Ambiguity issues in OCL postconditions

Jordi Cabot

Estudis d'Informàtica i Multimèdia, Universitat Oberta de Catalunya
Rbla. Poblenou, 156 E08018 Barcelona, Spain
jcabot@uoc.edu

Abstract: There are two different approaches to specify the behavior of the operations of an Information System. In the imperative approach, the operation effect is defined by means of specifying the set of actions (creation of objects and links, attribute updates…) to apply over the system state. With the declarative approach, the effect is defined by means of contracts stating the conditions that the system state must satisfy *before* (precondition) and *after* (postcondition) the operation execution.

From a specification point of view, the declarative approach is preferable. The main issue regarding declarative specifications is their ambiguity. Commonly, there are many different system states that satisfy an operation postcondition. However, in general, only one of them is the one the designer had in mind when defining the operation, and thus, that state should be the only one considered valid at the end of the operation execution. In this paper, we identify some of the common ambiguities appearing in OCL postconditions and provide a default interpretation for each of them in order to improve the usefulness of declarative specifications.

1. Introduction

A conceptual schema (CS) is the representation of the general knowledge of a domain. In conceptual modeling, we call Information Base (IB) the representation of the state of the CS (the set of existing objects and links) in the Information System.

The state of the IB changes due to the application of the modifying actions issued by the execution of the operations defined in the CS. An action is the fundamental unit of behavior specification [11]. Among the possible actions we have the creation of a new object or link, its deletion, the update of an attribute and so forth.

The effect of the operations can be specified in two different ways, *imperatively* or *declaratively* [16]. In an imperative specification the designer explicitly defines the set of actions to be applied over the IB. In a declarative specification the designer defines a contract for each operation. The contract consists of a set of pre and postconditions. A precondition defines a set of conditions over the operation input and the IB that must hold when the operation is invoked. The postcondition states the set of conditions that must be satisfied by the IB at the end of the operation execution. We assume that pre and postconditions are specified in OCL.

From a specification point of view, the declarative approach is preferable since it allows a more abstract definition of the operation effect and defers until a later stage most of the implementation issues [16]. In this sense, the imperative definition of an operation can be regarded as a lower-level definition of the operation effect. Moreover, declarative specifications are more concise than their imperative counterparts. This favours the readability of the contract specifications.

The main problem regarding declarative specifications is that they are *underspecifications* [16], i.e. in general there are several possible states of the IB that may verify the postcondition of an operation contract. This means that a declarative specification may have several equivalent implementations (i.e. imperative versions). We have a different version for each set of actions that, given a state of the IB verifying the precondition, evolve the IB to one of the possible states verifying the postcondition of the operation contract. This problem is not specific of OCL declarative specifications but common to other purely declarative languages used to specify operation contracts as JML, Eiffel or logic languages. However, it is especially relevant in OCL contracts because of the high expressiveness of the OCL.

The definition of a postcondition precise enough to characterize a single state of the IB is cumbersome and error-prone [5],[15]. For instance, it would require to specify in the postcondition all objects and links not modified by the operation (*frame problem* [5]). There are other ambiguities too. Consider a postcondition as $o.at_1=o.at_2$, where o represents an arbitrary object and at_1 and at_2 two attributes. Given an initial state of the IB, states obtained after assigning to at_1 the value of at_2 satisfy the postcondition. However, states where at_2 is changed to hold the value of at_1 or where the same value is assigned to both attributes satisfy the postcondition as well. Strictly speaking, all three interpretations are correct since all satisfy the postcondition. However, most probably, only the first one (where we assign the value of at_2 over at_1) represents the behaviour the designer meant when defining the postcondition.

We believe it is really important designers are aware of the different ambiguities included in a postcondition and of the whole set of possible states that satisfy the postcondition due to such ambiguities. Then, they may decide to extend/rephrase the postcondition in order to prevent some of the undesired states. This may imply to complement the postcondition with additional information regarding the parts of the system state that cannot be changed by the operation [4, 5], or even, to mix the declarative specification with imperative constructs [2].

