Reflections on the Object Constraint Language

Ali Hamie, Franco Civello , John Howse, Stuart Kent, Richard Mitchell

Distributed Information Systems Research Group,
IT Faculty, University of Brighton, Brighton BN25 3HP, UK.
http://www.biro.brighton.ac.uk/,
a.a.hamie@brighton.ac.uk

Abstract. The object Constraint Language (OCL), which forms part of the UML set of modelling notations, is a precise, textual language for expressing constraints that cannot be shown diagrammatically in UML. This paper reflects on a number of aspects of the syntax and semantics of the OCL, and makes proposals for clarification or extension. Specifically, the paper suggests that: the concept of flattening collections of collections is unnecessary, state models should be connectable to class models, defining object creation should be made more convenient, OCL should be based on a 2-valued logic, set subtraction should be covered more fully, and a "let" feature should be introduced.

1 Introduction

The Object Constraint Language [12] is a precise, textual language designed to complement the largely graphical UML [11]. Specifically, OCL supports the expression of invariants, preconditions and postconditions, allowing the modeller to define precise constraints on the behaviour of a model, without getting embroiled in implementation details.

OCL is the culmination of recent work in object-oriented modelling [1, 2, 3, 8] (which has selected ideas from formal methods to combine with diagrammatic, object-oriented modelling resulting in a more precise, robust and expressive notation. Syntropy [1] extended OMT [13] with a Z-like textual language for adding invariants to class diagrams and annotating transitions on state diagrams with preconditions and postconditions. Catalysis [2, 3] has done something very similar. OCL adopts a simple non-symbolic syntax and restricts itself to a small set of core concepts.

One of the most important aspects of OCL is that it is part of the Unified Modelling Language, which has recently become a standard modelling language, under the auspices of the Object Management Group. As a result, it is likely to get much greater exposure and use than previously proposed formal specification languages such as VDM [9] and Z [14], and work invested in ensuring that it is correct and appropriate for its purpose is therefore more likely to reap a dividend than work on the aforementioned languages.

The purpose of this paper is to contribute to discussions on the correctness and appropriateness of OCL. We identify a number of issues which, in our opinion, need to be resolved; where possible we suggest a solution, or at least an outline direction for further investigation.

The paper is organised as follows. Section 2 deals with navigation in object-oriented modelling, in particular navigating from collections. Section 3 considers object states. Section 4 considers object creation and the feature `allInstances`. Section 5 looks at the issue of undefined values. Section 6 proposes adding more collection operations. Section 7 suggests allowing local definitions. And section 8 briefly summarises the issues examined and proposes that future semantics work on OCL be driven by the needs of CASE tool builders and users.

2 Navigation in OO Modelling

Navigation in OO modelling means following links from one object to locate another object or a collection of objects. It is possible to navigate across many links, and hence to navigate from a collection to a collection. Navigation is at the core of OCL. OCL expressions allow us to write constraints on the behaviour of objects identified by navigating from the object or objects which are the focus of the constraint. At the specification level, the expressions appear in invariants, preconditions and postconditions.

In this section we review some of the issues concerning the meaning of navigation expressions, and outline a semantics for them which takes account of these issues. We conclude by examining what the OCL specification says about navigation expressions and suggest that the notion of flattening collections of collections is not needed.

