond to the actual parameter values of the message and are thus fixed, but note that parameters of kind *out* may carry the unspecified value. We set $returnVal = ?$ by default, i.e., the potential return value is left unspecified until it is calculated. Basically, $returnVal \in I^?(t)$ is only relevant for synchronous operation calls, where it gets a value $\in I(t)$ 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 corresponding return value simply remains unspecified.

Help Sets and Functions. In the remainder of this article, we need some helper 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 straightforward, 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 $\sigma_{calledAsynchOpParam,c}$.

3.2.3 Input queues

Set $\Sigma_{inputQueue}$ is used to store events, i.e., in our terms operation calls and signals that are sent to objects and still waiting to be dispatched. While other events like *change events*, *time events*, and implicit *completion events* invoked by Statecharts have to be considered in a more general notion of an input queue, we here restrict to those events that are relevant for evaluating OCL expressions. We later refer to input queues to update the system state when a signal or operation is dispatched. This enables us to change the set of currently executed operations accordingly, which is essential for a well-defined semantics of OCL message operations.

We define $\Sigma_{inputQueue} =$

$$\bigcup_{c\in CLASS} \{ \ \sigma_{inputQueue,c} : \Sigma_{CLASS,c} \to \mathcal{P}(I_{CLASS} \times OP \times \mathcal{ID} \times (SIG_c^* \cup OP_c^*) \times \mathcal{ID}) \ \},$$

where each function $\sigma_{inputQueue,c}$ maps to a set of sent signals and operations. Again, we make use of the general set I_{CLASS} instead of Σ_{CLASS}, as we cannot assume that the source object from which an asynchronous message was sent is still existing in the current system state. However, for a message that represents a synchronous operation call, we assume that the corresponding source object exists at least until the invoked operation terminates and returns. In the latter case, we can therefore refer to objects of set $\Sigma_{CLASS} \subset I_{CLASS}$.

Finally, we keep the actual parameter values of waiting messages in set $\Sigma_{inputQueueParam} =$

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

A function $\sigma_{inputQueueParam,c}$ gives the actual parameter values of the waiting messages w.r.t. a given object $\underline{oid}$.

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

4 Formal semantics of OCL

The formal semantics of OCL currently lacks a formalization of

- operations on predefined collection type `OrderedSet`,
- global variable definitions (called *def-clauses* in OCL),
- operations on Statechart states, and
- operators and operations to access and reason about OCL messages.

Note that operations defined for ordered sets are basically the same as for sequences and that 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 have to be regarded here. As we already integrated Statecharts to OCL in a previous article (Flake & Mueller 2003), we now formally define operators and operations to access and reason about OCL messages.

First, we formally define the semantic domain of OCL messages by $I(OclMessage) =$

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

where the set $I(OclMessage(op))$ for a given operation $op = (\omega : t_c \times t_1 \times \ldots \times t_n \to t) \in OP_c^*$ 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 destination object to which the message is sent. Set $I(OclMessage(sig))$ for a signal $sig = (\omega : t_c \times t_1 \times \ldots \times t_n) \in SIG_c^*$ is defined in the same way.

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 shown in Section 2.1.

Syntax. The basic syntactical elements of OCL expressions are defined by a so-called *data signature* $\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$ (Ivner et al. 2003, Appendix A.2.8), where

- $T_{\mathcal{M}}$ is the set of *type expressions* $T_{\text{Expr}}(t)$ for types $t \in T_B \cup T_E \cup T_C \cup T_S$,
- $\leq$ is a type hierarchy over $T_{\mathcal{M}}$, 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 $\text{Expr} = \bigcup_{t\in T_{\mathcal{M}}} \text{Expr}_t$ 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* (Ivner et al. 2003, Appendix 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