In this paper we follow an alternative approach. For each ambiguous OCL expression that may appear in a postcondition we define its default interpretation. These default interpretations represent usual designers' assumptions about how that expression should be tackled when implementing/validating the operation. As an example, a default interpretation for the $X=Y$ ambiguous expression commented above is that X should take the value of Y. According to this interpretation, states where Y takes the value of X or where both take a different value, should be considered invalid.

The detection of common types of ambiguities and the proposal of a default interpretation for each of them are the main contributions of this paper. We believe that our approach offers several benefits. It improves the quality of the applications

by means of detecting possible errors in the specification of the operation contracts and by ensuring that the application behaviour is better aligned with the designer intentions. Moreover, it opens the possibility of leveraging current MDA and MDD methods and tools by allowing code-generation from declarative specifications (once translated to an equivalent imperative version) in the final technology platform. Until now, an automatic translation was not feasible because of the high number of possible imperative versions for each declarative specification. The default interpretation reduces the number of possible alternatives (to just one, in the best case) and could be used to guide the code-generation process. As an additional benefit, the discussion presented in the paper may help to achieve a deeper understanding of the semantics of operation contracts and their ambiguity problems.

The rest of the paper is structured as follows. Next section presents a set of ambiguous OCL expressions and provides their default interpretation. Section 3 covers some inherently ambiguous postconditions. Section 4 compares our approach with related work and, finally, section 5 present some conclusions and further research.

2. Ambiguities in operation contracts

Given the contract of an operation *op* and an initial state *s* of an IB (where *s* verifies the precondition of *op*) there exist, in general, a set of final states $set_{s'}$ that satisfy the postcondition of *op*. All implementations of *op* leading from *s* to a state $s' \in set_{s'}$ must be considered correct. Obviously, *s'* must also be consistent with the integrity constraints defined in the CS, but, assuming a strict interpretation of operation contracts [12], the verification of those constraints does need to be part of the contract.

Even though, strictly speaking, all states in $set_{s'}$ are correct, only a small subset $acc_{s'} \subset set_{s'}$ could probably be accepted as such by the designer. The other states satisfy the postcondition but do not represent the expected behaviour of the operation intended by the designer. In most cases $|acc_{s'}| = 1$ (i.e. from the designer's point of view there exists only a state *s'* that satisfies the postcondition).

The first aim of this section is to detect some common OCL expressions that, when appearing in a postcondition, increase the value of $|set_{s'}|$, that is, the expressions that introduce an ambiguity problem in the operation contract. We also consider the frame problem, which, in fact, appears because expressions required to specify conditions over the state of parts of the IB are missing in the postcondition.

Ideally, once the designer is aware of the ambiguities appearing in an operation *op*, he/she should define the postcondition of *op* precise enough to ensure that $acc_{s'} = set_{s'}$. However, this is not feasible in practice since then postconditions become much longer, cumbersome and error-prone [5],[15]. Therefore, the second aim of this section is to provide, for each ambiguity expression, a default interpretation that solves the ambiguity problem by selecting from $set_{s'}$ those that most probably represent the intention of the designer at the moment of writing that part of the postcondition.

The default interpretations express common assumptions used during the specification of operation contracts. They have been developed after analyzing many examples of operation contracts of different books, papers and case studies and comparing them, when available, with the operation textual description.

In what follows we present different ambiguity problems. We provide in the appendix a list of transformation rules we can apply over the original OCL expressions to extend the set of postconditions covered in this section.

2.1 Ambiguity 1: "State of objects and links not referenced in the postcondition"

In general, OCL expressions appearing in a postcondition only restrict the possible values of part of the elements of the IB (in particular, the ones referenced in the operation). The values of the rest of objects and links in the IB are left undefined, and thus, any state that modifies them is acceptable as long as the state verifies the postcondition.

Default interpretation: Nothing else changes.
This interpretation represents the most common adopted solution to the frame problem. It states that objects not explicitly referenced in the postcondition should remain unchanged in the IB. This implies that they cannot be created, updated or deleted during the transition to the new state of the IB. Similarly, links of associations not traversed during the evaluation of the postcondition cannot be created nor deleted.

Besides, for those objects that appear in the postcondition, only the properties (either attributes or association ends) mentioned in the postcondition definition may be updated.

