OCL: Syntax, Semantics and Tools

Mark Richters and Martin Gogolla

University of Bremen, FB 3, Computer Science Department
Postfach 330440, D-28334 Bremen, Germany
{mr|gogolla}@informatik.uni-bremen.de,
WWW home page: http://www.db.informatik.uni-bremen.de

Abstract. The Object Constraint Language OCL allows to formally specify constraints on a UML model. We present a formal syntax and semantics for OCL based on set theory including expressions, invariants and pre- and postconditions. A formal foundation for OCL makes the meaning of constraints precise and helps to eliminate ambiguities and inconsistencies. A precise language definition is also a prerequisite for implementing CASE tools providing enhanced support for UML models and OCL constraints. We give a survey of some OCL tools and discuss one of the tools in some more detail. The design and implementation of the USE tool supporting the validation of UML models and OCL constraints is based on the formal approach presented in this paper.

1 Introduction

The Unified Modeling Language (UML) [4, 19, 26] is a widely accepted standard for object-oriented modeling. The UML notation is largely based on diagrams. However, for certain aspects of a design, diagrams often do not provide the level of conciseness and expressiveness that a textual language can offer. Thus, textual annotations are frequently used to add details to a design. A special class of annotations are constraints that impose additional restrictions on a model. For this purpose, the Object Constraint Language (OCL) [18, 28] provides a framework for specifying constraints on a model in a formal way. OCL is a textual constraint language with a notational style similar to common object-oriented languages. OCL expressions are declarative and side effect-free. The language allows the modeler to specify constraints on a conceptual level helping to abstract from lower level implementation details.

Although designed to be a formal language, experience with OCL has shown that the language definition is not precise enough. Various authors have pointed out language issues related to ambiguities, inconsistencies or open interpretations [7, 11, 12]. In this paper, we present a formal foundation for OCL defining the abstract syntax and the semantics of OCL expressions, invariants, and pre- and postconditions. A formalization of OCL improves the language and helps to gain a more precise understanding of UML models and their interpretation. In our view, a precise language definition is also a prerequisite for implementing CASE tools providing enhanced support for UML models and OCL constraints.

We give a brief survey of OCL tools and discuss one of the tools in some more detail.

There are various different approaches in related work addressing formal aspects of OCL. A semantics for OCL without pre- and postconditions was first given in [22]. A graph-based semantics for OCL was developed by translating OCL constraints into expressions over graph rules [5]. A formal semantics was also provided by a mapping from OCL to a temporal logic [9]. An OCL extension to support temporal operators is proposed in [20]. The expressive power of OCL in terms of navigability and computability is discussed in [16]. Metamodels for OCL [1, 23] follow the metamodeling approach used in the UML standard. An approach for generating OCL constraints based on design patterns is described in [2]. Recently, a need for OCL features allowing behavioral constraints on occurrences of events, signals, and operation calls has been emphasized [15, 27].

The paper is structured as follows. Section 2 introduces a short example illustrating the application of OCL for specifying constraints. Section 3 defines UML object models and states providing the context for OCL. In Section 4, we briefly summarize the abstract syntax and semantics of OCL expressions which was first introduced in [22]. In addition to [22], we formally define the OCL notion of a context. Section 5 defines the syntax and semantics of pre- and postconditions. We explain the various additional syntactic possibilities and keywords like `result`, `isOclNew`, and `@pre` that may appear in postconditions. OCL tool support is discussed in Section 6. Section 7 presents some conclusions.

2 Specifying OCL Constraints

OCL is a textual language that allows to specify additional constraints on a UML model. A model always provides the context for constraints. Figure 1 shows a class diagram modeling employees, departments, and projects. Attributes like name, age, and salary represent properties that are common among all objects of a class. The operation raiseSalary can be invoked on employee objects. This is the only operation in our model to keep the example small. The operation signature defines a parameter and a return value of type Real. Relationships between the classes are modeled as associations WorksIn, WorksOn, and Controls.

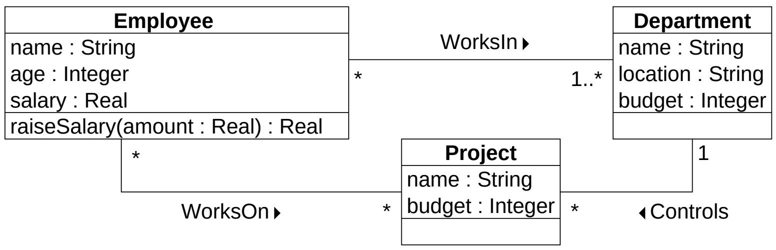


Fig. 1. UML class diagram of example model

OCL can be used to specify constraints concerning the static structure and the behavior of a system. Invariants are static structure constraints. An invariant is a condition that "must be true for all instances of that type at any time" [18, p. 7-6]. For example, the following invariant requires all department objects to have a non-negative budget.

```
context Department inv:
  self.budget >= 0
```

More complex constraints can be built by navigating along the associations between classes, for example:

```
-- Employees working on more projects than other employees
-- of the same department get a higher salary.
context Department inv:
  self.employee->forAll(e1, e2 |
    e1.project->size > e2.project->size
      implies e1.salary > e2.salary)
```

The `forAll` expression asserts a condition for each pair of employee objects working at the same department. The expression `e1.project` yields the set of projects the employee `e1` is working on. The OCL standard operation `size` determines the cardinality of that set. Role names like `project` and `employee` at the ends of associations are omitted in Figure 1. If no role name is given, the default in UML is to use the class name starting with a lowercase letter.

The behavioral interface of objects is defined by operations. Constraints on the behavior are specified in OCL by means of pre- and postconditions. Such a constraint defines a contract that an implementation of the operation has to fulfill [17]. It also provides the possibility for verifying the correctness of an implementation, e.g., in the style of Hoare logic [13].

```
-- If the amount is positive, raise
-- the salary by the given amount
context Employee::raiseSalary(amount : Real) : Real
  pre:  amount > 0
  post: self.salary = self.salary@pre + amount
        and result = self.salary
```

Pre- and postconditions will be discussed in detail in Section 5. In the next two sections, we will first discuss object models and simple OCL expressions that may appear in invariants. The definition for simple expressions is later extended to incorporate additional features that may appear only in postconditions.

3 Object Models and States

OCL expressions refer to various elements of a UML model. We therefore define an object model $\mathcal{M}$ to contain basically those elements of UML that are relevant for use with OCL. An object model

$$\mathcal{M} = (\textsc{Class}, \textsc{Att}_c, \textsc{Op}_c, \textsc{Assoc}, \mathrm{associates}, \mathrm{roles}, \mathrm{multiplicities}, \prec)$$

contains elements found in class diagrams. We only give an informal description of the components of an object model here. Examples in parentheses refer to the model shown in Figure 1. More details can be found in [22].

- $\text{C{\scriptsize LASS}}$ is a set of class names ($\text{C{\scriptsize LASS}} = \{\textit{Employee}, \textit{Department}, \textit{Project}\}$).
- $\text{A{\scriptsize TT}}_c$ is a set of operation signatures for functions mapping an object of class c to an associated attribute value ($\text{A{\scriptsize TT}}_{\text{Employee}} = \{\text{name} : \textit{Employee} \rightarrow \textit{String}, \dots\}$).
- $\text{O{\scriptsize P}}_c$ is a set of signatures for user-defined operations of a class c without side effects (these are tagged *isQuery* in a UML model, our example does not have any).
- $\text{A{\scriptsize SSOC}}$ is a set of association names ($\text{A{\scriptsize SSOC}} = \{\text{WorksIn}, \text{WorksOn}, \text{Controls}\}$).
  - associates is a function mapping each association name to a list of participating classes ($\text{associates} : \text{WorksIn} \mapsto \langle \textit{Employee}, \textit{Department} \rangle$).
  - roles is a function assigning each end of an association a role name ($\text{roles} : \text{WorksIn} \mapsto \langle \textit{employee}, \textit{department} \rangle$).
  - multiplicities is a function assigning each end of an association a multiplicity specification ($\text{multiplicities} : \text{WorksIn} \mapsto \langle \mathbb{N}_0, \mathbb{N} \rangle$).
- $\prec$ is a partial order on $\text{C{\scriptsize LASS}}$ reflecting the generalization hierarchy of classes (there is no generalization used in our example, therefore $\prec = \emptyset$).

