View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

Enhancing OCL for Specifying Pre- and Postconditions

Shane Sendall and Alfred Strohmeier

Swiss Federal Institute of Technology
Department of Computer Science
Software Engineering Laboratory
1015 Lausanne-EPFL
Switzerland

email: {Shane.Sendall, Alfred.Strohmeier}@epfl.ch

ABSTRACT This paper proposes a number of enhancements to UML's Object Constraint Language to improve its usability for specifying operations by pre- and postconditions. In particular, we propose notational shortcuts and semantic modifications to OCL so that it can be more effectively used by developers. Also, the paper discusses an approach for specifying, in OCL, events and exceptions which are output and raised by an operation.

KEYWORDS Unified Modeling Language, Object Constraint Language, Pre- and Postcondition, Declarative Operation Specification.

1 Introduction

OCL [11][5] can be used to describe pre- and postconditions of system[1] operations. Pre- and postconditions written in OCL allow developers to precisely express the behavior of a system without necessarily expressing how the behavior is achieved in terms of object collaborations. In this way, pre- and postconditions written in OCL offer an alternative to the graphical diagrams of UML which tend to be solution-oriented.

Our approach for specifying pre- and postconditions of system operations uses OCL. A description of a single system operation is called an operation schema. It is a hybrid of formal approaches such as Z [9] and VDM [3]: hybrid in the sense that they are based on their formal counterparts but are targeted to developers that are more comfortable with procedural programming languages rather than declarative languages. In [6], an approach for mapping use cases to operation schemas has been proposed which highlights how use cases and operation schemas are complementary in making a precise description of system behavior. Our approach has been successfully taught to students and practitioners and used in a number of small-to-medium sized projects.

We believe that OCL offers a notation that is more attractive to the developer in terms of usability and readability when compared to more formal approaches such as Z, VDM, etc. It offers a mathematically less-demanding set of operators and concepts, and it also has a query-like style that is familiar to developers that are versed in database query languages.

The goal of this paper is to propose extensions to OCL that make writing pre- and postconditions less laborious and that result in more readable specifications. We also propose an approach for specifying events and exceptions that are output and raised by the operation. Finally, we propose some guidelines for using OCL when specifying pre- and postconditions.

The paper has many sections, but there is a continuous thread of proposals for enhancements to OCL throughout. Section 2 states our frame assumption which makes possible several of the

1. When we say system we could also be meaning component or subsystem.

syntactic shorthands. Section 3 proposes some shorthand notations that could be employed in OCL to allow more concise conditions. Section 4 proposes a new approach for specifying changes in sets from the pre-state to the post-state; a similar idea is also applied to object attributes whose types are numbers. Section 5 proposes a declaration clause where all variables that are used in a pre-/postcondition are declared. Section 6 proposes a new pseudo-value in addition to *undefined*. Section 7 proposes an approach for specifying events and exceptions. Section 8 poses some open questions. And finally, section 9 concludes the paper.

2 Frame Assumption

The frame of the specification is the list of all variables that can be changed by the operation [4]. Formal approaches such as Z, VDM, Larch, etc. explicitly state the variables that may possibly change and in their postconditions they state what happens to each one of these variables—even for those variables that stay unchanged. However, these approaches soon become cumbersome to write, particularly for specifications that have complex conditional branches. One approach that avoids this extra work is to imply a "... and nothing else changes" rule when dealing with specifications [1]. This approach can significantly reduce the size of the specification and thus increase its readability. However, stating "... and nothing else changes" is not sufficient in certain cases. For example, let's consider a class model of a multi-cabin elevator control system, see figure 1, which consists of a number of lift cabins and floors with two associations between them, *IsFoundAt* and *HasDestination*.

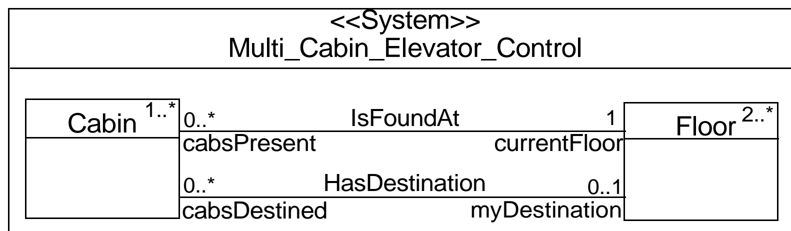

Fig. 1. Partial Class Diagram for a Multi-Cabin Elevator Control System

If we wanted an operation to remove a cabin from service, we would write the following condition in the postcondition, where the context is the system object.

```
self.cabin = self.cabin@pre->excluding (cab)
```

The "... and nothing else changes" rule is not sufficient in this case because it would present the following expansion:

```
self.cabin = self.cabin@pre->excluding (cab)
and IsFoundAt and HasDestination stays unchanged
```

These conditions are contradictory, however, because the last condition requires all associations to stay unchanged, which is clearly not the case according to the first condition.