2.2 Ambiguity 2: "Equality expressions"

By far, the most common operator in postconditions is the equality operator. Given an expression *X.a=Y.b* (where *X* and *Y* are two arbitrary OCL expressions and *a* and *b* two properties), there are three kinds of changes over the initial state resulting in a new state satisfying the expression. We can either assign the value of *b* to *a*, assign the value of *a* to *b* or assign to *a* and *b* an alternative value *c*.

Note that if either operand of the equality comparison is a constant value or is defined with the *@pre* operator then just a possible final state exists. Since the value of that operand cannot be modified, the only possible change is to assign its value to the other operand. This applies also to other ambiguities described in this section.

Default interpretation: The order of the operands in the equality expression reflects the desired change
We believe that the common interpretation for an expression *X.a=Y.b* is that *a* must take the value of *b*. This should be the only final state considered valid at the end of the operation. If a designer was meant to define that b should take the value of *a* he/she would have surely written the expression as *Y.b = X.a*. In the same way, if the desired final state was that the one where the value of *a* and *b* was equal to *c*, most

probably, he/she would have included in the postcondition the expression *X.a=c and X.b=c*.

2.3 Ambiguity 3: "if-then-else expressions"

An *if-then-else* expression evaluates to false when the *if* condition is satisfied but the *then* condition is evaluates to false or, reversely, when the *if* condition evaluates to false but the *else* expression is not satisfied. Therefore, given an *if-then-else* expression included in a postcondition *p*, there are two groups of final states that satisfy *p*: 1 – States where the *if* and the *then* condition are satisfied or 2 – states where the *if* condition evaluates to false and the *else* condition evaluates to true.

Default interpretation: Do not falsify the *if* clause.
We believe the desired behaviour for *if X then Y else* Z expressions is to evaluate *X* and enforce *Y* or *Z* depending on the value of X. According to this interpretation, states obtained by means of falsifying *X* do not represent the designer's intention. Implementations of postconditions that modify the *X* expression to ensure that *X* evaluates to false are not acceptable (even if, for some states of the IB, it could be easier falsifying *X* to always avoid enforcing *Y* instead of enforcing *Y* or *Z* depending on the value of *X*).

2.4 Ambiguity 4: "*includes and includesAll* expressions"

Given an initial state *s* and a postcondition of type *X->includesAll(Y)*, all final states where, at least, the objects of *Y* have been included in *X* satisfy the postcondition. However, states that, apart from the objects in *Y*, add other objects to *X* also satisfy the postcondition. Moreover, another possible group of final states that satisfy the expression are those where *Y* evaluates to an empty set, since by definition all sets include the empty set.

For *includes* expressions we follow the same reasoning. The only difference is that, for those expressions, *Y* does not return a set of objects but a single instance.

Default interpretation: Minimum number of insertions over the collection *X* and no changes over *Y*
Following this assumption, the new state *s'* should be obtained by means of adding to *s* the minimum number of links necessary to satisfy the operation postcondition. For expressions such as *X->includes(Y)* or *X->includesAll(Y)* (where *X* and *Y* are two arbitrary OCL expressions) a maximum of *X@pre->size()+Y->size()* links must be created. States including additional insertions are not acceptable.

States where *Y* is modified to ensure that it returns an empty result are neither acceptable.

This interpretation may seem similar to the one defined to deal with the frame problem. The difference is that there we addressed the creation and deletion of links of associations not referenced in the postcondition, while this one tackles minimal modifications over elements that are referenced in the postcondition.

2.5 Ambiguity 5: "*excludes and* excludes*All* expressions"

Given an initial state *s* and a postcondition of type *X->excludesAll(Y)*, all final states where, at least, the objects of *Y* have been removed from the collection of objects returned by *X* satisfy the postcondition. However, states that, apart from the objects in *Y*, remove other objects from *X* also satisfy the postcondition.

Additionally, states where *Y* does not return any object also satisfy the postcondition since then, clearly, *X* also excludes all the objects in *Y*.

For *excludes* expressions we follow the same reasoning. The only difference is that for those expressions *Y* does not return a set of objects but a single instance.

Default interpretation: Minimum number of deletions over the collection *X* and no changes over *Y*

According to this interpretation, the acceptable states are those where the new state *s'* is obtained by means of adding to the initial state *s* the minimum number of links necessary to satisfy the operation postcondition and where *Y* has not been modified to ensure that it returns an empty set.

