An OCL Extension for Low-coupling Preserving Contracts

I. Nunes

Lisbon University, Bloco C5, Piso 1, Campo Grande,
1749-016 Lisboa, Portugal
in@di.fc.ul.pt

Abstract. Design by contract, as introduced by B.Meyer, is of increasing importance to the OO community in the specification, reuse, and monitoring of classes. We strongly feel that class libraries of all programming languages should be equipped with contracts, insofar as these constitute a powerful and simple interface definition. Very powerful and expressive contracts can be written using the OCL language, although for operations with many effects on the system state, these contracts can become unmanageable and incomprehensible. In order to maintain contracts at a manageable level of complexity, we claim that the OCL powerful mechanism of navigation through associations should be used moderately when building contracts, and that the effects of non-query operations should be allowed to be referred to within pre- and post-conditions. To achieve that purpose, we propose an extension to OCL and present a formal semantics for it.

1 Introduction

The definition of object behaviour asks for an expressiveness that the UML graphical notation, although very rich and powerful, can not provide. Complex constraints are sometimes necessary to specify the operations that define the behavioural interface of objects.

The Object Constraint Language (OCL) [8,15] is an integral part of the UML standard and, being a textual constraint language, it is more appropriate for that task. Constraints on the behaviour of objects are specified in OCL by means of pre- and post-conditions which constitute the contract that client and supplier objects have to fulfill [7]. Invariants, which are constraints that concern the static structure, insofar as they must be true for all instances of the type at any time, are also complex constraints that are easily defined with OCL. We call class contracts the assertions that constitute both its invariant and the pre and post-condition pairs for all its operations. Contracts have an important role in several contexts and uses.

Contracts are important in software implementation and reuse. Implementors of some class A wish for contracts that help them in building implementations of A,

that is, they wish for assertions that: i) leave no doubts about the properties of the resulting states of operations of A (post-conditions and invariants); ii) assert what are the assumptions that can be made about the initial states of operations (pre-conditions and invariants); iii) can be proved against implementations, that is, allow for the verification of the correctness of implementations.

Designers and implementors of classes that are clients of a class A wish for contracts in A that help them in their own code writing, that is, they wish assertions in A that: i) are easy to understand; ii) clearly specify the obligations (pre-conditions and invariants) that they should accomplish in order to *honour the operation contracts with A*; iii) clearly specify the benefits (post-conditions and invariants) that they will have if they *sign the operation contract with A*, that is, if they invoke the operation of A in a state that satisfies its pre-condition.

Dynamic contract validation is also very useful in software testing and debugging. In the same way as several assertion languages have associated code generation tools that generate validation code in some specific programming language, which allows the validation of contracts at run-time [1,2,3,4,6,7], also the USE tool [12] allows validation of OCL constraints against UML models by checking snapshots of a system.

1.1 The Problem

In contracts for operations with many direct and indirect effects on the system state, contracts can become unmanageable and incomprehensible. We should try to maintain contracts at a manageable level of complexity.

Moreover, contracts should bring no more class coupling than the one that is necessary to understand and further implement a given operation. Low coupling is a concern that should be present in all modeling tasks. What is the use of having a system of classes built in a way where this was taken into account, if the contracts that are associated with classes suffer from high coupling?

Keeping contracts simple and manageable, and preserving low coupling asks for abstraction mechanisms that the OCL language (as well as other assertion languages) lacks.

1.2 Our Proposal

OCL constraints should, then, be written taking all these concerns into account. Although the OCL gives us a very powerful mechanism of navigation through associations, nevertheless we claim that this capacity should be used moderately when building contracts.

Navigation through more than two association levels wide, brings serious difficulties in understanding and managing contracts. It also dramatically increases class coupling.

The way we propose dealing with this negative effect leads us to another claim: that the effects of non-query operations should be allowed to be referred to in pre and post-conditions. The exact meaning of this claim will be explained below.

1.3 Outline of the Paper

This paper has 5 sections. In section 2 we describe the approach that is commonly used when defining OCL pre and post-conditions, and we argue why this is not a satisfactory approach. In section 3 we propose a different approach to building operation contracts and we claim the need for non-query operation effects to be allowed in pre and post-conditions. In section 4 we propose a new kind of OCL pre and post-conditions and define its semantics using the approach of [14,16]. Finally, in section 5 we present the conclusions.

2 OCL Pre and Post-conditions

In this section we use the example presented in [9], where the concept of meta-assertion was introduced as a means to allow the specification of contracts for complex classes, without the drawback of increasing coupling. We will show how the expressive power of OCL in terms of navigability can be used to specify powerful pre and post conditions, and we will argue that contracts obtained in this way can become much too complex to be appealing to use.