The interpretation of an object model is the set of possible system states. A system state includes objects, links and attribute values. A system may be in different states as it changes over time. Therefore, a system state is also called a snapshot of a running system. With respect to OCL, we can, in many cases, concentrate on a single system state given at a discrete point in time. For example, a system state provides the complete context for the evaluation of class invariants. For pre- and postconditions, however, it is necessary to consider two consecutive states (see Section 5).

A single system state for an object model $\mathcal{M}$ is a structure $\sigma(\mathcal{M}) = (\sigma_{\text{C{\scriptsize LASS}}}, \sigma_{\text{A{\scriptsize TT}}}, \sigma_{\text{A{\scriptsize SSOC}}})$ where the finite sets $\sigma_{\text{C{\scriptsize LASS}}}(c)$ contain all objects of a class $c \in \text{C{\scriptsize LASS}}$ currently existing in the system state, functions $\sigma_{\text{A{\scriptsize TT}}}$ assign attribute values to each object, and the finite sets $\sigma_{\text{A{\scriptsize SSOC}}}(as)$ contain links connecting objects for each association $as \in \text{A{\scriptsize SSOC}}$.

4 OCL Expressions

The definition of OCL expressions is based upon a signature $\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$ for an object model $\mathcal{M}$ providing a set of types $T_{\mathcal{M}}$, a relation $\leq$ on types reflecting the type hierarchy, and a set of operations $\Omega_{\mathcal{M}}$. The signature contains the initial set of syntactic elements upon which we build the expression syntax.

For example, the signature for the model in Fig. 1 is

$$\begin{aligned}\Sigma_{\mathcal{M}} = (&\{\mathit{Employee}, \mathit{Department}, \mathit{Project}, \mathit{Integer}, \mathit{Real}, \mathit{Set}(\mathit{Project}), \dots\},\\ &\{\mathit{Integer} \le \mathit{Real}, \dots\},\\ &\{+ : \mathit{Integer} \times \mathit{Integer} \to \mathit{Integer},\\ &\ \mathrm{name} : \mathit{Employee} \to \mathit{String},\\ &\ \mathrm{project} : \mathit{Employee} \to \mathit{Set}(\mathit{Project}), \dots\})\ .\end{aligned}$$

The set of types $T_{\mathcal{M}}$ includes type expressions for constructing the OCL collection types *Collection*(t), *Set*(t), *Sequence*(t), and *Bag*(t) which are parameterized by an element type t. Note that $\Omega_{\mathcal{M}}$ only includes signatures of side effect-free operations such as standard arithmetic, attribute access and navigation by role names. These operations are intended to be available as part of side effect-free OCL expressions, for example, in invariants. The operation raiseSalary in class *Employee* is expected to cause side effects to the system state and is therefore not included in the signature.

A semantics for a data signature $\Sigma_{\mathcal{M}}$ is a mapping associating each type $t \in T_{\mathcal{M}}$ with a domain $I(t)$ and each operation $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_{\mathcal{M}}$ with a function $I(\omega)(\sigma) : I(t_1) \times \cdots \times I(t_n) \to I(t)$. The parameter σ denotes a system state with a set of objects, their attribute values and association links. It is required for evaluating attribute access and navigation operations.

4.1 Syntax of Expressions

We define the syntax of expressions inductively so that more complex expressions are recursively built from simple structures. For each expression the set of free occurrences of variables is also defined. The latter is necessary for determining the scope of a variable and the possible context of an expression as discussed in Section 4.3.

Definition 1 (Syntax of expressions)
Let $\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \le, \Omega_{\mathcal{M}})$ be a data signature over an object model $\mathcal{M}$. Let $\mathrm{Var} = \{\mathrm{Var}_t\}_{t \in T_{\mathcal{M}}}$ be a family of variable sets where each variable set is indexed by a type t. The syntax of expressions over the signature $\Sigma_{\mathcal{M}}$ is given by a set $\mathrm{Expr} = \{\mathrm{Expr}_t\}_{t \in T_{\mathcal{M}}}$ and a function $\mathrm{free} : \mathrm{Expr} \to \mathcal{F}(\mathrm{Var})$ defined as follows.

i. If $v \in \mathrm{Var}_t$ then $\boldsymbol{v} \in \mathrm{Expr}_t$ and $\mathrm{free}(v) := \{v\}$.
ii. If $v \in \mathrm{Var}_{t_1}, e_1 \in \mathrm{Expr}_{t_1}, e_2 \in \mathrm{Expr}_{t_2}$ then **let** $\boldsymbol{v = e_1}$ **in** $\boldsymbol{e_2} \in \mathrm{Expr}_{t_2}$ and $\mathrm{free}(\mathrm{let}\ v = e_1\ \mathrm{in}\ e_2) := \mathrm{free}(e_2) - \{v\}$.
iii. If $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_{\mathcal{M}}$ and $e_i \in \mathrm{Expr}_{t_i}$ for all $i = 1, \dots, n$ then $\boldsymbol{\omega(e_1, \dots, e_n)} \in \mathrm{Expr}_t$ and $\mathrm{free}(\omega(e_1, \dots, e_n)) := \mathrm{free}(e_1) \cup \cdots \cup \mathrm{free}(e_n)$.
iv. If $e_1 \in \mathrm{Expr}_{\mathrm{Boolean}}$ and $e_2, e_3 \in \mathrm{Expr}_t$ then **if** $\boldsymbol{e_1}$ **then** $\boldsymbol{e_2}$ **else** $\boldsymbol{e_3}$ **endif** $\in \mathrm{Expr}_t$ and $\mathrm{free}(\mathrm{if}\ e_1\ \mathrm{then}\ e_2\ \mathrm{else}\ e_3\ \mathrm{endif}) := \mathrm{free}(e_1) \cup \mathrm{free}(e_2) \cup \mathrm{free}(e_3)$.

v. If $e \in \mathrm{Expr}_t$ and $t' \leq t$ or $t \leq t'$ then $\boldsymbol{(e \text{ asType } t')} \in \mathrm{Expr}_{t'}$, $\boldsymbol{(e \text{ isTypeOf } t')} \in \mathrm{Expr}_{\mathrm{Boolean}}$, $\boldsymbol{(e \text{ isKindOf } t')} \in \mathrm{Expr}_{\mathrm{Boolean}}$ and $\mathrm{free}((e \text{ asType } t')) := \mathrm{free}(e)$, $\mathrm{free}((e \text{ isTypeOf } t')) := \mathrm{free}(e)$, $\mathrm{free}((e \text{ isKindOf } t')) := \mathrm{free}(e)$.

vi. If $e_1 \in \mathrm{Expr}_{\mathit{Collection}(t_1)}$, $v_1 \in \mathrm{Var}_{t_1}, v_2 \in \mathrm{Var}_{t_2}$, and $e_2, e_3 \in \mathrm{Expr}_{t_2}$ then $\boldsymbol{e_1 \rightarrow \mathbf{iterate}(v_1; v_2 = e_2 \mid e_3)} \in \mathrm{Expr}_{t_2}$ and $\mathrm{free}(e_1 \rightarrow \mathrm{iterate}(v_1; v_2 = e_2 \mid e_3)) := (\mathrm{free}(e_1) \cup \mathrm{free}(e_2) \cup \mathrm{free}(e_3)) - \{v_1, v_2\}$.

An expression of type t' is also an expression of a more general type t. For all $t' \leq t$: if $e \in \mathrm{Expr}_{t'}$ then $e \in \mathrm{Expr}_t$. □

A variable expression (i) refers to the value of a variable. Variables (including the special variable `self`) may be introduced by the context of an expression, as part of an iterate expression, and by a let expression. Let expressions (ii) do not add to the expressiveness of OCL but help to avoid repetitions of common sub-expressions. Operation expressions (iii) apply an operation from $\Omega_{\mathcal{M}}$. The set of operations includes:

– predefined data operations: `+`, `-`, `*`, `<`, `>`, `size`, `max`
– attribute operations: `self.age`, `e.salary`
– side effect-free operations defined by a class
– navigation by role names: `e.employee`
– constants: `25`, `'aString'`

As demonstrated by the examples, an operation expression may also be written in OCL path syntax as $e_1.\omega(e_2, \ldots, e_n)$. This notational style is common in many object-oriented languages. It emphasizes the role of the first argument as the "receiver" of a "message". If e_1 denotes a collection value, an arrow symbol is used in OCL instead of the period: $e_1 \rightarrow \omega(e_2, \ldots, e_n)$. Collections may be bags, sets, or lists. An if-expression (iv) provides an alternative selection of two expressions depending on the result of a condition given by a boolean expression.