For expressions like *X->excludes(Y)* or *X->excludesAll(Y)* a maximum of *X@pre->size()–Y->size()* may be deleted.

2.6 Ambiguity 6: "*forAll* iterators"

There are two possible approaches to ensure that an expression like *X ->forAll(Y)* (where *X* represents an arbitrary expression and *Y* a boolean expression) is satisfied in a new state of the IB. We can either ensure that, in the final state, all elements in *X* verify *Y* or to ensure that *X* results in an empty collection, since a *forAll* iterator over an empty collection always returns *true*.

Default interpretation: Do not empty the collection expression

The desired behaviour is to ensure that all elements in *X* verify the condition *Y* and not to force *X* to be empty.

2.7 Ambiguity 7: "*oclIsTypeOf* and *oclIsKindOf* operators"

The condition *obj.oclIsTypeOf(C)* requires type *C* to be one of the classifiers of *obj*. Therefore, new states where *C* is added to the list of classifiers of *obj* satisfy the condition, regardless of any other modifications to the list of classifiers of *obj*. States where additional classifiers have been added or some classifiers removed from *obj* also satisfy the condition.

Similarly with *obj.oclIsKindOf(C)* expressions. The only difference is that, for these expressions, we only require that *C* or one of its subtypes is added to *obj*.

On the contrary, conditions like *not obj.oclIsTypeOf(C)* establish that in the new state *obj* cannot be instance of *C* (and likewise with *not obj.oclIsKindOf(C)*, where *obj* cannot be instance of *C* or instance of one of its subtypes). Therefore all states

verifying this condition are valid even if they add/remove other classifiers from the list of classifiers of *obj*.

Default interpretation: Minimum number of specializations/generalizations
When defining this kind of expressions, we assume that the desired behaviour is just to express the minimum set of specializations/generalizations required to satisfy the postcondition. Therefore, new states where *obj* has been specialized also to other classifiers apart from *C* are not valid (unless required due to other expressions appearing in the postcondition). As an example, for expressions like *obj.oclIsKindOf(C)* only the classifier *C* or one of its subtypes may be added to *obj* during the transition to the new state.

For *not obj.oclIsTypeOf(C)* expressions, no other classifiers (apart from *C*) should be removed from *obj*. Similarly, conditions like *not obj.oclIsKindOf(C)* require that *obj* is generalized to a direct supertype of *C*. No other generalizations should be applied.

3. Inherently ambiguous postconditions

In some sense, all postconditions may be considered ambiguous since, in general, there are several states of the IB that verify a given postcondition. However, for most postconditions, the default interpretations presented in section 2 allow to disambiguate them by means of determining which state is the preferred among the possible ones.

Nevertheless, some postconditions are inherently ambiguous (also called non-deterministic [2]). We cannot define a default interpretation for them since, among all possible states satisfying the postcondition, there does not exists a state clearly more appropriate than the others. As an example assume a postcondition with an expression $a>b$. There is a whole family of states verifying the postcondition (all states where *a* is greater than *b*), all of them equally correct, even from the designer point of view or, otherwise, he/she would have expressed the relation between the values of *a* and *b* more precisely (for instance saying that $a=b+c$).

We believe it is worth to identify these inherent ambiguous postconditions since most times the designer does not define them on purpose but by mistake. Table 3.1 shows a list of expressions that cause a postcondition to become inherently ambiguous. The list is not exhaustive but contains the most representative ones.

Table 3.1 List of ambiguous expressions

Expression	Ambiguity description
post: B_1 ***or … or*** B_n	At least a B_i condition should be true but it is not defined which one/s
X<>Y, X>Y, X>=Y, ***X<Y, X<=Y***	The exact relation between the values of *X* and *Y* is not stated

