# Using Previous Property Values in OCL Postconditions: An Implementation Perspective

Heinrich Hussmann, Frank Finger, Ralf Wiebicke
Dresden University of Technology
Department of Computer Science
Email hussmann@inf.tu-dresden.de

## Abstract

This short paper discusses the practical implementation of the OCL language construct “@pre”, which allows to access the previous value of an object property in the postcondition for an operation. A problem with the current definition of OCL/UML 1.3 is pointed out and possible solutions are suggested.

## 1 Introduction

When an OCL constraint is used as a postcondition for an operation, it is often necessary to refer to the values of object properties before execution of the operation. This is possible in OCL by postfixing the property name with “@pre”. The authors are reporting here from practical experience in implementation and usage of a support tool for OCL which compiles OCL constraints into executable Java code /HDF00/. In this tool, a postcondition is compiled into a Java statement, which is executed at runtime after completion of the respective method. This makes it very easy to access property values in the system state after completion of the method; however, the “previous” values (before execution) are no longer available.

The approach implemented in our compiler for dealing with the “@pre” property is to generate for each “@pre”-access to a property a separate code fragment which is to be executed just before invocation of the method body. This code fragment determines the value for the property and stores it in a temporary local variable. This variable is used after completion of the method, during evaluation of the postcondition, to refer to the vale of the “@pre”-access to the property. This implementation concept is straightforward and relatively easy to realise, so we think it corresponds rather well to the general spirit of OCL.

## 2 Mixing Previous and Actual Values in Navigation Paths

The concept described above works very well for navigation through a composition of property accesses (where all but the last properties are association ends), as long as each navigation path is either fully evaluated in the “pre” state or fully evaluated in the “post” state. Things get more difficult when a navigation path mixes property accesses in the “pre” and “post” state.

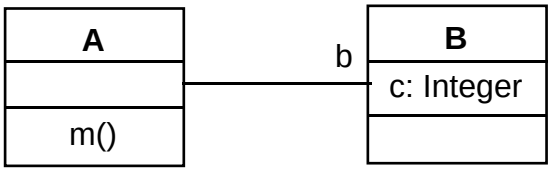
The OCL standard /OCL1.3/ contains a short discussion of such “mixed” navigation paths as well as a short example (on p. 7-21):

“ When the pre-value of a property evaluates to an object, all further properties that are accessed of this object are the new values (upon completion of the operation) of this object. So:

```
a.b@pre.c     -- takes the old value of property b of a, say x
              -- and then the new value of c of x.

a.b@pre.c@pre -- takes the old value of property b of a, say x
              -- and then the old value of c of x.“
```

For a more thorough discussion, let us give some more details for this example. A possible class diagram is the following one:




Based on this class diagram, the expressions of the example from the OCL example can be used in a postcondition for *m()*. The generic expression *a* from the example above is replaced here by *self*:

```
context A::m() post: ... self.b@pre.c ...
```

To study concrete evaluations of the OCL expressions under discussion, let us assume concrete object configurations for the “pre” and post” states.

“Pre” state contains:

| | |
|---|---|
| Object a0 of class A such that | a0.b = b0 |
| Object b0 of class B such that | b0.c = 1 |
| Object b1 of class B such that | b1.c = 3 |

“Post” state contains:

| | |
|---|---|
| Object a0 of class A such that | a0.b = b1 |
| Object b0 of class B such that | b0.c = 2 |
| Object b1 of class B such that | b1.c = 3 |

Informally speaking, this definition of states assumes that the execution of method *m()* on object *a0* changes the value of the *c* property of *b0* as well as the value of the *b* property of *a0*.

Now consider the evaluation of the example expressions for *a0*:

```
self.b@pre.c   = a0.b@pre.c
               = b0.c      (Evaluating b in “pre” state)
               = 2         (Evaluating c in “post” state)
```

The evaluation of this expression is easy to realise with the implementation idea from above: Before execution, i.e. in the “pre” state, the evaluation of the expression *self.b* gives the value *b0*. This value is a Java object reference, which is stored in a variable. When the postcondition is evaluated, the expression *self.b@pre* is replaced by the variable value, i.e. *b0*. Accessing the object *b0* through this reference automatically gives the correct (new) value for property *c*.