An asType expression (v) can be used in cases where static type information is insufficient. It corresponds to the `oclAsType` operation in OCL and can be understood as a cast of a source expression to an equivalent expression of a (usually) more specific target type. The target type must be related to the source type, that is, one must be a subtype of the other. The isTypeOf and isKindOf expressions correspond to the `oclIsTypeOf` and `oclIsKindOf` operations, respectively. An expression (e isTypeOf t') can be used to test whether the type of the value resulting from the expression e has the type t' given as argument. An isKindOf expression (e isKindOf t') is not as strict in that it is sufficient for the expression to become true if t' is a supertype of the type of the value of e. Note that OCL defines these type casts and tests as operations with parameters of type *OclType*. In contrast to OCL, we technically define them as first class expressions which has the benefit that we do not need the metatype *OclType*. Thus the type system is kept simple while preserving compatibility with standard OCL syntax. A related discussion about the removal of OCL metatypes can be found in the paper by Rumpe in this proceedings.

An iterate expression (vi) is a general loop construct which evaluates an argument expression e_3 repeatedly for all elements of a collection which is given by a source expression e_1. Each element of the collection is bound in turn to the variable v_1 for each evaluation of the argument expression. The argument expression e_3 may contain the variable v_1 to refer to the current element of the collection. The result variable v_2 is initialized with the expression e_2. After each evaluation of the argument expression e_3, the result is bound to the variable v_2. The final value of v_2 is the result of the whole expression. The iterate construct is probably the most important kind of expression in OCL. We will shortly see how other OCL constructs can be equivalently defined in terms of an iterate expression.

4.2 Semantics of Expressions

The semantics of expressions is made precise in the following definition. A context for evaluation is given by an environment $\tau = (\sigma, \beta)$ consisting of a system state σ and a variable assignment $\beta : \mathrm{Var}_t \to I(t)$. A system state σ provides access to the set of currently existing objects, their attribute values, and association links between objects. A variable assignment β maps variable names to values.

Definition 2 (Semantics of expressions)
Let Env be the set of environments $\tau = (\sigma, \beta)$. The semantics of an expression $e \in \mathrm{Expr}_t$ is a function $I[\![\, e \,]\!] : \mathrm{Env} \to I(t)$ that is defined as follows.

i. $I[\![\, v \,]\!](\tau) = \beta(v)$.
ii. $I[\![\, \text{let } v = e_1 \text{ in } e_2 \,]\!](\tau) = I[\![\, e_2 \,]\!](\sigma, \beta\{v / I[\![\, e_1 \,]\!](\tau)\})$.
iii. $I[\![\, \omega(e_1, \dots, e_n) \,]\!](\tau) = I(\omega)(\tau)(I[\![\, e_1 \,]\!](\tau), \dots, I[\![\, e_n \,]\!](\tau))$.
iv. $I[\![\, \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ endif} \,]\!](\tau) = \begin{cases} I[\![\, e_2 \,]\!](\tau) & \text{if } I[\![\, e_1 \,]\!](\tau) = \text{true}, \\ I[\![\, e_3 \,]\!](\tau) & \text{if } I[\![\, e_1 \,]\!](\tau) = \text{false}, \\ \bot & \text{otherwise.} \end{cases}$
v. $I[\![\, (e \text{ asType } t') \,]\!](\tau) = \begin{cases} I[\![\, e \,]\!](\tau) & \text{if } I[\![\, e \,]\!](\tau) \in I(t'), \\ \bot & \text{otherwise.} \end{cases}$

$I[\![\, (e \text{ isTypeOf } t') \,]\!](\tau) = \begin{cases} \text{true} & \text{if } I[\![\, e \,]\!](\tau) \in I(t') - \bigcup_{t'' < t'} I(t''), \\ \text{false} & \text{otherwise.} \end{cases}$

$I[\![\, (e \text{ isKindOf } t') \,]\!](\tau) = \begin{cases} \text{true} & \text{if } I[\![\, e \,]\!](\tau) \in I(t'), \\ \text{false} & \text{otherwise.} \end{cases}$

vi. $I[\![\, e_1 \to \text{iterate}(v_1; v_2 = e_2 \mid e_3) \,]\!](\tau) = I[\![\, e_1 \to \text{iterate}'(v_1 \mid e_3) \,]\!](\tau')$ where $\tau' = (\sigma, \beta')$ and $\tau'' = (\sigma, \beta'')$ are environments with modified variable assignments

$$\begin{aligned} \beta' &:= \beta\{v_2 / I[\![\, e_2 \,]\!](\tau)\} \\ \beta'' &:= \beta'\{v_2 / I[\![\, e_3 \,]\!](\sigma, \beta'\{v_1 / x_1\})\} \end{aligned}$$

and iterate$'$ is defined as:[1]

(a) If $e_1 \in \mathrm{Expr}_{Sequence(t_1)}$ then $I[\![\, e_1 \to \mathrm{iterate}'(v_1 \mid e_3) \,]\!](\tau') =$

$$\begin{cases} I[\![\, v_2 \,]\!](\tau') \\ \quad \text{if } I[\![\, e_1 \,]\!](\tau') = \langle\rangle, \\ I[\![\, \mathrm{mkSequence}_{t_1}(x_2, \dots, x_n) \to \mathrm{iterate}'(v_1 \mid e_3) \,]\!](\tau'') \\ \quad \text{if } I[\![\, e_1 \,]\!](\tau') = \langle x_1, \dots, x_n \rangle. \end{cases}$$

(b) If $e_1 \in \mathrm{Expr}_{Set(t_1)}$ then $I[\![\, e_1 \to \mathrm{iterate}'(v_1 \mid e_3) \,]\!](\tau') =$

$$\begin{cases} I[\![\, v_2 \,]\!](\tau') \\ \quad \text{if } I[\![\, e_1 \,]\!](\tau') = \emptyset, \\ I[\![\, \mathrm{mkSet}_{t_1}(x_2, \dots, x_n) \to \mathrm{iterate}'(v_1 \mid e_3) \,]\!](\tau'') \\ \quad \text{if } I[\![\, e_1 \,]\!](\tau') = \{x_1, \dots, x_n\}. \end{cases}$$

(c) If $e_1 \in \mathrm{Expr}_{Bag(t_1)}$ then $I[\![\, e_1 \to \mathrm{iterate}'(v_1 \mid e_3) \,]\!](\tau') =$

$$\begin{cases} I[\![\, v_2 \,]\!](\tau') \\ \quad \text{if } I[\![\, e_1 \,]\!](\tau') = \emptyset, \\ I[\![\, \mathrm{mkBag}_{t_1}(x_2, \dots, x_n) \to \mathrm{iterate}'(v_1 \mid e_3) \,]\!](\tau'') \\ \quad \text{if } I[\![\, e_1 \,]\!](\tau') = \{\!\{x_1, \dots, x_n\}\!\}. \end{cases}$$

□

The semantics of a variable expression (i) is the value assigned to the variable. A let expression (ii) results in the value of the sub-expression e_2. Free occurrences of the variable v in e_2 are bound to the value of the expression e_1. An operation expression (iii) is interpreted by the function associated with the operation. Each argument expression is evaluated separately. The result of an if-expression (iv) is given by the then-part if the condition is true. If the condition is false, the else-part is the result of the expression. The result of a cast expression (v) using asType is the value of the expression, if the value lies within the domain of the specified target type, otherwise it is undefined. A type test expression with isTypeOf is true if the expression value lies exactly within the domain of the specified target type without considering subtypes. An isKindOf type test expression is true if the expression value lies within the domain of the specified target type *or* one of its subtypes. An iterate expression (vi) loops over the elements of a collection and allows the application of a function to each collection element. The function results are successively combined into a value that serves as the result of the whole iterate expression. This kind of evaluation is also known in functional style programming languages as *fold* operation.