$$\begin{array}{l}
MSG_{e_{dest}\hat{}\hat{}\omega(e_1,...,e_n)} = \\
\quad \{ \ \langle callId, eVal_{dest}, v_1, ..., v_n \rangle \ \mid \ \exists c' \in CLASS : \\
\qquad eVal_{dest} = I[[e_{dest}]](\tau_{pre}, \tau_{post}) \in I_{CLASS}(c') \setminus \{\bot\} \\
\qquad \wedge \ (\ \exists msg = (\omega : t_{c'} \times t_1 \times ... \times t_n \to t) \in OP^*_{c'} \\
\qquad\qquad \vee \ \exists msg = (\omega : t_{c'} \times t_1 \times ... \times t_n) \in SIG^*_{c'} \ ) \\
\qquad \wedge \ \forall i \in \{1, \ldots, n\} : eVal_i = I[[e_i]](\tau_{pre}, \tau_{post}) \in I^?(t_i) \setminus \{\bot\} \\
\qquad \wedge \ \forall i \in \{1, \ldots, n\} : (eVal_i \neq \ ? \ \Rightarrow \ eVal_i = v_i) \\
\qquad \wedge \ \langle eVal_{dest}, msg, callId \rangle \in \sigma_{sentMsg,c}(\underline{oid}, op, opId) \\
\qquad \wedge \ \exists anyVal \in I^?(OclAny) : \langle v_1, \ldots, v_n, anyVal \rangle \in \\
\qquad\qquad \sigma_{sentMsgParam,c}(\underline{oid}, op, opId, eVal_{dest}, msg, callId) \ \}
\end{array}$$

Figure 2: Semantics of OCL Expressions with Message Operator

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) $e_{dest} \in$ 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) $e_i \in$ Post-Expr$_{t_i}$ for all $i = 1, \ldots, n$,

then $e_{dest}\hat{}\hat{}\omega(e_1, \ldots, e_n) \in$ Post-Expr$_t$ and $e_{dest}\hat{}\omega(e_1, \ldots, e_n) \in$ 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. In the following, let Env be the set of all 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} (before operation execution) and τ_{post} (after operation execution). Thus, the interpretation function for expressions e specified in postconditions becomes $I[[e]] : Env \times Env \to I(t)$.

The semantics of OCL message operator ^^ is defined over environments $(\tau_{pre}, \tau_{post})$ in the context of a given object $\underline{oid} \in \Sigma_{CLASS,c}$ 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 $MSG_{e_{dest}\hat{}\hat{}\omega(e_1,...,e_n)}$ that keeps all relevant messages sent (cf. Figure 2).

In the following, let m be the number of elements in $MSG_{e_{dest}\hat{}\hat{}\omega(e_1,...,e_n)}$. For each $i \in \{1, \ldots, m\}$, let $x_i = \langle callId_i, eVal_{dest}, v_{1,i}, \ldots, v_{n,i} \rangle$ be a distinct element of set $MSG_{e_{dest}\hat{}\hat{}\omega(e_1,...,e_n)}$ 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 $x_i \in MSG_{e_{dest}\hat{}\hat{}\omega(e_1,...,e_n)}$, such that we can define the corresponding *sequence of messages sent* as follows, using double angle brackets to denote a sequence of elements:

$$I[[e_{dest}\hat{}\hat{}\omega(e_1, ..., e_n)]](\tau_{pre}, \tau_{post}) = \langle\langle x_1, ..., x_m \rangle\rangle \ .$$

If at least one $I[[e_i]](\tau_{pre}, \tau_{post})$, $1 \leq i \leq n$, evaluates to $\bot$, the 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 $MSG_{e_{dest}\hat{}\hat{}\omega(e_1,...,e_n)}$. Note that we could have also defined a semantics that evaluates to $\bot$ in this case.

Furthermore, it is not clearly defined in the OCL 2.0 proposal whether the destination 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, this means that we have to distinguish between objects and their identifiers, in contrast to the current formal OCL semantics. We cannot assume that $I[[e_{dest}]](\tau_{pre}, \tau_{post})$ evaluates to an object $\underline{eVal}_{dest}$ that still exists at the time of postcondition evaluation. Instead, we interpret $I[[e_{dest}]](\tau_{pre}, \tau_{post})$ as an object identifier $\in I_{CLASS}(c')$ only. To further indicate that we are only interested in an object identifier here, we do not underline $eVal_{dest}$ in Figure 2.

The meaning of $eVal_{dest} = \bot$ is now that the object identifier $eVal_{dest}$ is not defined w.r.t. the complete execution of the operation under consideration. In this case, $I[[e_{dest}\hat{}\hat{}\omega(e_1, ..., e_n)]](\tau_{pre}, \tau_{post})$ results in the empty sequence.