This example deals with points, polygons (whose vertices are points) and drawings (which are composed of polygons). Each one of these types defines an operation of movement by given distances both horizontally (dh) and vertically (dv).

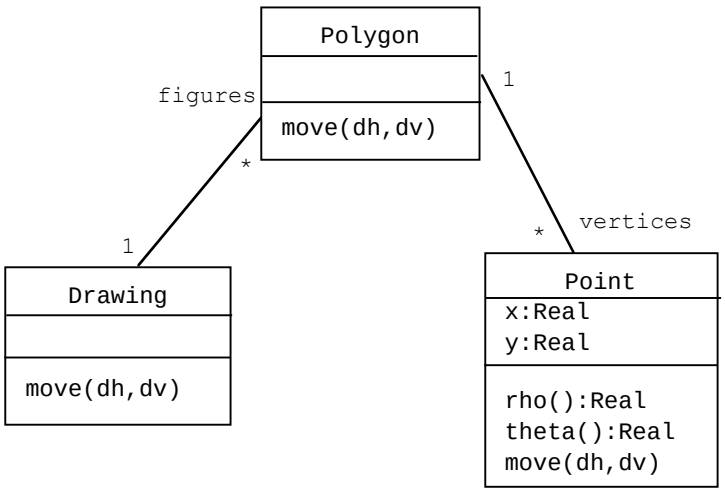

Fig. 1.

Making use of the navigability power of OCL, the move operation in class Drawing can be specified in the following way:

```
context Drawing :: move(dh:Real, dv:Real)
    pre:
   post: figures -> forall (f|
              f.vertices -> forall (v|
                  v.x=v.x@pre+dh and v.y=v.y@pre+dv))
```

Apart from being somewhat complicated, if we adopt this kind of contracts for implementors to use them we will be somehow ignoring the careful design that Figure 1 shows.

This post-condition increases the coupling between the classes of the system: clients of class Drawing must know about class Point in order to understand the result of applying the move method to a drawing composed of polygons! This should not be necessary; after all, a set of polygons abstracts away the structured set of points that constitute the drawing. As we know, strong coupling brings undesirable designs due to the decreasing in extension and reuse. The coupling that is shown in Figure 1 should not be increased. There, class Point is a stranger to class Drawing. Here, in the post-condition, class Drawing must know about the vertices of polygons.

We should be able to act over a drawing of polygons solely through the polygons themselves. The ideal way to do this would be something like:

```
context Drawing :: move(dh:Real, dv:Real)
    pre:
    post: figures -> forall (f|f.somethingAbout_f_only)
```

that would reveal the changes operated in the drawing only through their most direct state revealing queries.

The concern shown in design pattern "Don't talk to strangers" [5], should also apply to design by contract: restrictions must be placed on what objects can be spoken about in contracts: i) the current object; ii) a parameter of the method; iii) an attribute of the current object; iv) an element of a collection which is an attribute of the current object; v) an object created within the method.

The intent here is also to avoid coupling a client to knowledge of indirect objects and the internal representations of direct objects. Direct objects are a client's *familiars*, indirect objects are *strangers* and a client should only talk about familiars, not about strangers.

3 The Need for Non-query Operations in OCL Pre and Post-conditions

We claim an approach [9] where contracts refer to the contracts of other methods allowing to say, for example, that the *result* of moving a drawing is the same as the *result* of moving all its polygons. In this case we would have:

```
context Drawing :: move(dh:Real, dv:Real)
    pre:
    post: figures -> forall (f|f.move(dh,dv))

context Polygon :: move(dh:Real, dv:Real)
    pre:
    post: vertices -> forall (v|v.move(dh,dv))

context Point :: move(dh:Real, dv:Real)
    pre:
    post: x = x@pre + dh  and  y = y@pre + dv
```

The intended meaning of the first contract, for example, is: the state that results from the execution of the command `move` applied to an object of type `Drawing`, is the same as the state that results from applying the `move` operation to all its polygons. In other words, the post-conditions of all commands `move` applied to all the drawing polygons are true in the resulting state.

The idea here is that, whenever a non-query operation appears in a post-condition, its semantics is the same as the one of its own post-condition. In the same way, whenever a non-query operation `op` appears in a pre-condition, its semantics is the one of `op`'s pre-condition. In this way, we are able to represent the result of an operation by writing only the conditions that are of the direct responsibility of the enclosing class. We do this without creating unnecessary query methods for querying objects that are "strangers" to client classes.