One must be careful when using iterate expression on sets and bags. For sets and bags as source collection there may be expressions where the result of the whole iterate expression depends on the order in which collection elements are selected for application. The following expression concatenates the names of all employees working on a project. However, the names in the resulting string may

[1] The constructor operations $\mathrm{mkSequence}_t$, mkBag_t, and mkSet_t are in $\Omega_{\mathcal{M}}$ and provide the abstract syntax for collection literals like `Set{1,2}` in concrete OCL syntax.

appear in any order, because it is unspecified in which order the elements in the set `p.employee` are applied.

```
p.employee->iterate(e : Employee;
  names : String = '' | names.concat(e.name))
```

A number of important OCL constructs such as `exists`, `forAll`, `select`, `reject`, `collect`, and `isUnique` are defined in terms of iterate expressions. In [18] the intended semantics of these expressions is given by postconditions with iterate-based expressions. The following schema shows how these expressions can be translated to equivalent iterate expressions. A similar translation is given in [6].

$$
\begin{aligned}
&I[\![\, e_1 \rightarrow \text{exists}(v_1 \mid e_3) \,]\!](\tau) = \\
&\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{false} \mid v_2 \text{ or } e_3) \,]\!](\tau)
\end{aligned}
$$

$$
\begin{aligned}
&I[\![\, e_1 \rightarrow \text{forAll}(v_1 \mid e_3) \,]\!](\tau) = \\
&\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{true} \mid v_2 \text{ and } e_3) \,]\!](\tau)
\end{aligned}
$$

$$
\begin{aligned}
&I[\![\, e_1 \rightarrow \text{select}(v_1 \mid e_3) \,]\!](\tau) = \\
&\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = e_1 \mid \\
&\qquad \text{if } e_3 \text{ then } v_2 \text{ else } v_2 \rightarrow \text{excluding}(v_1) \text{ endif}) \,]\!](\tau)
\end{aligned}
$$

$$
\begin{aligned}
&I[\![\, e_1 \rightarrow \text{reject}(v_1 \mid e_3) \,]\!](\tau) = \\
&\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = e_1 \mid \\
&\qquad \text{if } e_3 \text{ then } v_2 \rightarrow \text{excluding}(v_1) \text{ else } v_2 \text{ endif}) \,]\!](\tau)
\end{aligned}
$$

$$
\begin{aligned}
&I[\![\, e_1 \rightarrow \text{collect}(v_1 \mid e_3) \,]\!](\tau) = \\
&\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{mkBag}_{\textit{type-of-}e_3} \mid \\
&\qquad v_2 \rightarrow \text{including}(v_1)) \,]\!](\tau)
\end{aligned}
$$

$$
\begin{aligned}
&I[\![\, e_1 \rightarrow \text{isUnique}(v_1 \mid e_3) \,]\!](\tau) = \\
&\quad I[\![\, e_1 \rightarrow \text{iterate}(v1; v2 = \text{true} \mid \\
&\qquad v_2 \text{ and } e_1 \rightarrow \text{count}(v_1) = 1) \,]\!](\tau)
\end{aligned}
$$

4.3 Expression Context

An OCL expression is always written in some syntactical context. Since the primary purpose of OCL is the specification of constraints on a UML model, it is obvious that the model itself provides the most general kind of context. In our approach, the signature $\Sigma_{\mathcal{M}}$ contains types (e.g., object types) and operations (e.g., attribute operations) that are imported from a model, thus providing a context for building expressions that depend on the elements of a specific model.

On a much smaller scale, there is also a notion of context in OCL that simply introduces variable declarations. This notion is closely related to the syntax for constraints written in OCL. A context clause declares variables in invariants, and parameters in pre- and postconditions. The following example declares a variable `e` which is subsequently used in an invariant expression.

```
context e : Employee inv:
  e.age > 18
```

The next example declares a parameter `amount` which is used in a pre- and postcondition specification.

```
context Employee::raiseSalary(amount : Real) : Real
  pre:  amount > 0
  post: self.salary = self.salary@pre + amount
        and result = self.salary
```

Here we use the second meaning of context, that is, a context provides a set of variable declarations. The more general meaning of context is already subsumed by our concept of a signature as described above. A similar distinction between *local* and *global* declarations is also made in [8]. In their paper, the authors extend the OCL context syntax to include global declarations and outline a general approach to derive declarations from information on the UML metamodel level.

A *context for an invariant* (corresponding to the nonterminal `classifier-Context` in the OCL grammar [18]) is a declaration of variables. The variable declaration may be implicit or explicit. In the implicit form, the context is written as

context C **inv:**
  `<expression>`

In this case, the `<expression>` may use the variable `self` of type C as a free variable. In the explicit form, the context is written as

context $v_1 : C_1, \ldots, v_n : C_n$ **inv:**
  `<expression>`

The `<expression>` may use the variables $v_1, \ldots, v_n$ of types $C_1, \ldots, C_n$ as free variables. The OCL grammar actually only allows the explicit declaration of at most one variable in a classifierContext. This restriction seems unnecessarily strict. Having multiple variables is especially useful for constraints specifying key properties of attributes. The example (taken from [18, p. 7-18])

```
context Person inv:
  Person.allInstances->forAll(p1, p2 |
    p1 <> p2 implies p1.name <> p2.name)
```

could then be just written as:

```
context p1, p2 : Person inv:
  p1 <> p2 implies p1.name <> p2.name
```

A *context for a pre-/postcondition* (corresponding to the nonterminal `operationContext` in the OCL grammar) is a declaration of variables. In this case, the context is written as

```
context C :: op(p_1 : T_1, ..., p_n : T_n) : T
  pre: P
  post: Q
```

This means that the variable `self` (of type C) and the parameters $p_1, \ldots, p_n$ may be used as free variables in the precondition P and the postcondition Q. Additionally, the postcondition may use `result` (of type T) as a free variable. The details are explained in Section 5.

An *invariant* is an expression with boolean result type and a set of (explicitly or implicitly declared) free variables $v_1 : C_1, \ldots, v_n : C_n$ where $C_1, \ldots, C_n$ are classifier types. An invariant

```
context v_1 : C_1, ..., v_n : C_n inv:
  <expression>
```

is equivalent to the following expression without free variables that must be valid in all system states.

```
C_1.allInstances->forAll(v_1 : C_1 |
   ...
   C_n.allInstances->forAll(v_n : C_n |
     <expression>
   )
   ...
)
```

5 Pre- and Postconditions

The definition of expressions in Section 4 is sufficient for invariants and queries where we have to consider only single system states. For pre- and postconditions, there are additional language constructs in OCL which enable references to the system state before the execution of an operation and to the system state that results from the operation execution. The general syntax of an operation specification with pre- and postconditions is defined as

```
context C :: op(p_1 : T_1, ..., p_n : T_n)
  pre: P
  post: Q
```

First, the context is determined by giving the signature of the operation for which pre- and postconditions are to be specified. The operation `op` which is defined as part of the classifier C has a set of typed parameters $\textsc{Params}_{\mathrm{op}} = \{p_1, \ldots, p_n\}$. The UML model providing the definition of an operation signature also specifies the direction kind of each parameter. We use a function $kind : \textsc{Params}_{\mathrm{op}} \to \{\mathrm{in}, \mathrm{out}, \mathrm{inout}, \mathrm{return}\}$ to map each parameter to one of these kinds. Although UML makes no restriction on the number of return parameters, there is usually only at most one return parameter considered in OCL which is referred to by the keyword `result` in a postcondition. In this case, the signature is also written as $C :: \mathtt{op}(p_1 : T_1, \ldots, p_{n-1} : T_{n-1}) : T$ with T being the type of the `result` parameter.

The precondition of the operation is given by an expression P, and the postcondition is specified by an expression Q. P and Q must have a boolean result type. If the precondition holds, the contract of the operation guarantees that the postcondition is satisfied after completion of `op`. Pre- and postconditions form a pair. A condition defaults to true if it is not explicitly specified.

Note that in the previous section, we have talked about side effect-free operations. Now we are discussing operations that usually have side effects. Table 1 summarizes different kinds of operations in UML. Operations in the table are classified by the existence of a return parameter in the signature, whether they are declared as being side effect-free (with the tag *isQuery* in UML), the state before and after execution, and the languages in which (1) the operation body can be expressed (Body), and (2) the operation may be called (Caller).