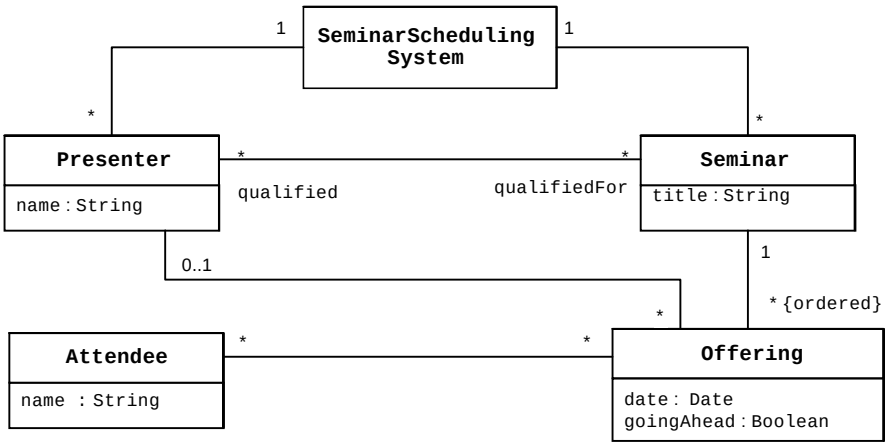

Fig. 1. A class diagram for a seminar scheduling system

2.1 Example Model

Fig. 1 presents a small, contrived example of a class model in UML for a simple system that supports scheduling of offerings of seminars to a collection of attendees by presenters who must be qualified for the seminars they present. A full description of the notation can be found in [11] and a distilled description can be found in [4].

2.2 Navigating from single objects

Navigation expressions start with an object, which can be explicitly declared or given by a context. For example, a declaration such as `s:Seminar` means that `s` is a variable that can refer to an object taken from the set of objects conforming to type `Seminar`. Here, the type name is used to represent the set of objects in the model that conform to the type.

A navigation expression is written using an attribute or role name, and an optional parameter list. Given the earlier declaration, the OCL expression `s.title` represents the value of the attribute `title` for the object represented by `s`. An OCL expression can also use the name `self` to refer to a contextual instance. In the following example, `self` refers to an instance of `Seminar`.

```
Seminar
      self.title
```

Navigating from an object via an association role can result in a single object or a collection, depending on the cardinality annotations of the association role. A collection is, by default, a set. For example, given the declaration `p:Presenter`, the expression `p.qualifiedFor` results in the set of seminars `p` is qualified to present.

The association between `Seminar` and `Offering` has the annotation `{ordered}` on the `offering` role. As a result, the expression `s.offering`, where `s` is a seminar, results in a sequence. Notice that this means that the operator "`.`" is overloaded, because it can map from an object to a set, to a bag, or to a sequence.

2.3 Navigating from collections

Assume we have the declaration `p:Presenter`. The OCL navigation expression `p.qualifiedFor.title` (which is an abbreviation of the following expression `p.qualifiedFor->collect(title)`) involves navigating first from a single object and then from a collection, namely the set of seminars for which presenter `p` is qualified. This is because the expression parses as `(p.qualifiedFor).title`. The result of this expression is obtained by applying `title` to each member of the set `p.qualifiedFor`.

Similarly, navigating from a bag yields a bag and navigating from a sequence yields a sequence (but see section 2.4). This means that every property (attribute or

association role) must, in general, be applicable to a set, a bag or a sequence, and this can be seen in terms of overloading of the navigation operators. For example, within the model of **Fig**. 1, we have the following overloaded versions of the "`_.name`" and "`_.date`" operators (the symbol "_" indicates the position of the argument):

```
_.name: Presenter → String
_.name: Set(Presenter) → Bag(String)
_.name: Bag(Presenter) → Bag(String)
_.date: Offering → Date
_.date: Sequence(Offering) → Sequence(Date)
```

Hence, the OCL expressions `p.name`,`(p.qualifiedFor).name`, `(p.qualifiedFor->asBag).name`, and `(s.Offering).date` are well-typed. The operator `asBag` converts a set or a sequence into a bag.

The overloaded versions of the operator "`_.property`" (`property` is an attribute or association role) must satisfy the axioms:

```
Set{}.property = Bag{}
(s->including(e)).property =
  (s->excluding(e).property)->including(e.property)

Bag{}.property = Bag{}
(b->including(e)).property =
  (b.property)->including(e.property)

Sequence{}.property = Sequence{}
(q->including(e)).property =
  (q.property)->including(e.property)
```

Intuitively, these axioms define that applying `property` to a collection yields a second collection, obtained by applying `property` to each element of the original collection. In the axioms, `s` is a set, `b` is a bag and `q` is a sequence, `e` is some element. Here `e.property` returns a single element; we can give similar axioms for the case where `e.property` returns a collection.