4.2 OCL message operations

The operation signature of OCL message operation `hasReturned()` is

$$\begin{array}{l} I_{hasReturned:OclMessage \to Boolean} : \\ \quad I(OclMessage) \to I(Boolean) \ , \end{array}$$

and the signatures of the other operations `result()`, `isOperationCall()`, and `isSignalSent()` are very similar.

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

$$\begin{array}{l} I[[\omega(e_1, \ldots, e_n)]](\tau_{pre}, \tau_{post}) = \\ \quad I(\omega)(\tau_{post}) \ (\ I[[e_1]](\tau_{pre}, \tau_{post}), \\ \qquad\qquad \ldots, I[[e_n]](\tau_{pre}, \tau_{post}) \ ) \ . \end{array}$$

We define the semantics of OCL message operations over environments $(\tau_{pre}, \tau_{post})$ in the context of a given object $\underline{oid} \in \Sigma_{CLASS,c}$ 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 $opId$ (again, we implicitly assume that the operation execution has just terminated).

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 a given OCL message $\langle callId, destId, v_1, \dots, v_n \rangle \in I(OclMessage)$ has returned and to determine its result, i.e.,

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

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()` can easily be derived from the previous definition and is omitted here for the sake of brevity.

The semantics of operations `isSignalSent()` and `isOperationCall()` is also easily obtained from the formal definition of operation `hasReturned()`. The main difference is that the regarded functions $\sigma_{calledSynchOp,c}$ and $\sigma_{calledSynchOpParam,c}$ are replaced correspondingly, as we now have to consider both synchronous and asynchronous operation calls for `isOperationCall()` and signals sent for operation `isSignalSent()`. Furthermore, condition $returnVal \neq$? does not need to hold for these two operations and can simply be omitted.

5 Dynamic semantics

So far, it is not defined how a system state is actually built. The OCL 2.0 semantics simply assumes that a system state $\langle \Sigma_{CLASS}, \Sigma_{ATT}, \Sigma_{ASSOC} \rangle$ is given with 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. While this structure is easy to obtain from a concrete (implementation of a) running system, the situation becomes more complicated when OCL messages are considered, as corresponding information about specific parts of the *execution history* has to be available in subsequent system states for the evaluation of postconditions.

5.1 Traces

Due to the information to keep for subsequent system states, we have to define *traces*, i.e., sequences of system states that keep track of all *noteworthy changes* within a running system. In the context of checking OCL constraints, we are, for instance, not interested in every single attribute value change that occurs during execution of an operation. Instead, we are interested in those system states in which something happens that is of relevance for evaluation of OCL expressions, e.g., when an operation terminates and a corresponding postcondition has to be checked.

In the simplest case, e.g., when (an implementation of) the system is executed on a single CPU, there is a clear temporal order on the system execution. But when (the implementation of) the system is distributed, we have a partial order among the system executions. This problem can be treated in an ideal case by introducing a *global clock* that allows for a global view on the system.

We here follow the idea of a global view on the system. A well-defined system state sequence called *trace* for an instantiation of an extended object model $\mathcal{M}$ is an (infinite) sequence of system states, i.e.,

$$trace(\mathcal{M}) = \langle\langle\ \sigma(\mathcal{M})_{[0]}, \sigma(\mathcal{M})_{[1]}, \dots, \sigma(\mathcal{M})_{[i]}, \dots \rangle\rangle.$$

The first trace element $\sigma(\mathcal{M})_{[0]}$ denotes the initial system state. Given a system state $\sigma(\mathcal{M})_{[i]}$, $i \in \mathbb{N}_0$, the next system state $\sigma(\mathcal{M})_{[i+1]}$ is added to the trace when for at least one *noteworthy change* occurs. The set of these changes is identified in the next subsection.

5.2 When to check constraints

It is quite obvious when to check pre- and postconditions, i.e., just before and just after execution of the according operation. Invariants of an object, in contrast, have to be checked when certain parts of the *object status* change. In this context, we use the term *object status* to refer to the whole information stored for an object in a system state, in contrast to the *object states* which refer to the currently activated Statechart states of the object.