Table 1. Different kinds of operations in UML

Return value	side effect-free	States	Body	Caller
no	no	pre-state $\neq$ post-state allowed	AL	AL
yes	no	pre-state $\neq$ post-state allowed	AL	AL
yes	yes	pre-state = post-state required	OCL	OCL, AL

The first row of the table describes operations without a return value. These are used to specify side effects on a system state. Therefore, the post-state usually differs from the state before the operation call. Since specifying side effects is out of the scope of OCL expressions, the body of the operation must be expressed in some kind of *Action Language* (AL). Furthermore, the operation cannot be used without restriction as part of an OCL expression because all operations in an OCL expression must be tagged *isQuery*. The same arguments apply to operations with a return value that are listed in the second row. The third kind of operations are those operations which may be used in OCL without restrictions. Because their execution does not have side effects, the pre- and post-states are always equal. Often, the body of the operation can be specified with an OCL expression. It might be desirable for an action language to make use of these kinds of operations by including OCL as a sub-language.

5.1 Motivating Example

Before we give a formal definition of operation specifications with pre- and postconditions, we demonstrate the fundamental concepts by means of an example. Figure 2 shows a class diagram with two classes A and B that are related to each other by an association R. Class A has an operation `op()` but no attributes. Class B has an attribute c and no operations. The implicit role names `a` and `b` at the association ends allow navigation in OCL expressions from a B object to the associated A object and vice versa.

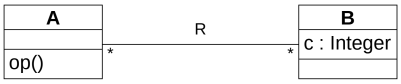
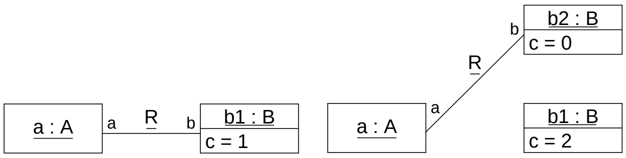


Fig. 2. Example class diagram

Figure 3 shows an example for two consecutive states of a system corresponding to the given class model. The object diagrams show instances of classes A and B and links of the association R. The left object diagram shows the state before the execution of an operation, whereas the right diagram shows the state after the operation has been executed. The effect of the operation can be described by the following changes in the post-state: (1) the value of the attribute c in object $\underline{b}_1$ has been incremented by one, (2) a new object $\underline{b}_2$ has been created, (3) the link between $\underline{a}$ and $\underline{b}_1$ has been removed, and (4) a new link between $\underline{a}$ and $\underline{b}_2$ has been established.



(a) Pre-state with objects $\underline{a}$ and $\underline{b}_1$.

(b) Post-state. The object $\underline{b}_2$ did not exist in the pre-state.

Fig. 3. Object diagrams showing a pre- and a post-state

For the following discussion, consider the OCL expression `a.b.c` where `a` is a variable denoting the object $\underline{a}$. The expression navigates to the associated object of class B and results in the value of the attribute c. Therefore, the expression evaluates to 1 in the pre-state shown in Figure 3(a).

As an example of how the OCL modifier `@pre` may be used in a postcondition to refer to properties of the previous state, we now look at some variations of the expression `a.b.c` that may appear as part of a postcondition. For each case, the result is given and explained.

– `a.b.c = 0`
  Because the expression is completely evaluated in the post-state, the navi-

gation from $\underline{a}$ leads to the $\underline{b}_2$ object. The value of the attribute c of $\underline{b}_2$ is 0 in Figure 3(b).

– `a.b@pre.c = 2`
  This expression refers to both the pre- and the post-state. The previous value of `a.b` is a reference to object $\underline{b}_1$. However, since the `@pre` modifier only applies to the expression `a.b`, the following reference to the attribute c is evaluated in the post-state of $\underline{b}_1$, even though $\underline{b}_1$ is not connected anymore to $\underline{a}$. Therefore, the result is 2.
– `a.b@pre.c@pre = 1`
  In this case, the value of the attribute c of object $\underline{b}_1$ is taken from the pre-state. This expression is semantically equivalent to the expression `a.b.c` in a precondition.
– `a.b.c@pre = ⊥`
  The expression `a.b` evaluated in the post-state yields a reference to object $\underline{b}_2$ which is now connected to $\underline{a}$. Since $\underline{b}_2$ has just been created by the operation, there is no previous state of $\underline{b}_2$. Hence, a reference to the previous value of attribute c is undefined.

Note that the `@pre` modifier may only be applied to operations not to arbitrary expressions. An expression such as `(a.b)@pre` is syntactically illegal.

OCL provides the standard operation `oclIsNew` for checking whether an object has been created during the execution of an operation. This operation may only be used in postconditions. For our example, the following conditions indicate that the object $\underline{b}_2$ has just been created in the post-state and $\underline{b}_1$ already existed in the pre-state.

– `a.b.oclIsNew = true`
– `a.b@pre.oclIsNew = false`

5.2 Syntax and Semantics of Postconditions

All common OCL expressions can be used in a precondition P. Syntax and semantics of preconditions are defined exactly like those for plain OCL expressions in Section 4. Also, all common OCL expressions can be used in a postcondition Q. Additionally, the `@pre` construct, the special variable `result`, and the operation `oclIsNew` may appear in a postcondition. In the following, we extend Definition 1 for the syntax of OCL expressions to provide these additional features.

Definition 3 (Syntax of expressions in postconditions)
Let op be an operation with a set of parameters $\textsc{Params}_{\mathrm{op}}$. The set of parameters includes at most one parameter of kind "return". The basic set of expressions in postconditions is defined by repeating Definition 1 while substituting all occurrences of Expr_t with Post-Expr_t. Furthermore, we define that

– Each non-return parameter $p \in \textsc{Params}_{\mathrm{op}}$ with a declared type t is available as variable: $p \in \mathrm{Var}_t$.

– If $\textsc{Params}_{\text{op}}$ contains a parameter of kind "return" and type t then `result` is a variable: $\texttt{result} \in \text{Var}_t$.
– The operation $\texttt{oclIsNew} : c \to \mathit{Boolean}$ is in $\Omega_{\mathcal{M}}$ for all object types $c \in T_{\mathcal{M}}$.

The syntax of expressions in postconditions is extended by the following rule.

vii. If $\omega : t_1 \times \cdots \times t_n \to t \in \Omega_{\mathcal{M}}$ and $e_i \in \text{Post-Expr}_{t'}$ for all $i = 1, \ldots, n$ then $\boldsymbol{\omega_{@\text{pre}}(e_1, \ldots, e_n)} \in \text{Post-Expr}_t$.

□

All general OCL expressions may be used in a postcondition. Moreover, the basic rules for recursively constructing expressions do also apply. Operation parameters are added to the set of variables. For operations with a return type, the variable `result` refers to the operation result. The set of operations is extended by `oclIsNew` which is defined for all object types. Operations $\omega_{@\text{pre}}$ are added for allowing references to the previous state (vii). The rule says that the `@pre` modifier may be applied to all operations, although, in general, not all operations do actually depend on a system state (for example, operations on data types). The result of these operations will be the same in all states. Operations which do depend on a system state are, e.g., attribute access and navigation operations.

For a definition of the semantics of postconditions, we will refer to *environments* describing the previous state and the state resulting from executing the operation. An environment $\tau = (\sigma, \beta)$ is a pair consisting of a system state σ and a variable assignment β (see Section 4.2). The necessity of including variable assignments into environments will be discussed shortly. We call an environment $\tau_{\text{pre}} = (\sigma_{\text{pre}}, \beta_{\text{pre}})$ describing a system state and variable assignments before the execution of an operation a *pre-environment*. Likewise, an environment $\tau_{\text{post}} = (\sigma_{\text{post}}, \beta_{\text{post}})$ after the completion of an operation is called a *post-environment*.

Definition 4 (Semantics of postcondition expressions)
Let Env be the set of environments. The semantics of an expression $e \in \text{Post-Expr}_t$ is a function $I[\![e]\!] : \text{Env} \times \text{Env} \to I(t)$. The semantics of the basic set of expressions in postconditions is defined by repeating Definition 2 while substituting all occurrences of Expr_t with Post-Expr_t. References to $I[\![e]\!](\tau)$ are replaced by $I[\![e]\!](\tau_{\text{pre}}, \tau_{\text{post}})$ to include the pre-environment. Occurrences of τ are changed to τ_{post} which is the default environment in a postcondition.