The benefits of this approach are complexity decreasing and all the ones that low coupling brings. Suppose the post-condition of `Point`'s move operation changes. Because none of `Drawing` and `Polygon` classes refer explicitly, in their own move operation post-conditions, to the effects of moving a vertex, they are safe from changes. Not even `Polygon`'s `move` operation must have its post-condition changed, because it only refers to the *effects* of moving its vertices in an implicit way. If we had chosen the first approach - the one that makes full use of OCL navigation power - we would have to change both `Drawing` and `Polygon` `move` post-conditions. Remember that pre and post- conditions are also used to generate monitoring code.

Even though we have to understand the post-conditions of some non-query operations in order to understand the post-condition that refers to them, we think that this is preferable to having to understand a huge, complicated assertion - divide to conquer.

We have chosen to allow non-query operations in pre and post-conditions, with the above described informal semantics (see section 4 for the formal semantics), because we followed the general idea in OCL that "the semantics of an object operation is given by the semantics of the associated OCL expression" [16]. Adding to this, we consider that the context (here understood as pre or post-condition definition) in which the non-query operation appears dictates its meaning - whether it denotes the pre or the

post-condition of the operation it represents. In [9,10] we introduced, instead, a special syntactic construct to denote meta-assertions - assertions that refer to other assertions. The benefit this latter approach brings is the capacity of referring to a pre-condition in the context of a post-condition, that is, the ability to say that the execution of a given method establishes the pre-condition of another. The other way around, that is, saying that the post-condition of a given method must be true in order to another method to be executed does not make sense, because post-conditions allow us to talk about previous values of elements which, in a pre-condition, are not known.

The following example illustrates another drawback [10] in using the first approach of section 2, and the way our approach deals with it. We have the hierarchy of bank account classes shown in Figure 2.

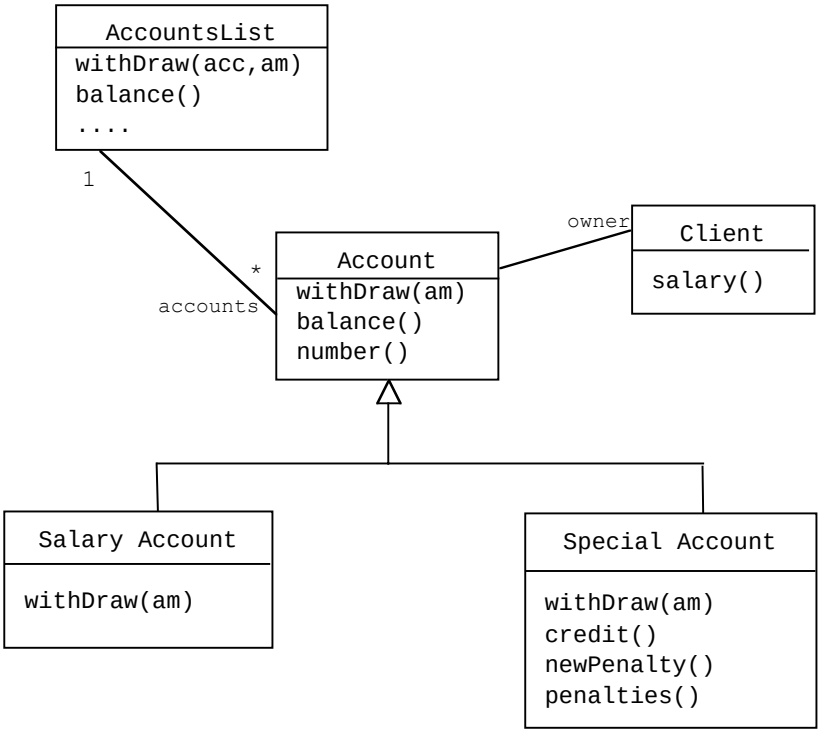

Fig. 2.

The contracts for operation `withDraw` for these classes could be the following:

```
Context Account::withDraw(amount:Real)

pre: amount  ≤ balance()
post: balance() = balance@pre() – amount
```

```
Context SalaryAccount::withDraw(amount:Real)

pre: amount  ≤ balance() + owner.salary()
post: balance() = balance@pre() – amount


Context SpecialAccount::withDraw(amount:Real)

pre: amount  ≤ balance() + credit()
post: balance() = balance@pre() – amount and
          amount = balance@pre()+credit@pre() implies
             penalties() = penalties@pre()+1
```

Furthermore, the contract for operation `newPenalty` in class `SpecialAccount` is the following:

```
Context SpecialAccount::newPenalty()

pre:
post: penalties() = penalties@pre()+1
```