OCL specifies navigation from collections by using the feature `collect`, which takes a collection and an expression as arguments and yields a collection obtained by applying the expression to each element in the collection. When the type of the expression is also a collection then the result can be seen as a collection of collections. According to the OCL documentation, a collection of collections is automatically flattened. Such a view is easy to teach to modellers, but hard to define without falling into traps. For instance, a well-defined function will satisfy

```
x = y implies f(x) = f(y)
```

where `x` and `y` are values and `f` is a function. Consider the following OCL navigation expression.

```
sss.presenter->collect(qualifiedFor)
```

where `sss` is an object of type `SeminarSchedulingSystem`. The first part of the expression

```
sss.presenter
```

yields a set of presenters. The full expression, without flattening, yields a bag of sets of seminars, such as

```
Bag{ Set{s1, s2}, Set{s2, s3} }
```

With flattening, the full expression yields a bag of seminars, such as

```
Bag{ s1, s2, s2, s3 }
```

In the flattening step, no elements are lost or gained (we just lose structure). The two expressions above are of types `Bag(Set(Seminar))` and `Bag(Seminar)`, respectively. Thus, any well-defined function we wish to specify on elements of type `Bag(Seminar)` will not apply to elements of type `Bag(Set(Seminar))`, unless we specify it in various overloaded forms. There would be as many overloaded forms as there are possible levels of structure in the model.

If, instead, OCL defined the result of navigating via collections simply in terms of left-to-right parsing, there would be no need for any concept of flattening. For instance,

```
sss.presenter.qualifiedFor.offering
```

is parsed as

```
( (sss.presenter).qualifiedFor) ).offering
```

whose meaning can be found by repeated application of navigation from one collection to another. Each application of navigation yields a collection, which is the source of the next navigation. This does not entail building a collection of collections of collections and then flattening it.

2.4 Navigating from sequences

According to the OCL document, navigating from a sequence yields another sequence. For example, given the declaration `s:Seminar`, the expression `s.offering` results in the sequence of offerings for seminar `s`. The expression `s.offering.attendee` results in the sequence of attendees for all offerings of seminar `s`. The value of this expression is obtained by applying the association role `attendee` to each element of the sequence `s.offering`. This results in a sequence of sets which is then flattened to give the desired sequence. However, there are many ways to flatten sequence of sets, which would result in different sequences.

OCL does not indicate how such collections of collections are flattened. In addition, there are situations where it is not appropriate to get a sequence when navigating from a sequence. For example, given a seminar `s` we would be more interested in the bag of all attendees for all offerings of `s` rather than in the (underspecified) sequence.

3 States

In object-oriented modelling, class diagrams can be supplemented by state diagrams. A state diagram for a given object type shows the possible states an object of this type can be in, together with the transitions that move an object from one state to another. A state diagram contributes to the behavioural specification of a type in a model. An object state is an abstraction of its detailed property values. **Fig**. 2 shows a state diagram of `Offering` with two states, `Scheduled` and `Cancelled`, meaning that an offering of a seminar can be scheduled or cancelled but not both. There are several ways of connecting class diagrams and state diagrams. One approach is taken by Syntropy [1], which amounts to treating states as dynamic subtypes, so that an object can move from one type to another. A second approach is to treat states as if they were boolean attributes in class diagrams. In UML it is not clear how to connect class diagrams and state diagrams, and OCL does not clarify the issue.

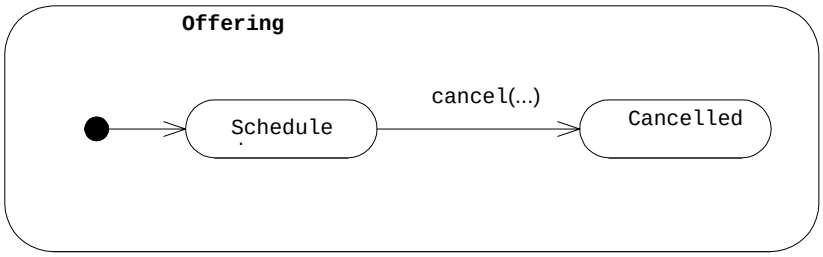

Fig. 2. A state diagram for a seminar offering

If UML allows states to be represented as dynamic subtypes on a class diagram then the OCL feature `oclIsKindOf` can be used to assert that an object is in a given state. For example, we could use `o.oclIsKindOf(Scheduled)` to assert that offering `o` is in the state `Scheduled`.