– For all $p \in \textsc{Params}_{\text{op}} : I[\![p]\!](\tau_{\text{pre}}, \tau_{\text{post}}) = \beta_{\text{post}}(p)$.
  • Input parameters may not be modified by an operation:
  $\text{kind}(p) = \text{in}$ implies $\beta_{\text{pre}}(p) = \beta_{\text{post}}(p)$.
  • Output parameters are undefined on entry:
  $\text{kind}(p) = \text{out}$ implies $\beta_{\text{pre}}(p) = \bot$.
– $I[\![\texttt{result}]\!](\tau_{\text{pre}}, \tau_{\text{post}}) = \beta_{\text{post}}(\texttt{result})$.
– $I[\![\texttt{oclIsNew}]\!](\tau_{\text{pre}}, \tau_{\text{post}})(\underline{c}) = \begin{cases} \text{true} & \text{if } \underline{c} \notin \sigma_{\text{pre}}(c), \\ \text{false} & \text{otherwise.} \end{cases}$

vii. $I[\![\, \omega_{@\mathrm{pre}}(e_1, \ldots, e_n) \,]\!](\tau_{\mathrm{pre}}, \tau_{\mathrm{post}}) =$
$I(\omega)(\tau_{\mathrm{pre}})(I[\![\, e_1 \,]\!](\tau_{\mathrm{pre}}, \tau_{\mathrm{post}}), \ldots, I[\![\, e_n \,]\!](\tau_{\mathrm{pre}}, \tau_{\mathrm{post}}))$

□

Standard expressions are evaluated as defined in Definition 2 with the post-environment determining the context of evaluation. Input parameters do not change during the execution of the operation. Therefore, their values are equal in the pre- and post-environment. The value of the `result` variable is determined by the variable assignment of the post-environment. The `oclIsNew` operation yields true if an object did not exist in the previous system state. Operations referring to the previous state are evaluated in context of the pre-environment (vii). Note that the operation arguments may still be evaluated in the post-environment. Therefore, in a nested expression, the environment only applies to the current operation, whereas deeper nested operations may evaluate in a different environment.

With these preparations, the semantics of an operation specification with pre- and postconditions can be precisely defined as follows. We say that a precondition P *satisfies* a pre-environment τ_{pre} – written as $\tau_{\mathrm{pre}} \models P$ – if the expression P evaluates to true according to Definition 2. Similarly, a postcondition Q satisfies a pair of pre- and post-environments, if the expression Q evaluates to true according to Definition 4:

$$\begin{aligned} \tau_{\mathrm{pre}} \models P \quad &\text{iff} \quad I[\![\, P \,]\!](\tau_{\mathrm{pre}}) = \mathrm{true} \\ (\tau_{\mathrm{pre}}, \tau_{\mathrm{post}}) \models Q \quad &\text{iff} \quad I[\![\, Q \,]\!](\tau_{\mathrm{pre}}, \tau_{\mathrm{post}}) = \mathrm{true} \end{aligned}$$

Definition 5 (Semantics of operation specifications)
The semantics of an operation specification is a set $R \subseteq \mathrm{Env} \times \mathrm{Env}$ defined as

$$\begin{aligned} I[\![\;& \textbf{context } C :: \mathsf{op}(p_1 : T_1, \ldots, p_n : T_n) \\ & \quad \textbf{pre: } P \\ & \quad \textbf{post: } Q \;]\!] \;=\; R \end{aligned}$$

where R is the set of all pre- and post-environment pairs such that the pre-environment τ_{pre} satisfies the precondition P and the pair of both environments satisfies the postcondition Q:

$$R = \{(\tau_{\mathrm{pre}}, \tau_{\mathrm{post}}) \mid \tau_{\mathrm{pre}} \models P \;\wedge\; (\tau_{\mathrm{pre}}, \tau_{\mathrm{post}}) \models Q\}$$

□

The satisfaction relation for Q is defined in terms of both environments since the postcondition may contain references to the previous state. The set R defines all legal transitions between two states corresponding to the effect of an operation. It therefore provides a framework for a correct implementation.

Definition 6 (Satisfaction of operation specifications)
An operation specification with pre- and postconditions is satisfied by a program S in the sense of total correctness if the computation of S is a total function $f_S : \mathrm{dom}(R) \to \mathrm{im}(R)$ and $\mathrm{graph}(f_S) \subseteq R$. □

In other words, the program S accepts each environment satisfying the precondition as input and produces an environment that satisfies the postcondition. The definition of R allows us to make some statements about the specification. In general, a reasonable specification implies a non-empty set R allowing one or more different implementations of the operation. If $R = \emptyset$, then there is obviously no implementation possible. We distinguish two cases: (1) no environment satisfying the precondition exists, or (2) there are environments making the precondition true, but no environments do satisfy the postcondition. Both cases indicate that the specification is inconsistent with the model. Either the constraint or the model providing the context should be changed. A more restrictive definition might even prohibit the second case.

5.3 Examples

Consider the operation raiseSalary from the example in Section 2. The operation raises the salary of an employee by a certain amount and returns the new salary.

```
context Employee::raiseSalary(amount : Real) : Real
  pre:  amount > 0
  post: result = self.salary
  post: self.salary = self.salary@pre + amount
```

The precondition only allows positive values for the amount parameter. The postcondition is specified as two parts which must both be true after executing the operation. This could equivalently be rephrased into a single expression combining both parts with a logical *and*. The first postcondition specifies that the result of the operation must be equal to the salary in the post-state. The second postcondition defines the new salary to be equal to the sum of the old salary and the amount parameter. All system states making the postcondition true, after a call to raiseSalary has completed, satisfy the operation specification.

The above example gives an exclusive specification of the operation's effect. The result is uniquely defined by the postconditions. Compare this with the next example giving a much looser specification of the result.

```
context Employee::raiseSalary(amount : Real) : Real
  pre:  amount > 0
  post: result > self.salary@pre
```

The result may be any value greater than the value of the salary in the previous state. Thus, the postcondition does not even prevent the salary from being decreased. However, what the example should make clear, is that there may not only exist many post-states but also many bindings of the result variable satisfying a postcondition. This is the reason why we have to consider both the system state *and* the set of variable bindings for determining the environment of an expression in a postcondition.

The following example shows the evaluation of the expression `a.b@pre.c`. An informal explanation was given in Section 5.1. With the previous syntax and semantics definitions, we are now able to give a precise meaning to this expression. Numbers in parentheses at the right of the transformations show which rule of Definition 4 (and Definition 2) has been applied in each step.

$$
\begin{aligned}
& I[\![\, c(b_{@\mathrm{pre}}(a)) \,]\!](\tau_{\mathrm{pre}}, \tau_{\mathrm{post}}) \\
& \quad = I(c)(\tau_{\mathrm{post}})(I[\![\, b_{@\mathrm{pre}}(a) \,]\!](\tau_{\mathrm{pre}}, \tau_{\mathrm{post}})) && \text{(iii)} \\
& \quad = I(c)(\tau_{\mathrm{post}})(I(b)(\tau_{\mathrm{pre}})(I[\![\, a \,]\!](\tau_{\mathrm{pre}}, \tau_{\mathrm{post}}))) && \text{(vii)} \\
& \quad = I(c)(\tau_{\mathrm{post}})(I(b)(\tau_{\mathrm{pre}})(\beta(a))) && \text{(i)} \\
& \quad = I(c)(\tau_{\mathrm{post}})(I(b)(\tau_{\mathrm{pre}})(\underline{a})) \\
& \quad = I(c)(\tau_{\mathrm{post}})(\underline{b}_1) \\
& \quad = 2
\end{aligned}
$$

6 OCL Tools

There are many CASE tools supporting drawing of UML diagrams and features like code generation and reverse engineering. However, support for OCL and semantic analysis of models is rarely found in these tools.

There are several tasks related to OCL for which tool support seems beneficial. For example, syntax checking of constraints helps in writing syntactically correct expressions. The next step could be an interpreter enabling the evaluation of expressions. Given a snapshot of a system, it could check the correctness of the snapshot with respect to the constraints. An alternative way for checking constraints is based on code generation. OCL expressions are transformed into statements of the implementation language. The generated code is responsible for detecting constraint violations.