In the pursuit of practical convenience and in particular conciseness we keep the "... and nothing else changes" rule, but we emphasize that this rule needs to be weakened in situations of implicit removal (example shown above) and implicit override (not shown here).

3 Shorthand Enhancements to OCL

Our proposals for shorthand enhancements to OCL are biased towards a procedural programming language-like style, in the belief not only that developers are more familiar with such styles but that it is in fact a more concise way of expressing conditions.

Proposal 1: Allow the suppression of the else part of the if-then-else construct when the resulting type of the if-then-else expression is boolean.

For case distinctions, we use *if-then-else* conditions, in contrast to many formal approaches which prefer to use the *implies* construct. Without the frame assumption, we would have to write:

```
if car.kind = #Ford then
    john.carsInterestedIn = john.carsInterestedIn@pre + 1
else
    john.carsInterestedIn = john.carsInterestedIn@pre
endif
```

Putting the frame assumption into practice, we are not required to state which objects and associations stay unchanged, and the *else* part is therefore always true:

```
if car.kind = #ford then
    john.carsInterestedIn = john.carsInterestedIn@pre + 1
else
    true
endif
```

Applying proposal 1, the *else* part can be made implicit:

```
if car.kind = #ford then
    john.carsInterestedIn = john.carsInterestedIn@pre + 1
endif
```

An implicit *else true* promotes a smaller, more readable postcondition.

Thus, every time a postcondition has an *if-then* statement (without the *else* part), the expression is of type boolean and an implicit *else true* is implied.

Proposal 2: Add an elsif part to the if-then-else construct.

When dealing with a large number of cases (case distinction), embedding *if-then-else* constructs within another *if-then-else* construct can become problematic, in terms of notational clarity, as the depth of embedding increases. Therefore, adding an *elsif* part to the *if-then-else* construct becomes useful in such situations. The *elsif* addition is directly derivable from nested *if-then-else* constructs, e.g.:

```
if condA then
    A
elsif condB then
    B
else
    C
endif
```

is equivalent to:

```
if condA then
    A
else
    if condB then
        B
    else
        C
    endif
endif
```

Proposal 3: Allow the use of a semi-colon as a replacement of a logical "and", and use it as a boolean expression terminator.

When writing a large pre-/postcondition, separating all expressions by logical "and" operators is cumbersome and reduces the readability of the pre-/postcondition. We, thus, propose to use a semi-colon as a boolean expression terminator. Although, we realize that this gives the conditions a very procedural look, we believe that a semi-colon is less obtrusive for the specification writer and has a more natural look, as shown by this simple example:

```
boolExprA
and if boolExprB then
    boolExprC
    and boolExprD
endif
and boolExprE
```

is equivalent to:

```
boolExprA;
if boolExprB then
    boolExprC;
    boolExprD;
endif;
boolExprE;
```

Proposal 4: Allow the use of the aggregate notation for denoting composite values.

We propose the optional use of an Ada-style aggregate notation for denoting composite values. The components of a record, the value attributes of an object, and the parameters of an event all correspond to composite values.

An aggregate can be written using named associations, i.e. a value is associated with each component denoted by its name, e.g. the attribute values of a company object:

```
(name => "Microsoft", headquarters => "Richmond", budget => 50.0E9)
```

Positional notation is also possible, but then the ordering of the components must be agreed upon by some convention, e.g. alphabetical order of the attribute names for objects:

```
(50.0E9, "Richmond", "Microsoft")
```

It is perfectly possible to nest aggregates, e.g.:

```
(firstName => "Denis", lastName => "Maillat", birthday => (1940, 1, 9))
```

The advantage of aggregates is that related values are kept together in one place. Moreover, it is possible to check that values are defined for all components, e.g. in an expression like:

```
company = (name => “Microsoft”, headquarters => “Richmond”, budget => 50.0E9)
```

which is equivalent to:

```
company.name =“Microsoft”;
company.headquarters = “Richmond”;
company.budget = 50.0E9;
```

It is sometimes useful to be able to qualify an aggregate by its type, e.g. the record type, the class name or the event type, yielding a so-called qualified aggregate. We propose to use the Ada-like “tick” notation, i.e. the type name precedes the aggregate, separated by an apostrophe, e.g.

```
Company’(50.0E9, “Richmond”, “Microsoft”)
```

Proposal 5: The oclIsNew property can be optionally parameterized by an aggregrate which states the initial values of all the attributes of the object created with the execution of the operation.