The second example is similar:

```
self.b@pre.c@pre    = a0.b@pre.c@pre
                    = b0.c@pre      (Evaluating b in “pre” state)
                    = 1             (Evaluating c in “pre” state)
```

For the implementation, one has to store the value of *b* (which is *b0*) and the value of *c* for *b0* (which is 1). This is still easy to achieve by simply evaluating the path expression leading to each “@pre”-property before execution of the method.

Now consider a third example which is not discussed in the OCL standard:

```
self.b.c@pre    = a0.b.c@pre
                = b1.c@pre      (Evaluating b in “post” state)
                = 3             (Evaluating c in “pre” state)
```

There is no reason why this third example should not be allowed, and the result is clearly defined. However, there is a serious problem in implementing an appropriate evaluation mechanism. Before invocation of the method, it is not yet known what the future value of the *b* property will be, and therefore it is not possible to store the value of *self.b.c@pre* for later use in the postcondition! Evaluation of the expression *self.b.c* in the “pre” state leads to the wrong result “1”.

In order to safe the necessary information before execution of the method *m()*, it is apparently necessary to store all potential values of *x.c* expressions, where *x* is an arbitrary object of class *B*. This causes a significant amount of overhead: There may be very many *B* objects, of which the respective property value is stored (but only for one of them the value is used). Moreover, it is a non-trivial problem to define an appropriate data structure for keeping these values. Altogether, this additional effort is too high compared to the goals of our OCL implementation, to provide an efficient and simple runtime evaluation mechanism. Therefore, our current prototype of the OCL compiler does not evaluate the third example correctly (but gives the same result “1” as for *self.b@pre.cpre*).

## 3 Potential Solutions for the Problem

We see the following basic options to solve the described problem:

*Option 1:* Build complex tools. The obvious first solution is to enhance the tools in such a way that they store the appropriate amount of information to comply with the current standard. As already mentioned above, this leads to poor runtime efficiency (speed and memory) of the implementation and very complex code. So this is not an optimal solution.

*Option 2:* Exclude the problematic case. It is possible to identify those cases where the above-described problem does not appear. The rule is that an “@pre”-access to a property should only follow either an expression which does not change between “pre” and “post” state (like *self*) or another “@pre”-access to a property. (In other words, in a navigation path through properties it is not allowed to navigate from a “post”-property to a “pre”-property. The other direction, from “pre” to “post”, is unproblematic.) This second option is not very convincing, since it restricts the language in a rather awkward way.

*Option 3:* Define a sub-language. Since it is easily possible to identify the problematic cases, an OCL tool can issue a warning message whenever the problematic situation appears. In this case, the user should be aware of the fact that the tool may deliver a wrong result under some circumstances. Theoretically speaking, this means that the tool supports correctly only a sub-language of OCL, and it is the responsibility of the specifier to transform his/her specifications in such a way that they comply to the supported sub-language. (As soon as a calculus for OCL becomes available, this transformation may be carried out even (semi-)automatically.) This is the approach we will follow with our prototype OCL tool.

## 4 Conclusion

In this paper, we have described a rather special problem in constructing efficient tool support for OCL. We think that the OCL community should agree upon a clear strategy for problems of this kind. For success of OCL in practice, it is an absolute necessity that powerful and easy to use tools become available, which for instance simplify the systematic test of software systems based on OCL specifications.

## References

/HDF00/ Heinrich Hussmann, Birgit Demuth, Frank Finger, Modular Architecture for a Toolset Supporting OCL, <<UML>> 2000 Conference, York (UK), October 2000.

/OCL1.3/ Object Management Group, OMG Unified Modeling Language Specification, Version 1.3, Chapter 7 “Object Constraint Language Specification”, 06/08/1999.