A comprehensive list enumerating the most important kinds of tools supporting OCL is given in [14]. The authors distinguish between tools doing (1) syntactical analysis, (2) type checking, (3) logical consistency checking, (4) dynamic invariant validation, (5) dynamic pre-/postcondition validation, (6) test automation, and (7) code verification and synthesis. The following (incomplete) list gives an overview of some OCL tools.

- Probably the first available tool for OCL was a parser developed by the OCL authors at IBM (and now maintained at Klasse Objecten). The parser is automatically generated from the grammar given in [18].
- An OCL toolset is being developed at the TU Dresden [14]. Part of the toolset is an OCL compiler [10] that also has been integrated with the open source CASE tool Argo/UML [25].
- An OCL interpreter is described in [29]. It is partly based on an OCL metamodel describing the abstract syntax of OCL as a UML model [23].
- A commercial tool named ModelRun [3] provides validation of invariants against snapshots.
- The USE tool [21, 24] allows validation of OCL constraints by checking snapshots of a system. The tool also provides support for an analysis of constraints.

Table 2 compares the tools with respect to the features they support. The table only gives a rough indication about what is provided by a specific tool. However, what can clearly be seen is that logical consistency checking and code verification are features that currently none of the tools we considered here offers.

Table 2. Some OCL tools and the features they support.

Feature	Tool				
	IBM Parser	Dresden OCL Toolkit	TU Munich	ModelRun	USE
(1) syntactical analysis	•	•	•	•	•
(2) type checking	–	•	•	•	•
(3) logical consistency checking	–	–	–	–	–
(4) dynamic invariant validation	–	–	•	•	•
(5) dynamic pre-/post-condition validation	–	–	–	–	•
(6) test automation	–	–	–	–	•
(7) code verification and synthesis	–	–	–	–	–

6.1 The USE Tool

In this section, we present a tool for validating UML models and OCL constraints based on the formal syntax and semantics of OCL and UML models given earlier in this paper. The USE tool [21, 24] has an interpreter for OCL expressions and a facility for animating snapshots of a system. Different snapshots can be interactively generated and checked against the invariants specified as part of a model.

The goal of model validation is to achieve a good design before implementation starts. There are many different approaches to validation: simulation, rapid prototyping, etc. In this context, we consider validation by generating snapshots as prototypical instances of a model and comparing them against the specified model. This approach requires very little effort from developers since models can be directly used as input for validation. Moreover, snapshots provide immediate feedback and can be visualized using the standard notation of UML object diagrams – a notation most developers are familiar with.

The result of validating a model can lead to several consequences with respect to the design. First, if there are reasonable snapshots that do not fulfill the constraints, this may indicate that the constraints are too strong or the model is not adequate in general. Therefore, the design must be revisited, e.g., by relaxing the constraints to include these cases. On the other hand, constraints may be too weak, therefore allowing undesirable system states. In this case, the constraints must be changed to be more restrictive. Still, one has to be careful about the fact that a situation in which undesirable snapshots are detected during validation and desired snapshots pass all constraints does not allow a general statement about the correctness of a specification in a formal sense. It only says that the model is correct with respect to the analyzed system states. However, some advantages of validation in contrast to a formal verification are the possibility to validate non-formal requirements, and that it can easily be applied by average modelers without training in formal methods.

Validating pre- and postconditions. In the following, we will focus on support for validating pre- and postconditions. This new feature was not available in the version of the tool presented in [24] but has been added recently. The input for the USE tool is a textual specification of a UML model. As an example we use the model introduced in Figure 1. The corresponding USE specification is given in Appendix A. It describes the model found in the class diagram and the OCL constraints.

Figure 4 shows a screenshot of the USE tool visualizing various aspects of the example model. The left side of the window shows static information about the model whereas the right side contains several views each showing a different aspect of the dynamic system.

A user produces different system states by (1) adding or deleting objects, (2) inserting and removing links between them, (3) setting attribute values, and (4) simulating operation calls. The screenshot shows a system state after the commands listed in Appendix B have been executed. The view at the bottom right also partially shows the sequence of commands that led to the current state. These commands can be issued directly although most of them can more conveniently be triggered by intuitive interactions with the graphical user interface. For example, objects can be created by selecting a class and dragging it onto the object diagram.

The view at the top shows an object diagram with several objects, their attribute values and links between these objects. The small view labelled “Class Invariants” indicates that the two invariants we have defined in Section 2 are satisfied by the current snapshot.

The automatically generated sequence diagram shows the message flow between objects. In this example, the operation raiseSalary with a parameter value 200 has been called for the employee Frank who previously had a salary of 4500. An interactive command window (not shown in the figure) reports the success of both the pre- and the postconditions. In case of a failing precondition the operation could not have been entered at all. A failing postcondition would be visualized with a red return arrow in the sequence diagram. In both cases, a detailed report on the evaluation of expressions gives hints to the user why the conditions failed.

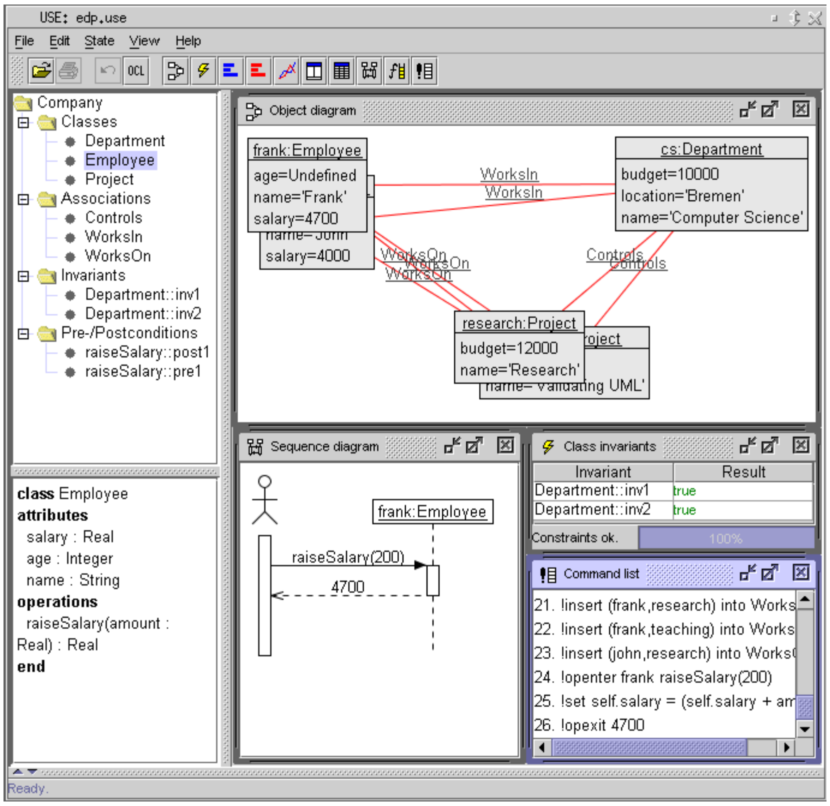


Fig. 4. Screenshot of USE tool

The commands simulating an operation call are the following.

```
-- call operation raiseSalary
!openter frank raiseSalary(200)
!set self.salary = self.salary + amount
!opexit 4700
```

An operation call is simulated by first issuing an `openter` command with a source expression, the name of the operation, and an argument list. The `openter` command has the following effect.

1. The source expression is evaluated to determine the receiver object.
2. The argument expressions are evaluated.
3. The OCL variable `self` is bound to the receiver object and the argument values are bound to the formal parameters of the operation. These bindings determine the local scope of the operation.
4. All preconditions specified for the operation are evaluated.
5. If all preconditions are satisfied, the current system state and the operation call is saved on a call stack. Otherwise, the operation call is rejected.

The side effects of an operation are specified with the usual USE commands for changing a system state. In the example, the `set` command assigns a new value to the salary attribute of the employee. After generating all side effects of an operation, the operation can be exited and its postconditions can be checked. The command `opexit` simulates a return from the currently active operation. The result expression given with this command is only required for operations that specify a result value. The `opexit` command has the following effect.

1. The currently active operation is popped from the call stack.
2. If an optional result value is given, it is bound to the special OCL variable `result`.
3. All postconditions specified for the operation are evaluated in context of the current system state and the pre-state saved at operation entry time.
4. All variable bindings local to the operation are removed.