It is often assumed that the status of an object only changes through operation calls. While this might be suitable in some application domains, the situation becomes different when objects are modeled in combination with Statecharts. In UML Statecharts, *elapsed time events* and *change events* can be specified to trigger transitions, e.g., `after(1 sec)` or `when(x > 100)`. These are basically monitors that permanently check for a condition to become true and then raise an internal event to trigger the according transition in the next run-to-completion-step (RTC-step). Thus, a new Statechart configuration is entered without any operation call. Similarly, signals consumed by an RTC-step can also cause a new Statechart configuration to be entered. Invariants must therefore be checked in such cases as well.

However, invariants do not have to be checked in all cases of status changes. For example, assume that a message is sent by an object $\underline{oid}$, such that the message history $\sigma_{sentMsg,c}(\underline{oid}, op, opId)$ changes and therefore the object status changes as well. Certainly, it is not necessary to check the invariants of the object yet.

Noteworthy Changes. Let $inv(c)$ denote the set of invariants of a class $c \in CLASS$. Let $inv^*(c)$ denote the full set of invariants for a class c, i.e.,

$$inv^*(c) = inv(c) \ \cup \bigcup_{c' \in parents(c)} inv(c').$$

Similarly, let $pre(op, c)$ and $post(op, c)$ denote the pre- and postconditions of an operation $op \in OP^*_c$. Recall that we assume that there is an *inheritance policy* for Statecharts that guarantees that state-related OCL expressions are well-defined over inheritance relationships among active classes with associated Statecharts.

We identified the following kinds of *noteworthy changes* relevant for evaluation of OCL expressions. In each case, we give a corresponding rule for updating the current system state $\sigma(\mathcal{M})_{[i]}$ and indicate which OCL constraints have to be checked. Note that different kinds of noteworthy changes might occur in parallel at the same instant of time, such that more than one of the rules might have to be applied on a given system state $\sigma(\mathcal{M})_{[i]}$. For example, a number of objects can be created at the same time on different nodes in a distributed system, and in addition one or more new links can be established.

Although we abstract from an explicit notion of time here, we have to assume a global view on the system to determine the set of noteworthy changes on the whole (even distributed) system at each instant of time.

In the following rules, we are using the $[i]$-annotation also for the components and functions defined in $\sigma(\mathcal{M})_{[i]}$, $i \in \mathbb{N}_0$.

(1) Let $\underline{oid}_1, \ldots, \underline{oid}_n$ be the **objects** of classes $c_j \in CLASS$, $1 \le j \le n$, that are newly **created**.

$$\forall j \in \{1, \ldots, n\} : \\ \Sigma_{CLASS, c_j[i+1]} = \Sigma_{CLASS, c_j[i]} \cup \{\underline{oid}_j\}$$

Task: Check invariants $inv^*(c_j)$ for all objects $\underline{oid}_j$ on system state $\sigma(\mathcal{M})_{[i+1]}$.

(2) Let $\underline{oid}_1, \ldots, \underline{oid}_m$ be the **objects** of classes $c_j \in CLASS$, $1 \le j \le m$, that are **destroyed**.

$$\forall j \in \{1, \ldots, m\} : \\ \Sigma_{CLASS, c_j[i+1]} = \Sigma_{CLASS, c_j[i]} \setminus \{\underline{oid}_j\}$$

(3) Let l_{as_j}, $1 \le j \le k$, be the **links** of associations $as_j \in ASSOC$ that are newly **established**.

$$\forall j \in \{1, \ldots, k\} : \\ \Sigma_{ASSOC, as_j[i+1]} = \Sigma_{ASSOC, as_j[i]} \cup \{l_{as_j}\}$$

(4) Let l_{as_j}, $1 \le j \le p$, be the **links** for associations $as_j \in ASSOC$ that are **removed**.

$$\forall j \in \{1, \ldots, p\} : \\ \Sigma_{ASSOC, as_j[i+1]} = \Sigma_{ASSOC, as_j[i]} \setminus \{l_{as_j}\}$$

(5) Let cfg_j, $1 \le j \le q$, be the **new state configurations** that are **reached** for objects $\underline{oid}_j$ of active classes c_j.

$$\forall j \in \{1, \ldots, q\} : \\ \sigma_{CONF, c_j}(\underline{oid}_j)_{[i+1]} = cfg_j$$

Task: Check $inv^*(c_j)$ for object $\underline{oid}_j$ on system state $\sigma(\mathcal{M})_{[i+1]}$.