The `withDraw(acc:Integer,amount:Real)` operation of class `AccountsList` can only succeed if its accounts set contains an account whose number is `acc`. Furthermore, that account must be such that it allows the withdrawal of the `amount` in question. Here we are faced with the following difficult decision: what does it mean for an account to allow the withdrawal of a certain amount? Well, that depends on the kind of account it is. We only need to look at the pre-conditions of the three existing kinds of accounts to see that they are all specified by different constraints. If we use the operations and associations that are described in Figure 2, and because OCL does not allow the use of non-query operations, we have to choose one of them. Due to their hierarchical relationship, we are constrained to choose the one of the parent class `Account`:

```
Context AccountsList::
                    withDraw(acc:Integer,amount:Real)

pre: accounts->exists(number()=acc) and
         accounts->select(number()=acc)->
                             forall(amount ≤ balance())
```

But then we would obtain a constraint that is too strong in some cases. If this pre-condition is used to somehow prevent the execution of the operation, then we should be more careful specifying it. An object of type `AccountsList` contains `Account` objects as well as `SalaryAccount` and `SpecialAccount` ones. Suppose the operation `withdraw(acc,amount)` is invoked in a state where the account whose number is `acc` is a `SpecialAccount`, and where the value of `amount` is greater than the account's `balance()` but smaller than the account's `balance()` plus the account's `credit()`. If the pre-condition that is tested is the `Account`'s `withdraw` one (as happens above) that

call will be rejected. This should not happen: in what the SpecialAccount object is concerned, the withdrawal of amount is perfectly legal.

Choosing SpecialAccount's withDraw pre-condition or SalaryAccount's one would not solve the problem either. We would be in danger of accepting withdrawals for Account objects that we should not accept.

The solution must take into account the type of the object over which the withDraw operation is to be invoked. In the same way as operations in OCL expressions represent the operations that conform to the type of the object over which they are invoked, the pre-condition that should be tested in this case should also conform to the type of the object in question. Our solution is to use the non-query operation withDraw(amount:Real) in the pre-condition of the AccountsList withDraw(acc:Integer,amount:Real) operation. Its semantics would be the same as the one of its pre-condition (not the execution of that operation):

```
Context AccountsList::
                    withDraw(acc:Integer,amount:Real)
pre: accounts->exists(number()=acc) and
        accounts->select(number()=acc)->
                          forall(withDraw(amount))
```

The intended meaning of this pre-condition is that an account whose number is acc must exist and the pre-condition of its withDraw operation must hold.

The post-condition for this operation must state that the balance of the account whose number is acc is subtracted by amount. This is what must happen whatever the kind of account that account is.

```
post: accounts->select(number()=acc)->
           forall(balance() = balance@pre() – amount)
```

However, if we look at the post-conditions of operation withdraw in classes SalaryAccount and SpecialAccount we see that we are missing something.

Suppose we use this post-condition to monitor our system. Suppose, further, that the withdraw(acc:Real) method in class SpecialAccount is ill-implemented, that is, it does not establish its full post-condition. If the post-condition that is monitored is the one above, it becomes impossible to trap the cases which violate the SpecialAccount's withdraw() post-condition.

Here again the solution must take into account the type of the object over which the withDraw operation is to be invoked. Our solution is to use the non-query operation withDraw(amount:Real) in the post-condition of the AccountsList withDraw(acc:Integer,amount:Real) operation. Its semantics would be the same as the one of its post-condition (not the execution of that operation):

```
post: accounts->select(number()=acc)->
                          forall(withDraw(amount))
```

The intended meaning of this post-condition is that the post-condition of the `withDraw` operation of the account whose number is `acc`, must hold.

The approach we propose would finally lead to the following contract specifications:

```
Context AccountsList::
                        withDraw(acc:Integer,amount:Real)
pre: accounts->exists(number()=acc) and
          accounts->select(number()=acc)->
                                forall(withDraw(amount))
post: accounts->select(number()=acc)->
                                forall(withDraw(amount))
```

The post-condition of the `withDraw` operation of `SpecialAccount` class should also be different from the one presented before. The

```
amount = balance@pre()+credit@pre() implies
                        penalties() = penalties@pre()+1
```

part of that post-condition, describes the post-condition of the `newPenalty` operation. With the proposed approach, we would get the following:

```
Context SpecialAccount::withDraw(amount:Real)
pre: amount  ≤ balance() + credit()
post: balance() = balance@pre() – amount and
            amount = balance@pre()+credit@pre() implies
                  newPenalty()
```

The `newPenalty()` operation represents the effects of executing that operation, not the execution itself.