We propose to allow in the oclIsNew property a parameter that is an aggregate and that specifies the values of all the value attributes of the object:

For example, a postcondition could state:

```
obj.oclIsNew (make => “Ford”, year => 2000);
```

which means that the object *obj* was created in the execution of the operation, and all its value attributes, i.e., *make* and *year*, were given initial values, *“Ford”* and *2000*, respectively.

The above expression is directly equivalent to:

```
obj.oclIsNew;
obj.make = “Ford”;
obj.year = 2000;
```

The proposed notation ensures that all attributes of a newly created object were initialized in a single place, and none of them were forgotten.

Proposal 6: The oclIsNew property can be applied to collections; it then takes one parameter which signifies the number of new objects that are created with the execution of the operation.

We propose to allow the creation of a collection of objects by introducing the oclIsNew property for collections, where the oclIsNew property takes as parameter the number of elements to be created. For example, a postcondition could state (assuming *colX: Collection (X)*):

```
colX.oclIsNew (n);
```

which is equivalent to:

```
colX->forall (x: X | x.oclIsNew);
colX->size = n;
```

Both conditions state that there were *n* objects of class *X* created with the execution of the operation and these objects are members of the collection *colX*.

4 OCL Operators

In OCL, when describing a change to the contents of a set in the postcondition, one is often forced to state the contents of a set in the post-state in relation to its contents in the pre-state, e.g.:

```
john.myFavoriteStamps = john.myFavoriteStamps@pre -> including (stamp345);
```

This approach has two disadvantages: it is not very concise when dealing with sets that are constructed by association traversal (imagine an expression that traverses two or more associations to form the set), and it does not offer incremental addition/removal of elements, i.e., we are forced to state in one place what happens to the set. We propose to apply the idea of the minimal set condition to OCL postcondition semantics. For each class and each association, we will consider their sets of instances and links, and claim that these are all minimal sets after execution of the operation. Otherwise stated, if C is a class, if Set(C)@pre is its set of its instances before the execution of the operation, and Set(C) is its set of its instances after the execution of the operation, then Set(C) is the minimal set containing Set(C)@pre and fulfilling the postcondition. Intuitively, Set(C) can be constructed by adding to Set(C)@pre all instances of C created by the operation. The same kind of idea can be applied to the links of an association A: Set(A) is then the minimal set containing Set(A)@pre and fulfilling the postcondition. The rule must hold for all classes and associations. There is a slight problem when we allow for the destruction of objects or removal of association links. For defining the semantics of the operation schema, the idea is then to gather the deleted entities into a temporary set, and rephrase the rule in the following way: let C be a class, let's denote by Minus(C) the set of instances of C destroyed by the operation, then Set(C) $\cap$ Minus(C) is empty, and Set(C) $\cup$ Minus(C) is the minimal set containing Set(C)@pre.

Moreover, and as already stated in the frame assumption, if an object is destroyed during the execution of an operation, i.e. the postcondition states that the object is destroyed, then all the association links it participates in are destroyed too, without having to say it explicitly in the postcondition!

Proposal 7: Apply the minimum set condition to all sets in OCL postconditions.

The minimum set principle can be shown on the elevator example of figure 1. For example, a postcondition may state that a cabin has been added to the system's set of cabins:

```
self.cabin->includes (cab);
```

this condition is therefore equivalent to the following:

```
self.cabin = self.cabin@pre->including (cab);
```

this is due to the fact that no other statements have been made about the state of *self.cabin*. Minimum sets can be very useful for stating postconditions incrementally, i.e., the final state of the set is the set in its pre-state with the addition of all the elements that were mentioned in *includes*/*includesAll* and the difference of all the elements that were mentioned in *excludes*/*excludesAll*. For example, we could define a fragment of the postcondition of an imagined operation called *updateCabinsOnline*, which adds and removes cabins to the system's set of cabins.

```
if currentMode = #maintenance then
    self.cabin->includes (serviceLift);
endif;
if allRequests->exists (r | r.status = #late) then
    self.cabin->includes (reservedLift);
endif;
self.cabin->excludes (cabPriority);
```

Clearly, this would be much harder to state using only set equalities. Also, when using equalities, it may happen that several exist in the postcondition that make an assertion about the same set, and then they are either equivalent, or they are inconsistent. Inconsistency cannot result from incremental modifications of a set, and we believe therefore that this approach is easier to use.