(6) Let op_j with $opId_j$, $1 \le j \le r$, be the operation executions of objects $\underline{oid}_j$ from classes $c_j \in CLASS$ that **sent a message** named msg_j with identifier $msgId_j$ and actual parameter values $v_{j,1}, ..., v_{j,n_j}$ to object $destId_j$ with call identifier $callId_j$.

$$\forall j \in \{1, \ldots, r\} : \\ \sigma_{sentMsg, c_j}(\underline{oid}_j, msg_j, msgId_j)_{[i+1]} = \\ \sigma_{sentMsg, c_j}(\underline{oid}_j, msg_j, msgId_j)_{[i]} \\ \cup \{\langle destId_j, op_j, callId_j \rangle\} \\ \text{and} \\ \sigma_{sentMsgParam, c_j}(\underline{oid}_j, op_j, opId_j, \\ destId_j, msg_j, callId_j)_{[i+1]} = \\ \langle v_{j,1}, ..., v_{j,n_j}, ? \rangle$$

(7) Let $\underline{oid}_j$, $1 \le j \le v$, be the objects of classes $c_j \in CLASS$ that **receive a message** named msg_j with call identifier $callId_j$ invoked by an operation execution of object $srcId_j$ of a class c'_j (identified by $srcOp_j$ and $srcOpId_j$).

$$\forall j \in \{1, \ldots, v\} : \\ \sigma_{inputQueue, c_j}(\underline{oid}_j)_{[i+1]} = \\ \sigma_{inputQueue, c_j}(\underline{oid}_j)_{[i]} \\ \cup \{\langle srcId_j, srcOp_j, srcOpId_j, msg_j, callId_j \rangle\} \\ \text{and} \\ \sigma_{inputQueueParam, c_j}(\underline{oid}_j, srcId_j, srcOp_j, \\ srcOpId_j, msg_j, callId_j)_{[i+1]} = \\ \sigma_{sentMsgParam, c'_j}(srcId_j, srcOp_j, srcOpId_j, \\ \underline{oid}_j, msg_j, callId_j)_{[i]}$$

(8) Let $sigSent_j = \langle srcId_j, srcOp_j, srcOpId_j, sig_j, callId_j \rangle$, $1 \le j \le w$, be the **signals** that are **consumed** by objects $\underline{oid}_j$ of classes $c_j \in CLASS$.

$$\forall j \in \{1, \ldots, w\} : \\ \sigma_{inputQueue, c_j}(\underline{oid}_j)_{[i+1]} = \\ \sigma_{inputQueue, c_j}(\underline{oid}_j)_{[i]} \setminus \{sigSent_j\}$$

(9) Let $opCalled_j = \langle srcId_j, srcOp_j, srcOpId_j, op_j, callId_j \rangle$, $1 \le j \le x$, be the **waiting operation calls** that are **dispatched** by objects $\underline{oid}_j$ of classes $c_j \in CLASS$.

$$\forall j \in \{1, \ldots, x\} : \\ \sigma_{inputQueue, c_j}(\underline{oid}_j)_{[i+1]} = \\ \sigma_{inputQueue, c_j}(\underline{oid}_j)_{[i]} \setminus \{opCalled_j\} \\ \text{and} \\ \sigma_{currentOp, c_j}(\underline{oid}_j, op_j)_{[i+1]} = \\ \sigma_{currentOp, c_j}(\underline{oid}_j, op_j)_{[i]} \\ \cup \{\langle srcId_j, srcOpId_j, \\ callId_j, newId_j, executing \rangle\}, \\ \text{where } newId_j \in \mathcal{ID} \text{ is a unique id for } op_j \\ \text{and} \\ \sigma_{currentOpParam, c_j}(\underline{oid}_j, op_j, newId_j)_{[i+1]} = \\ \sigma_{inputQueueParam, c_j}(\underline{oid}_j, srcId_j, srcOp_j, \\ srcOpId_j, op_j, callId_j)_{[i]}$$

Task: For all $j \in \{1, ..., x\}$, check $pre(op_j, c_j)$ of operation op_j with identifier $newId_j$ on system state $\sigma(\mathcal{M})_{[i+1]}$.