One of the anonymous referees pointed out two approaches that could be used to solve the problems we claim to solve. We discuss here the drawbacks of those approaches.

One of the pointed approaches uses the OCL message concept, introduced in [11], that allows to express messages sent by component classes or other constructs, through the construct `^`. The semantics of the message expression `obj^mess(args)` says that the OclMessageValue it represents (either an operation call or a UML signal) has been in the output queue of the sending instance at some point between "now" (if `obj^mess(args)` appears in a post-condition, "now" is the time at which the post-condition is evaluated) and a reference point in time (if `obj^mess(args)` appears in a post-condition, this reference is the time at which the pre-condition is evaluated). Thus, we could express in the post-condition of the `Drawing move` method, for example, that explicit calls were made to the `move` methods of all its polygons. This is very restrictive in our opinion, insofar as it reveals the implementation, and cannot be seen as specification any more. Moreover, the fact that some method `mess(args)` has been called during the execution of

some operation `op`, does not ensure that its effects are effective when `op` finishes execution... some other operation may happen, that is not specified in the post-condition, that modifies some of `mess(args)` results. The only thing we can be sure about when we see the construct `obj^mess(args)` in an operation `op's` post-condition is that, sometime during the execution of `op`, the method `mess(args)` has been called over the instance `obj`.

The approach we propose in this paper deals with state descriptions; it does not necessarily pre-selects any specific implementation like the OCL message expression `^` does. We say that the *result* of moving a drawing is the same as the *result* of moving all of its polygons, whatever the implementation that is chosen to accomplish it.

The second pointed out approach uses the possibility to define pre and post-conditions and invariants as separate OCL attributes/operations that can be referred to at various places. But this is far from being straightforward. The pre and post-conditions are instances of a special kind of ModelElement: the Constraint. This kind of elements have, in addition to the attributes and associations any ModelElement has, an attribute *body* which is a BooleanExpression and zero or more associated ModelElements - the *constrainedElements*. Every BehaviouralFeature (Operation or Method) has zero or more associated constraints, each with a stereotype - <<precondition>> and <<postcondition>> are enough in what concerns this paper. Let us go back to the pre-condition of the `withDraw` method of `AccountsList` class as proposed by our approach:

```
Context AccountsList::
                    withDraw(acc:Integer,v:Real)

pre: accounts->exists(number()=acc) and
        accounts->select(number()=acc)->
                          forall(withDraw(v))
```

Remember that the `withDraw(v)` that appears in the pre-condition is the evaluation of the pre-condition of the `withDraw` method that is defined in the class of the `accounts` element whose number is `acc`. To the best of our knowledge, there are two ways to (incompletely) accomplish the desired effect using Constraint elements: instead of `withDraw(v)` we would have:

```
classifier.lookupOperation("withDraw",S).constraint->
   select(c|c.stereotype.name="precondition").body
```

where S is a sequence containing the parameter types of the `withDraw` operation; `lookupOperation` is defined in section 3.3.8 of [11] and returns the Operation whose name and signature match the values given as arguments. The other way would be similar but, instead of `c.stereotype.name="precondition"` we would have `c.name="preName"` where `preName` is the name we would have given to the pre-condition of the `withDraw` method in all of the `Account` classes.

Apart from being quite complicated, this does not do the job. The body of a Constraint is a BooleanExpression. Let us suppose that the account which number is `acc`

is an ordinary `Account`. The pre-condition of the `Account` class `withDraw` method refers to the formal parameter `amount`. But we want to evaluate it for an amount of `v`. Unless we have a way to substitute the arguments for the names of its *constrainedoperation* formal parameters, the presence of this Constraint body in the pre-condition of the `withDraw` method of `AccountsList` is not right.

4 Extending OCL

In this section we propose an extension of OCL pre and post-conditions and present its semantics by extending the OCL semantics presented in [14,16].

In [14,16], all common OCL expressions could be used in pre-conditions. No extension was given for this type of constraints. Post-conditions, in turn, were defined as an extension to OCL expressions due to the construct `@pre`, the special variable `result`, and the operation `oclIsNew`. Here we must extend the definition that was given for the syntax of OCL expressions both for pre and post-conditions (which we will call assertions from here on).

We extend the notion of object model given in [14,16].