Proposal 8: Introduce the operators, "+=" and "-=", to allow the values of attributes, that are of a number type (real, integer, etc.), to be defined incrementally in a postcondition.

Similarly to the idea of minimum sets, we can reuse the same idea for object attributes that are number types. We propose to use the operators, "+=" and "-=". Thus, the value of the object attribute in the post-state is equivalent to the value in the pre-state plus all the right-hand sides of all "+=" operators used in the postcondition that refer to the object attribute, and minus all the right-hand sides of all "-=" operators that refer to the object attribute. For example,

```
x += 5;
x -= 4;
```

is equivalent to:

```
x = x@pre + 1;
```

Such a notation is especially useful when there are many case distinctions.

Unfortunately, the facility cannot be extended to more complex expressions (e.g. multiplication) because it relies on the commutativity of additions and subtractions.

5 Declarations

Currently, OCL only supports *let* statements for declaring variables that are used in pre- and postconditions. However, we believe that defining a single declaration clause for a pre- and post-condition pair is cleaner than possibly many *let* clauses that are dispersed within the conditions themselves.

Proposal 9: Add an explicit Declares clause for declaring all objects, collections and data values that are used in a pre- and postcondition pair.

Our proposal for a separate declaration clause is in line with the proposal of Cook et al. [2]. The clause uses the keyword **Declares** to indicate the start of the declaration. The declaration states all objects, collections and data values that are used in the pre- and postconditions. For example,

```
Declares:
    objX: X;
    setY: Set (Y);
```

Each entity declaration may be associated with a value expression, e.g.,

Declares:

```
atBottom: Boolean ::= true;
objX: X ::= setX->select (x: X | x.id = 34);
setY: Set (Y) ::= self.allMyYs;
```

A description of the grammar and usage of our approach (operation schemas) can be found here [10].

6 OCL Undefined

In OCL, a declared entity can be *undefined* in a number of different situations, e.g., type mismatch, improper expression, etc. It is also *undefined* in situations when no object reference exists, e.g.,

```
objY: Y ::= setOfYs->select (y: Y | y.id = 36)
```

The object *objY* is *undefined* if the select operator applied to the set *setOfYs* results in an empty set. However, we often find in our approach that we would like to differentiate between an expression that is not well-formed and one that just has a void reference. We propose, therefore, to introduce a new pseudo-value for void references. In this situation, the entity will have a *NULL* value. And thus we can test for this value in a postcondition.

Proposal 10: Introduce a pseudo-value, in addition to undefined, called NULL which can be used in cases when an object variable has a void reference.

Elaborating on the previous example, if the result of the select operation (RHS) evaluates to a single element then the declaration is well-formed and *objY* is an instance of *Y* that is present in the set *setOfYs*. However, if the RHS expression evaluates to more than one element then the declaration is type inconsistent, thus it is *undefined*. Otherwise if the RHS evaluates to an empty set then our proposal is that the declaration is still well-formed, i.e., it is not *undefined*, but *objY* has a *NULL* reference value.

The justification for differentiating the two cases is that we often want to throw an exception (discussed in the next section) if the object is *NULL*, but if the expression is *undefined* then the specification is invalid or incorrect.

7 Output Events and Exceptions

In our approach, we specify in the postcondition not only the changes to the system state, but also the events that are output with the execution of the operation. All output events are sent to actor instances. An event is a UML signal that is either normal or exceptional according to the value of its tagged value. Therefore in our approach, an exception is simply a kind of event which is output to an actor(s).

Declaration of output events are written in a separate clause called the Sends clause. The Sends clause is broken up into three (optional) sub-clauses: actor types together with the event types that they may receive, named event occurrences, and any sequencing constraints on named event occurrences.

For example, a sends clause could look like:

Sends:

```
Motor::{Stop}, Door::{Open};
stopLift: Stop, openLiftDoor: Open;
Sequence {stopLift, openLiftDoor};
```

The first line declares that the actor classes *Motor* and *Door* are allowed to be sent events of type *Stop* and *Open*, respectively. The second line declares two event occurrences. And the third line is a sequencing constraint which states that the event *stopLift* is sent before the event *openLiftDoor*.

The basic rule is that if there is no explicit ordering constraint between two events, then they are produced by the operation in any order. We do not state ordering constraints inside the postcondition, because there is no ordering between the conditions forming the postcondition.

The sending of events in the postcondition is declaratively described by stating that the event occurrence was placed on the event queue of the target actor instance. The underlying semantic implications of this can be found elsewhere [7].