(10) Let op_j with $opId_j$, $1 \le j \le y$, be the **executing operations** of objects $\underline{oid}_j$ that **terminate**.

Let $srcId_j$, $srcOp_j$, and $srcOpId_j$ be the object identifier, the operation name, and the operation identifier of the operation execution that initiated the execution of op_j with $opId_j$.

$$\forall j \in \{1, \ldots, y\} : \\ \sigma_{currentOp, c}(\underline{oid}_j, op_j)_{[i+1]} = \\ \big(\sigma_{currentOp, c_j}(\underline{oid}_j, op_j)_{[i]} \\ \setminus \{\langle srcId_j, srcOp_j, \\ srcOpId_j, opId_j, executing \rangle\} \big) \\ \cup \{\langle srcId_j, srcOp_j, \\ srcOpId_j, opId_j, returning \rangle\} \\ \text{and} \\ \sigma_{currentOpParam, c_j}(\underline{oid}_j, op_j, opId_j)_{[i+1]} = \\ \langle v_{j,1}, ..., v_{j,n_j}, returnVal_j \rangle$$

Note that it is not the scope of OCL to perform updates on the parameter values when an operation terminates, just as it is not the task of OCL to update attribute values. We therefore assume that the system performs the necessary updates on the actual parameter values $v_{j,1}, ..., v_{j,n_j}$ and the return value $returnVal_j$ of the terminating operation identified by $opId_j$. Thus, all parameters of kind *in* are still unchanged and the parameters of kind *inout* and *out* are already updated in system state $\sigma(\mathcal{M})_{[i]}$.

Task: For all $j \in \{1, ..., y\}$, check $post(op_j, c_j)$ of operation op_j with identifier $opId_j$ on system state $\sigma(\mathcal{M})_{[i+1]}$. For passive objects $\underline{oid}_j$, we here also check the invariants $inv^*(c)$ on system state $\sigma(\mathcal{M})_{[i+1]}$. In contrast, invariants of active objects are only checked after completion of RTC-steps, which is covered by noteworthy change (5).

(11) Let op_j with $opId_j$, $1 \leq j \leq z$, be the **terminated operations** of objects $\underline{oid}_j$ that **return**.

For each $j \in \{1, ..., z\}$, let $srcId_j \in I(t_{c'_j})$ be the corresponding source object identifier, $srcOp_j$ the invoking operation with identifier $srcOpId_j$, and $callId_j$ the message call identifier. Parameter values are returned for *synchronous* operation calls only, i.e.,

$$\begin{aligned}
&\forall j \in \{1, \ldots, z\}: \\
&\quad \sigma_{currentOp,c_j}(\underline{oid}_j, op_j)_{[i+1]} = \\
&\quad\quad \sigma_{currentOp,c_j}(\underline{oid}_j, op_j)_{[i]} \\
&\quad\quad\quad \setminus \{\langle\, srcId_j, srcOp_j, \\
&\quad\quad\quad\quad srcOpId_j, opId_j, returning\rangle\}
\end{aligned}$$

and

$$\begin{aligned}
&\text{if } \langle \underline{oid}_j, op_j, callId_j\rangle \in \\
&\quad \sigma_{calledSynchOp,c'_j}(srcId_j, srcOp_j, srcOpId_j) \\
&\text{then } \sigma_{calledSynchOpParam,c'_j}(srcId_j, srcOp_j, \\
&\quad\quad srcOpId_j, \underline{oid}_j, op_j, callId_j)_{[i+1]} = \\
&\quad \sigma_{currentOpParam,c_j}(\underline{oid}_j, op_j, opId_j)_{[i]}
\end{aligned}$$

For the sake of brevity, we have not explicitly defined all updates on the parameter functions $\sigma_{currentOpParam,c}$, $\sigma_{sentMsgParam,c}$, and $\sigma_{inputQueueParam,c}$. In particular, when an element from one of the sets $\sigma_{currentOp,c}$, $\sigma_{sentMsg,c}$, and $\sigma_{inputQueue,c}$ is removed, we have to remove the corresponding parameters, too. Moreover, note that updates on the attributes in set *ATT* and the actual parameters of operations are not in the scope of this OCL semantic definition, such that we implicitly assume the correct attribute and parameter values in each system state $\sigma(\mathcal{M})_{[i]}$. Nevertheless, all other components of system state $\sigma(\mathcal{M})_{[i]}$ that are not explicitly considered in the update sections remain unchanged for the subsequent system state $\sigma(\mathcal{M})_{[i+1]}$.