X+Y=W+Z ***(likewise with -,*,/,…)***	The exact relation between the values of the different variables is not stated.
X->exists(Y)	An element of *X* must verify *Y* but it is not defined which one
X->any(Y)=Z	Any element of *X* verifying *Y* could be the one equal to *Z*
X->union(Y)=Z ***(likewise with ∩,−,…)***	There are at least 2^n different ways to distribute the elements of Z between X and Y (n=|Z|) to ensure that the expression is satisfied
X.p->sum()=Y	There exist many combinations of single values that once added may result in *Y*
X.n_1.n_2…n_n=Y	We can either assign *Y* to the object/s obtained at the end of the navigation n_n or to change an intermediate link to obtain at n_n an object/s equal to *Y*
$Op_1()$ = $op_2()$	The values returned by two operations must coincide. Depending on its body, there may be several alternative ways to satisfy this equality

4. Related Work

Ambiguity problems of declarative specifications have been poorly studied apart from the frame problem [5],[15] and a couple of basic assumptions regarding object (and collection) creations and removals [14]. The most usual strategy to deal with the ambiguity problems in declarative specifications forces the designer to explicitly state in the postconditions which elements of the IB change and which remain the same (this is the case of [5], [4] and formal languages as Z, VDM or Larch). More recent approaches, as [2], try to combine the OCL with imperative languages to permit designers specify more clearly the semantics of the contracts.

However, none of them tries to automatically disambiguate declarative specifications without burdening the designer with the task of defining additional information in the postconditions. Besides, as we have commented before, the specification of completely precise postconditions is not feasible in practice. Nevertheless, such approaches could be useful to deal with the problematic postconditions of section 3.

The support for declarative specifications in current CASE and MDA tool is rather limited. Most of them only deal with imperative specifications (see [10] as a representative example). There exist several OCL tools allowing the definition of operation contracts (see, among others, [3], [6], [8],[7]). However, during the code-generation phase, the contracts are simply added as validation conditions. Contracts are transformed into *if-then* clauses that check at the beginning and at the end of the operation if the pre and postconditions are satisfied (and raise an exception otherwise). An exception is [1] that is able to check the correctness of an implementation with respect to its contract. None of them considers the ambiguity problems of the declarative specification, any state satisfying the postcondition is

considered valid without considering that, in fact, the designer would regard some of these *valid* states as invalid ones.

5. Conclusions and further research

In this paper we have detected several OCL expressions that, when included in a postcondition specification, introduce ambiguity problems. We define that a postcondition is ambiguous when it is an underspecification, i.e. when there are several states of the IB that satisfy it, even though, most probably, the designer would only consider as valid states a (small) subset of them.

Therefore, regardless how the designer decides to handle these ambiguity issues, we believe it is important he/she be aware of them since they may even indicate an error in the declarative specification (especially for the inherent ambiguities of section 3). In the declarative specifications we analyzed and according to the contract definition in natural language, many times the designers were unaware of the ambiguities present in their postconditions.

Additionally, we propose an approach to automatically disambiguate the postconditions by means of providing a default interpretation for each kind of ambiguous expression. The default interpretation determines, from the possible states satisfying the ambiguous expression, the one/s that best represents the designer's intention when specifying the postcondition.

Our proposed interpretations require some strong assumptions about how the postconditions are specified, yet we believe the assumptions reflect the way designers tend to (unconsciously?) specify the postconditions. They have been validated against two case studies of real-life applications (a Car Rental System [9] and an e-marketplace system [13]) as well as with other examples appearing in different books, papers and tutorials. Nevertheless, we would like to have the opportunity to discuss them among the members of the OCL community in order to see whether they are accepted or alternatives ones should be proposed (or even if there is no agreement in the existence of such default semantics for postconditions). Obviously, our approach cannot be applied when the proposed assumptions are not followed since then we may end up restricting some possible final states that should be considered valid.

There are other directions in which we plan to continue our work. First, we plan to extend the set of ambiguous expressions we detect. To facilitate an empirical validation of our approach we are also interested in developing an animator tool that given an operation contract and a state of the IB, applies our default interpretations to the contract postcondition in order to compute the new state for the IB. Moreover, we want to explore the possibility of generating (semi) automatically the implementation of an operation starting from its declarative specification. This translation would be useful to leverage current MDA tools, which only support code-generation from imperative specifications.

Acknowledgments

Thanks to people of the GMC group (and especially to Anna Queralt) for their many useful comments to previous drafts of this paper. This work has been partly supported by the Ministerio de Educacion y Ciencia (project TIN 2005-06053) and by the Generalitat de Catalunya (grant 2006 BE 00062).