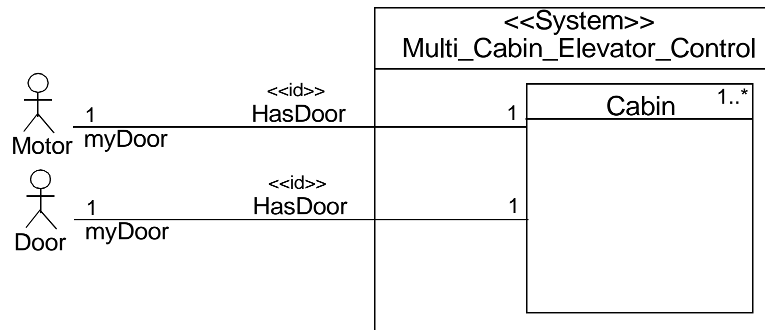

Fig. 2. A partial class diagram for the multi-cabin elevator control system which shows the traversable associations (stereotyped <<*id*>>) between the system and its actors

For example, in a postcondition one could write (according to figure 2):

```
if shouldStopLift then
    (cab.myMotor).events->includes (stopLift);
    (cab.myDoor).events->includes (openLiftDoor);
endif;
```

This example states that if *shouldStopLift* is true then the event instances *stopLift* and *openLiftDoor* are sent to the actor instances *cab.myMotor* and *cab.myDoor*. These actor instances are defined by traversing from the *cab* object to the *Motor*/*Door* actor instances via the *HasMotor*/*HasDoor* associations.

Proposal 11: The output of an event in an OCL postcondition is achieved by placing an event occurrence on an actor event queue(s).

8 Open Questions

OCL was created with the main purpose of providing navigation of UML models and consequently it is asymmetric with respect to associations. OCL's style of navigation has quite some advantages, e.g. there are not too many operators and they are easy to understand, but there are also some serious drawbacks.

First of all, the addition of a new link between two objects can be easily misinterpreted. For example, an expression like the following (see figure 1),

```
cab.cabsDestined->includes (groundfloor);
```

which means there is a new link between *cab* and *groundfloor*, can be easily misinterpreted as being a unidirectional link from *cab* to *groundfloor*, whereas the condition is strictly equivalent to:

```
groundfloor.myDestination->includes (cab);
```

More seriously, it is impossible to use the navigational notation for higher-order associations, and at least awkward to use it for handling association classes.

One solution to this problem would be to handle associations like sets of tuples (what they really are). The previous example could then be rewritten (where *self* is the system object, an instance of *Multi_Cabin_Elevator_Control*):

```
self.hasDestination->includes ((cab, groundfloor));
```

But, we are not sure whether this is possible in OCL, i.e., is it possible to traverse from a composite to an association in OCL?

9 Conclusion

This paper proposed a number of enhancements to UML's Object Constraint Language to improve its usability for defining pre- and postcondition of operations. In particular, we proposed notational shortcuts and semantic modifications to OCL so that it can be more effectively used by developers. Also, the paper discussed an approach for specifying events and exceptions in an OCL postcondition.

For examples of our approach applied to several small case studies see [8].

References

[1] A. Borigda, J. Mylopoulos and R. Reiter. *On the Frame Problem in Procedure Specifications*. IEEE Transactions on Software Engineering, Vol. 21, No. 10: October 1995, pp. 785-798.

[2] S. Cook, A. Kleppe, R. Mitchell, J. Warmer, A. Wills. *Defining the Context of OCL Expressions*. Second International Conference on the Unified Modeling Language: UML'99, Fort Collins, USA, 1999.

[3] C. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1986.

[4] C. Morgan. Programming from Specifications. Second Edition, Prentice Hall 1994.

[5] OMG Unified Modeling Language Specification, Version 1.3, June 1999; published by the OMG Unified Modeling Language Revision Task Force on its WEB site: http://uml.shl.com/artifacts.htm

[6] Shane Sendall and Alfred Strohmeier. *From Use Cases to System Operation Specifications*. UML 2000 — The Unified Modeling Language: Advancing the Standard, Third International Conference, Stuart Kent and Andy Evans (Ed.), LNCS, York, UK, October 2-6, 2000.

[7] Shane Sendall and Alfred Strohmeier. *Specifying System Behavior in UML*. Technical Report 2000/???, Swiss Federal Institute of Technology in Lausanne, Switzerland, 2000 (submitted for publication).

[8] Shane Sendall. *Specification Case Studies*. Electronic Resource available at http://lglwww.epfl.ch/~sendall/case-studies/

[9] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1989.

[10] Alfred Strohmeier and Shane Sendall. *Operation Schemas*. Electronic Resource available at http://lglwww.epfl.ch/~sendall/operation-schemas/

[11] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison-Wesley 1998.