Definition 1 (SYNTAX OF OBJECT MODELS)

The syntax of an object model is a structure

$$M= (\text{CLASS}, \text{ATT}_c, \text{OP}_c, \text{ASSERT}_{\text{OPc}}, \text{ASSOC}, \text{associates}, \text{roles}, \text{multiplicities}, <)$$

where CLASS, ATT_c, OP_c, ASSOC, associates, roles, multiplicities, and < are as defined in [14,16]; $\text{ASSERT}_{\text{OPc}}$ is an OP_c-indexed set of assertion pairs (that is, a finite set of name pairs). ♦

We also extend the notion of signature in order to define the universe of assertions. Moreover, the set of operations is no longer limited to side effect-free operations.

Definition 2 (DATA SIGNATURE)

The *syntax* of a data signature over an object model M is a structure $\Sigma_M = (T_M, \leq, \Omega_M, A_M)$ that provides a set of types T_M, a relation $\leq$ on types reflecting the type hierarchy, a set of operations Ω_M, and a set of pairs of assertions A_M. The set of operations Ω_M, may contain non-query operations.

The *semantics* of a data signature is a structure $I(\Sigma_M) = (I(T_M), I(\leq), I(\Omega_M), I(A_M))$ where $I(T_M)$, $I(\leq)$, and $I(\Omega_M)$, are as defined in [14,16], and $I(A_M)$ assigns each assertion pair $a \in A_M$ a total function $I(a): \rightarrow \text{Pre-Expr}_{\text{Boolean}} \times \text{Post-Expr}_{\text{Boolean}}$. The sets $\text{Pre-Expr}_{\text{Boolean}}$ and $\text{Post-Expr}_{\text{Boolean}}$ are the OCL pre-condition boolean expression set and the OCL post-condition boolean expression set, respectively (as defined in definitions 3 and 5 below). ♦

The syntax of general OCL expressions must be redefined in order to forbid non-query operations in general expressions. These non-query operations must only be allowed to appear in pre and post-conditions. Because we allowed non-query operations in the data signature, we must forbid them in general OCL expressions.

In what respects special OCL expressions - pre and post-conditions - we must define the syntax and semantics of pre-conditions and extend the ones of post-conditions.

Definition 3 (SYNTAX OF PRE-CONDITION EXPRESSIONS)

Let $\Sigma_M = (T_M, \leq, \Omega_M, A_M)$ be a data signature over an object model M. Let Var = $\{Var_t\}_{t \in T_M}$ be a T_M-indexed family of variable sets. The basic of expressions in pre-conditions is defined by repeating the definition given in [14,16] while substituting all occurrences of $Expr_t$ with Pre-$Expr_t$. Furthermore we define that if w: $t_1 \times ... \times t_n \rightarrow t \in \Omega_M$, and $e_i \in$ Pre-$Expr_{ti}$, and w is a non-query operation, then $w(e_1, ...,e_n) \in$ Pre-$Expr_{Boolean}$. ♦

The semantics of OCL expressions is defined over an *environment* which gives the context for evaluation. It consists of a system state σ and a variable assignment β: $Var_t \rightarrow I(t)$. A system state provides access to the set of currently existing objects, their attribute values, and association links between objects. A variable assignment maps variable names to values.

Definition 4 (SEMANTICS OF PRE-CONDITION EXPRESSIONS)

Let Env be the set of environments $\tau=(\sigma,\beta)$. The semantics of an expression $e \in$ Pre-$Expr_t$ is a function I[e]: Env $\rightarrow$ I(t). The semantics of the basic set of expressions in pre-conditions is defined by repeating the definition given in [14,16] while substituting all occurrences of $Expr_t$ with Pre-$Expr_t$.

Furthermore we define that

$$I[w(e_0, p_1{:}e_1,..., p_n{:}e_n)](\sigma,\beta) = I[first(I(a))] (\sigma',\beta').$$

where w is a non-query operation, e_0 is an expression that denotes the object that is the target of w, and $p_1,..., p_n$ are parameter names. The element *a* is the pair of assertions associated with operation w, that is, *a* is $ASSERT_w$. So, I(a) is the interpretation of *a* (as given in definition 2), that is, the pair composed of an OCL pre-condition boolean expression and an OCL post-condition boolean expression that represent w's pre-condition and post-condition, respectively. The assertion that is of interest here is the pre-condition, which is the first element of that pair. Remember that (σ,β) is the environment of some operation *op* whose pre-condition contains expression $w(e_0, p_1{:}e_1,..., p_n{:}e_n)$. The pair (σ',β') denotes the environment:

$$(\sigma\{self/I[e_0](\sigma,\beta)\} , \beta\{p_1/I[e_1](\sigma,\beta), \ldots, p_n/I[e_n](\sigma,\beta)\}) .$$

Argument expressions are evaluated and assigned to parameters that bind free occurrences of $p_1,..., p_n$ in the expression first(I(a)), that is, in w's pre-condition. Expres-

sion e_0 is evaluated and assigned to *self* . This means that w's pre-condition is evaluated in a state where the current object is the one denoted by expression e_0. ♦

Now we extend [14,16] syntax definition for post-conditions and give the semantics of the new syntactic elements.