6 Conclusion

Based upon our previous formalization that already captures Statecharts and state-related operations, this article presents further extensions to object models and system states, such that a formal semantics for OCL messages and corresponding operators and operations could be given.

We identified the situation that a destination object (i.e., an object to which a message is sent) might no longer exist at the time of postcondition evaluation of the invoking operation. In turn, when an asynchronous operation call is dispatched or a signal sent is consumed in a destination object, the source object might already be destroyed. It is thus necessary to distinguish between "real" objects and their identifiers, in contrast to the current OCL semantics.

For OCL messages, we used explicit call identifiers to distinguish different messages. When returning from a synchronous operation call, we simply update the relevant parameter values by referring to the call identifiers. This is an abstraction from the UML semantics that assume that a specific reply object is generated and sent (OMG 2003, Section 2.24).

Our formal definition of a trace currently relies on a *global view* on the executed system. However, this cannot generally be assumed, e.g., for distributed systems. It has therefore to be discussed whether it is necessary to distinguish between local and global OCL constraints.

One important remaining task is to complete the counterpart semantics, i.e., the metamodel-based OCL semantics. In particular, a semantics of Statechart states is still missing in the metamodel-based OCL semantics. Moreover, consistency among the two semantics should be reviewed.

Several publications of recent years apply *temporal extensions* of OCL, e.g., Ziemann & Gogolla (2002). The dynamic semantics presented in this article is a suitable basis for a formal semantics of such temporal OCL extensions. In this context, a future-oriented temporal OCL extension w.r.t. reachability of Statechart configurations has already been developed (Flake & Mueller 2003).

Acknowledgement

This work receives funding through the DFG project GRASP within the DFG priority programme 1064 "Integration von Techniken der Softwarespezifikation für ingenieurwissenschaftliche Anwendungen".

References

Flake, S. & Mueller, W. (2003), Formal Semantics of Static and Temporal State-Oriented OCL Constraints, *Software and System Modeling (SoSyM)*, **2**(3), Springer.

Ivner, A., Högström, J., Johnston, S., Knox, D. & Rivett, S. (2003), 'Response to the UML2.0 OCL RfP, Version 1.6', OMG Document ad/03-01-07.

Kleppe, A. & Warmer, J. (2000), Extending OCL to Include Actions, *in* A. Evans, S. Kent & B. Selic, eds, 'UML 2000 - The Unified Modeling Language. Advancing the Standard', York, UK, LNCS 1939, Springer, pp. 440–450.

Kleppe, A. & Warmer, J. (2002), The Semantics of the OCL Action Clause, *in* T. Clark & J. Warmer, eds, 'Object Modeling with the OCL: The Rationale behind the Object Constraint Language', LNCS 2263, Springer, pp. 213–227.

OMG, Object Management Group (2003), 'Unified Modeling Language 1.5 Specification', OMG Document formal/03-03-01.

Richters, M. (2001), A Precise Approach to Validating UML Models and OCL Constraints, PhD thesis, Universität Bremen, Bremen, Germany.

Stumptner, M. & Schrefl, M. (2000), Behavior Consistent Inheritance in UML, *in* A.H.F. Laender *et al.*, eds, '19th International Conference on Conceptual Modeling (ER 2000)', Salt Lake City, UT, USA, pp. 527–542.

Warmer, J. & Kleppe, A. (1999), *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley.

Ziemann, P. & Gogolla, M. (2002), An Extension of OCL with Temporal Logic, *in* J. Jürjens *et al.*, eds, 'Critical Systems Development with UML', Technische Universität München, Institut für Informatik, Germany, pp. 53–62.