References

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P. H.: The KeY tool, Integrating object oriented design and formal verification. Software and Systems Modeling 4 (2005) 32-54
2. Baar, T.: OCL and Graph-Transformations - A Symbiotic Alliance to Alleviate the Frame Problem. In: Proc. MODELS'05 Workshop on Tool Support for OCL and Related Formalisms, Technical Report, LGL-Report-2005-001 (2005) 93-109
3. Babes-Bolyai University. Object Constraint Language Environment 2.0. http://lci.cs.ubbcluj.ro/ocle/
4. Beckert, B., Schmitt, P. H.: Program verification using change information. In: Proc. 1st Int. Conf. on Software Engineering and Formal Methods (SEFM'03), (2003) 91-101
5. Borgida, A., Mylopoulos, J., Reiter, R.: On the frame problem in procedure specifications. IEEE Transactions on Software Engineering 21 (1995) 785-798
6. Borland. Borland® Together® Architect 2006. http://www.borland.com/us/products/together/
7. Dresden. Dresden OCL Toolkit. http://dresden-ocl.sourceforge.net/index.html
8. Dzidek, W. J., Briand, L. C., Labiche, Y.: Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In: Proc. MODELS 2005 Workshops, LNCS, 3844 (2005) 10-19
9. Frias, L., Queralt, A., Olivé, A.: EU-Rent Car Rentals Specification. LSI Research Report, LSI-03-59-R (2003)
10. Mellor, S. J., Balcer, M. J.: Executable UML. Object Technology Series. Addison-Wesley (2002)
11. OMG: UML 2.0 Superstructure Specification. OMG Adopted Specification (ptc/03-08-02) (2003)
12. Queralt, A., Teniente, E.: On the Semantics of Operation Contracts in Conceptual Modeling. In: Proc. CAiSE Short Papers 2005, CEUR Workshop Proceedings, 161 (2005)
13. Queralt, A., Teniente, E.: A Platform Independent Model for the Electronic Marketplace Domain. LSI Technical Report, LSI-05-9-R (2005)
14. Sendall, S.: Specifying reactive system behavior. Phd. Thesis. Dir: A. Strohmeier. École Polytechnique Fédérale de Lausanne (2002)
15. Sendall, S., Strohmeier, A.: Using OCL and UML to Specify System Behavior. In: Object Modeling with the OCL, The Rationale behind the Object Constraint Language. Springer-Verlag (2002) 250--280
16. Wieringa, R.: A survey of structured and object-oriented software specification methods and techniques. ACM Computing Surveys 30 (1998) 459-527

Appendix

This appendix provides a list of simple transformation rules between OCL expressions. These transformations help to extend the set of OCL expressions included in the ambiguity patterns of sections 2 and 3. We group the equivalences by the type of expressions they affect. The capital letters *X*, *Y* and *Z* represent arbitrary OCL expressions of the appropriate type. The letter *o* represents an arbitrary object.

Table A.1 List of substitution rules

Type	Rules	
Boolean types	X implies Y → if X then Y else true	(not X or Y) and (X or Z)→ if X then Y else Z
	A xor B → (A or B) and (not A or not B)	not (not A) → A
	not (A or B) → not A and not B	not (A and B) → not A or not B
	A or (B and C)→(A or B) and (A or C)	
Collection Types	X->count(o)>0 → X->includes(o)	X->count(o)=0 → X->excludes(o)
	Y->forAll(y1| X->count(y1)>0) → X->includesAll(Y)	Y->forAll(y1| X->count (y1)=0) → X->excludesAll(Y)
	X->size()=0 → X-> isEmpty()	X->size()>0 → X->notEmpty()
Predef. iterators	X->select(Y)->size()>0 → X->exists(Y)	not X->exists(Y) → X->forAll(not Y)
	X->reject(Y) → X->select(not Y)	X->one(Y) → X->select(Y)->size()=1
	X->select(Y)->size()=0 → X->forAll(not Y)	X->select(Y)->size()=X->size() → X->forAll(Y)
	X->select(Y)->forAll(Z) → X->forAll(Y implies Z)	X->select(Y)->exists(Z) → X->exists(Y and Z)
	not X->forAll(Y) → X->exists(not Y)	