Definition 5 (SYNTAX OF POST-CONDITION EXPRESSIONS)

Let $\Sigma_M = (T_M, \leq, \Omega_M, A_M)$ be a data signature over an object model M. Let Var = $\{Var_t\}_{t \in T_M}$ be a T_M-indexed family of variable sets. The basic of expressions in post-conditions is defined by repeating the definition for post-conditions given in [14,16] (the one that extended OCL expressions with *@pre*, *result* and *oclIsNew*). Furthermore we define that if w: $t_1 \times ... \times t_n \rightarrow t \in \Omega_M$, and $e_i \in$ Post-$Expr_{ti}$, and w is a non-query operation, then $w(e_1, ...,e_n) \in$ Post-$Expr_{Boolean}$. ♦

We should emphasize here that the type of `obj.mess(args)` where `mess(args)` is a non-query operation, is Post-$Expr_{Boolean}$. This means that we cannot have `obj.mess(args).other()` where the class where `other()` is defined is not Boolean.

Now we need not one, but two environments in order to evaluate post-conditions: the environment before the execution and the one that results from the execution. The system state and variable assignments before the execution of an operation constitute the *pre-environment* $\tau_{pre}=(\sigma_{pre},\beta_{pre})$. Likewise, the system state and variable assignments after the execution of an operation constitute the *post-environment* $\tau_{post}=(\sigma_{post},\beta_{post})$.

Definition 6 (SEMANTICS OF POST-CONDITION EXPRESSIONS)

Let Env be the set of environments $\tau=(\sigma,\beta)$. The semantics of an expression $e \in$ Post-$Expr_t$ is a function $I[e]:Env \times Env \rightarrow I(t)$. The semantics of the basic set of expressions in post-conditions is defined by repeating the definition given in [14,16] (the one that extended OCL expressions with *@pre*, *result* and *oclIsNew*). Furthermore we define that

$$I[w(e_0, p_1:e_1,..., p_n:e_n)](\sigma_{pre},\beta_{pre})(\sigma_{post},\beta_{post}) = I[second(I(a))](\sigma'_{pre},\beta'_{pre})(\sigma'_{post},\beta'_{post}) .$$

where w is a non-query operation, e_0 is an expression that denotes the object that is the target of operation w call, and $p_1,..., p_n$ are parameter names. The element *a* is the pair of assertions associated with operation w, that is, *a* is $ASSERT_W$. So, I(a) is the interpretation of *a* (as given in definition 2), that is, the pair composed of an OCL pre-condition boolean expression and an OCL post-condition boolean expression that represent w's pre-condition and post-condition, respectively. The assertion that is of interest here is the post-condition, which is the second element of that pair. Remember that $(\sigma_{pre},\beta_{pre})$ and $(\sigma_{post},\beta_{post})$ are the pre and post-environments of some operation whose post-condition contains expression $w(e_0, p_1:e_1,..., p_n:e_n)$. Let us call this operation *op*. The state σ'_{pre} denotes:

$$\sigma_{pre}\{self/I[e_0]\ (\sigma_{pre},\beta_{pre})\ (\sigma_{pre},\beta_{pre})\}\ .$$

This means that the pre-state in which w's post-condition is going to be evaluated is such that the object *self* is the same as operation w target object before being modified by operation *op*. The variable assignment β'_{pre} denotes:

$$\beta_{pre}\{p_1/I[e_1]\ (\sigma_{pre},\beta_{pre})\ (\sigma_{pre},\beta_{pre}),\ \ldots,\ p_n/I[e_n]\ (\sigma_{pre},\beta_{pre})\ (\sigma_{pre},\beta_{pre})\}\ .$$

This means that all w parameters that appear in w's post-condition are going to be assigned the values the argument expressions in w's call had at *op*'s pre-state.