In our example, the postcondition is satisfied and the operation has been removed from the call stack. We give another example that shows how operation calls may be nested in the simulation. It also shows that postconditions may be specified on operations without side effects. An OCL expression is given to describe the computation of a side effect free operation. In the example, we use a recursive definition of the factorial function.

```
model NestedOperationCalls

class Rec
operations
  fac(n : Integer) : Integer =
    if n <= 1 then 1 else n * self.fac(n - 1) endif
end

constraints

context Rec::fac(n : Integer) : Integer
  pre:  n > 0
  post: result = n * fac(n - 1)
```

The postcondition of the operation `Rec::fac` reflects the inductive case of the definition of the factorial function. The following commands show the computation of 3!.

```
!create r : Rec
!openter r fac(3)
!openter r fac(2)
!openter r fac(1)
!opexit 1
!opexit 2
!opexit 6
```

The operation calls are exited in reverse order and provide result values that satisfy the postcondition. Figure 5 shows the sequence diagram generated from this call sequence. The stacked activation frames in the diagram emphasize the recursion.

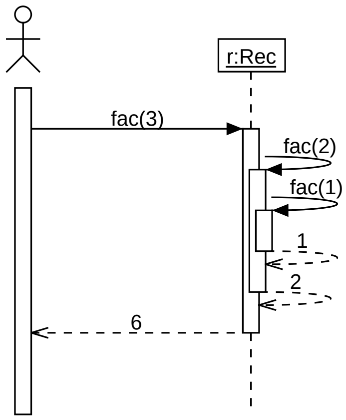


Fig. 5. Sequence diagram for recursive operation call

7 Conclusion

OCL is an important part of UML. Many constraints that cannot be expressed in the UML diagram notation can be elegantly stated with OCL expressions. We argued that a formal language like OCL also should have a formal semantics and presented our approach to developing a precise semantics of OCL. This semantics covers expressions, invariants, contexts, and pre- and postconditions. The concepts and results presented here are implemented in an UML/OCL CASE tool for rapid prototyping and validation of UML designs including OCL expressions.

References

1. T. Baar and R. Hähnle. An integrated metamodel for OCL types. In R. France, B. Rumpe, J.-M. Bruel, A. Moreira, J. Whittle, and I. Ober, editors, *Proc. OOPSLA 2000, Workshop Refactoring the UML: In Search of the Core, Minneapolis, Minnesota, USA, 2000.*, 2000.
2. T. Baar, R. Hähnle, T. Sattler, and P. H. Schmitt. Entwurfsmustergesteuerte Erzeugung von OCL-Constraints. In K. Mehlhorn and G. Snelting, editors, *Informatik 2000, 30. Jahrestagung der Gesellschaft für Informatik*, pages 389–404, Sept. 2000.
3. BoldSoft. Modelrun, 2000. Internet: `http://www.boldsoft.com/products/modelrun/index.html`.

4. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
5. P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency checking and visualization of OCL constraints. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 294–308. Springer, 2000.
6. T. Clark. Type checking UML static diagrams. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 503–517. Springer, 1999.
7. S. Cook, A. Kleppe, R. Mitchell, B. Rumpe, J. Warmer, and A. Wills. The Amsterdam manifesto on OCL. Technical Report TUM-I9925, Technische Universität München, Dec. 1999.
8. S. Cook, A. Kleppe, R. Mitchell, J. Warmer, and A. Wills. Defining the context of OCL expressions. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 372–383. Springer, 1999.
9. D. Distefano, J.-P. Katoen, and A. Rensink. On a temporal logic for object-based systems. In S. F. Smith and C. L. Talcott, editors, *Formal Methods for Open Object-Based Distributed Systems IV - Proc. FMOODS'2000, September, 2000, Stanford, California, USA*. Kluwer Academic Publishers, 2000.
10. F. Finger. Design and implementation of a modular OCL compiler. Diplomarbeit, Dresden University of Technology, Department of Computer Science, Software Engineering Group, Germany, Mar. 2000.
11. M. Gogolla and M. Richters. On constraints and queries in UML. In M. Schader and A. Korthaus, editors, *The Unified Modeling Language – Technical Aspects and Applications*, pages 109–121. Physica-Verlag, Heidelberg, 1998.
12. A. Hamie, F. Civello, J. Howse, S. Kent, and R. Mitchell. Reflections on the Object Constraint Language. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998*, pages 137–145, 1998.
13. C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Communications of the ACM*, 12:576–580, 1969.
14. H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 278–293. Springer, 2000.
15. 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. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 440–450. Springer, 2000.
16. L. Mandel and M. V. Cengarle. On the expressive power of OCL. In *FM'99 - Formal Methods. World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 1999. Proceedings, Volume I*, volume 1708 of *LNCS*, pages 854–874. Springer, 1999.
17. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

18. OMG. Object Constraint Language Specification. In *OMG Unified Modeling Language Specification, Version 1.3, June 1999* [19], chapter 7.
19. OMG, editor. *OMG Unified Modeling Language Specification, Version 1.3, June 1999*. Object Management Group, Inc., Framingham, Mass., Internet: `http://www.omg.org`, 1999.
20. S. Ramakrishnan and J. McGregor. Extending OCL to support temporal operators. In *Proceedings of the 21st International Conference on Software Engineering (ICSE99) Workshop on Testing Distributed Component-Based Systems, LA, May 16 - 22, 1999*, 1999.
21. M. Richters. The USE tool: A UML-based specification environment, 2001. `http://www.db.informatik.uni-bremen.de/projects/USE/`.
22. M. Richters and M. Gogolla. On formalizing the UML Object Constraint Language OCL. In T. W. Ling, S. Ram, and M. L. Lee, editors, *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, volume 1507 of *LNCS*, pages 449–464. Springer, 1998.
23. M. Richters and M. Gogolla. A metamodel for OCL. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 156–171. Springer, 1999.
24. M. Richters and M. Gogolla. Validating UML models and OCL constraints. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 265–277. Springer, 2000.
25. J. Robbins et al. Argo/UML CASE tool, 2001. `http://www.argouml.org`.
26. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
27. S. Sendall and A. Strohmeier. From use cases to system operation specifications. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 1–15. Springer, 2000.
28. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.
29. M. Wittmann. Ein Interpreter für OCL. Master's thesis, Ludwig-Maximilians-Universität München, 2000.

Appendix

A USE Specification of the example model

```
-- model

model Company

class Employee
attributes
  name : String
  age : Integer
  salary : Real
operations
```

```
  raiseSalary(amount : Real) : Real
end

class Department
attributes
  name : String
  location : String
  budget : Integer
end

class Project
attributes
  name : String
  budget : Integer
end

association WorksIn between
  Employee[*]
  Department[1..*]
end

association WorksOn between
  Employee[*]
  Project[*]
end

association Controls between
  Department[1]
  Project[*]
end

-- OCL constraints

constraints

context Department inv:
  self.budget >= 0

-- Employees working on more projects than other
-- employees of the same department get a higher salary.
context Department inv:
  self.employee->forAll(e1, e2 |
    e1.project->size > e2.project->size
      implies e1.salary > e2.salary)

-- If the amount is positive, raise
-- the salary by the given amount
context Employee::raiseSalary(amount : Real) : Real
  pre:  amount > 0
  post: self.salary = self.salary@pre + amount
```

```
    and result = self.salary
```

B Commands for Animating a Model

```
-- create department
!create cs:Department
!set cs.name = 'Computer Science'
!set cs.location = 'Bremen'
!set cs.budget = 10000

-- create employee john
!create john : Employee
!set john.name = 'John'
!set john.salary = 4000

-- create employee frank
!create frank : Employee
!set frank.name = 'Frank'
!set frank.salary = 4500

-- establish WorksIn links
!insert (john,cs) into WorksIn
!insert (frank,cs) into WorksIn

-- create project research
!create research : Project
!set research.name = 'Research'
!set research.budget = 12000

-- create project teaching
!create teaching : Project
!set teaching.name = 'Validating UML'
!set teaching.budget = 3000

-- establish Controls links
!insert (cs,research) into Controls
!insert (cs,teaching) into Controls

-- establish WorksOn links
!insert (frank,research) into WorksOn
!insert (frank,teaching) into WorksOn
!insert (john,research) into WorksOn

-- call operation raiseSalary
!openter frank raiseSalary(200)
!set self.salary = self.salary + amount
!opexit 4700
```