The state σ'_{post} denotes:

$$\sigma_{post}\{self/I[e_0]\ (\sigma_{pre},\beta_{pre})\ (\sigma_{post},\beta_{post})\}\ .$$

This means that the post-state in which w's post-condition is going to be evaluated is such that the object *self* is the same as operation w target object after being modified by *op*. The variable assignment β'_{post} denotes:

$$\beta_{post}\{p_1/I[e_1]\ (\sigma_{pre},\beta_{pre})\ (\sigma_{post},\beta_{post}),\ \ldots,\ p_n/I[e_n]\ (\sigma_{pre},\beta_{pre})\ (\sigma_{post},\beta_{post})\}\ .$$

This means that all w parameters that appear in w's post-condition are going to be assigned the value the argument expressions in w's call have at *op*'s post-state. ♦

We should also redefine the semantics of the `@pre` construct in order to establish that `obj.m@pre(args)` is the same as `obj.m(args)` for non-query `m(args)`. We have defined extensions to the syntax of OCL pre-conditions in order to be able to express, in pre-conditions, the pre-conditions of non-query operations (not the operation execution). We have defined extensions to the syntax of OCL post-conditions in order to be able to express, in post-conditions, the post-conditions of non-query operations (not the operation execution). The semantics of a non-query operation, when it appears in post-conditions, is the same as the semantics of its own post-condition.

5 Conclusions

Trying to maintain the complexity of contracts manageable, and class coupling as low as possible are important goals when modeling systems of classes. In order to pursue these quality factors, we claimed that the OCL powerful mechanism of navigation through associations should be used moderately when building contracts.

We proposed a way of specifying operation contracts through OCL constraints that offers the abstraction mechanisms needed to maintain contracts at a manageable level of complexity and allow to preserve low coupling. In order to be possible to follow this way of specifying contracts, we claimed the need to represent the effects of non-query operations in post-conditions. We also motivated the need to represent, in a

given operation pre-condition, the pre-conditions of other operations. We proposed an extension of OCL expressions to cope with those needs: we extended the set of operations that are allowed in OCL pre and post-conditions. Then we presented a semantics for those operations capitalizing on the work of [14,16]: we extended their concept of object model and of data signature over an object model, and we defined the semantics of non-query operations over those extended structures.

References

1. K.Arnaut and R.Simon, The .NET Contract Wizard: Adding Design by Contract to Languages other than Eiffel, In Proceedings of TOOLS 39, 2001, California, pp.14-23. IEEE Computer Society, 2001.
2. D.Bartezko, C.Fischer, M.Moller and H.Wehrheim: Jass-Java with assertions, in Workshop on RunTime Verification held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01, 2001.
3. R.B.Findler and M.Felleisen, Contract Soundness for Object-Oriented Languages, OOPSLA 2001.
4. iContract HomePage. http://www.reliable-systems.com/tools/iContract/iContract.htm.
5. C.Larman, *Applying UML and Patterns*, 2nd edition, PH-PTR 2002.
6. G.T. Leavens, K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs, JML: notations and tools supporting detailed design in Java, OOPSLA 2000 Companion.
7. B.Meyer, Object-Oriented Software Construction, 2nd edition, Prentice-Hall PTR, ISBN 0-13-629155-4, 1997.
8. OMG, editor. Object Constraint Language Specification. In OMG Unified Modeling Language Specification, Version 1.3, June 1999, chapter 7.
9. I.Nunes, Design by Contract Using Meta-Assertions, in Journal of Object Technology, vol. 1, no. 3, special issue: TOOLS USA 2002 proceedings, pages 37-56. http://www.jot.fm/issues/issue_2002_08/article3.
10. I.Nunes, Polymorphism in (Meta)-Contract Verification of Object-Oriented Systems, accepted for publication in proceedings of Int. Conf. on Software Engineering Research and Practice (SERP '03). CSREA Press, 2003.
11. Response to the UML 2.0 OCL Rfp (ad/2000-09-03), revised submission, version 1.6, OMG document ad/2003-01-07.
12. M.Ritchers. The USE tool: A UML-based specification environment, 2001. http://www.db.informatik.uni-bremen.de/projects/USE/.
13. M.Ritchers and M.Gogolla, Validating UML models and OCL constraints. In A.Evans, S.Kent, and B.Selic, editors, UML2000 proceedings, vol. 1939 LNCS, pps. 265-277, Springer 2000.
14. M.Ritchers and M.Gogolla, OCL: Syntax, Semantics and Tools. In T.Clark and J.Warmer, editors, Advances in Object Modeling with the OCL, pps. 43-69, Springer, Berlin, LNCS 2263, 2001.
15. J.Warmer and A.Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.
16. J.Warmer, A.Kleppe, T.Clark, A.Ivner, J.Högström, M.Gogolla, M.Richters, H.Hussmann, S.Zschaler, S.Johnston, D.Frankel, and C.Bock. Object Constraint Language 2.0. Technical report, Submission to the OMG, 2001.