If states are represented as boolean attributes then the corresponding attributes could be used to represent states in OCL. For example the expression `p.Scheduled` would be true if `p` is in state `Scheduled`, and false otherwise. These state-model attributes can related to other properties by means of invariants. For example, the state `Cancelled` in **Fig**. 2 can be related to the attribute `goingAhead` in **Fig**. 1 by an obvious invariant.

Yet another way would be to introduce a function:

```
_in_:Presentation,State → Boolean
```

where (`p in Scheduled`) is true if `p` is in state `Scheduled`, and false otherwise, and where `State` would be an enumerated type of object states.

From the point of view of using OCL, the mapping to boolean attributes is, perhaps, the easiest to explain to modellers. However, from the point of view of providing an integrated semantics for UML, treating states as dynamic types might be the most elegant approach: substating then has the same semantics as inheritance, dynamic classes in class diagrams are just states in state diagrams, there can be associations targeted and sourced on states (dynamic classes), and so on.[1] Whichever approach is chosen, it should be clear to modellers how the names of states can be defined in terms of class model properties, and how they can be used in OCL expressions.

4 Object creation

OCL provides a type operation `allInstances`, which delivers a set of all instances of a given type. For example, `Presentation.allInstances` would be a set of all instances of type `Presentation` in the model *at a given point in time*. Although the italicised condition is not explicitly covered in the OCL documentation, it has been inferred from a private communication on object creation with Jos Warmer, one of the authors of the OCL. In general, for a given type `T`, the meaning of `T.allInstances` is the set of all elements of type `T` at some moment in the life of a model containing type `T`.

The set `T.allInstances` can change as a result of creation operations associated with the type `T`. One use of `allInstances` is in the postcondition of an operation specification to assert that an object has been created. In the example system, one result of executing an operation `schedule` is the creation of a new offering. In order to assert that a new offering `o` is created, we need to assert that it did not exist prior to executing the operation but does exist after executing the operation. We can use the `allInstances` operation, as follows:

```
(Offering.allInstances - Offering.allInstances@pre)
                              -> includes(o)
```

where `Offering.allInstances@pre` is the set of offerings that existed in the model prior to executing `schedule`. Asserting that a new object has been created is such a common thing to do that we propose the introduction of a limited number of convenient abbreviations. Here are two candidates.

---

[111] Note that this semantics is not necessarily in accordance with the semantics of state diagrams as currently described in the UML 1.1. documentation. Discussion of the relationship between these two approaches appears in [10].

```
SeminarSchedulingSystem::schedule( s:Seminar, d : Date )

post:
      self.seminar.offering->exists( o : Offering |
         Offering.allInstances-Offering.allInstances@pre
                  ->includes( o )
         and  o.seminar = s
         and  o.date = d
         and  o.attendee->isEmpty
         and  o.presenter->isEmpty
         and  o.goingAhead)
```

Fig. 3. Specification of operation `schedule`

The first recognises that asserting creation in a postcondition often involves saying "there is a new object o of type `T` and it has the following properties." For example, in the model of **Fig**. 1, the postcondition of an operation to schedule a new presentation of a seminar is given in **Fig**. 3.

Loosely, this begins by saying that after the schedule operation there exists an offering which was not in the set of offerings before the operation, and continues by defining four properties of the new offering (seminar, date, attendee and presenter). This is such a common idiom that a combined operator to assert existence and newness would be useful, as in **Fig**. 4.

```
SeminarSchedulingSystem::schedule( s:Seminar, d : Date )

post:
      self.seminar.offering->existsNew( o : Offering |
         and  o.seminar = s
         and  o.date = d
         and  o.attendee->isEmpty
         and  o.presenter->isEmpty
         and  o.goingAhead)
```

Fig. 4. Alternative specification of operation `schedule`

Now the newness is captured in the operator and the body of the quantified expression concentrates on defining what properties the new object should have.

Our second candidate for a convenient operator associated with creation is inspired by the `allInstances` operator. An operator `newInstances`, as in, for example,

```
Offering.newInstances
```

could be used in postconditions to mean exactly those instances of type `Offering` that did not exist in the pre-state. The Catalysis method [3] has something similar. We see no harm in having several overlapping ways to talk about new objects.

5 Undefined Values

The OCL document [12] (p7) admits the possibility that some expressions may be undefined when evaluated. Having an undefined value could be important for a number of purposes. It could serve as the result of an illegal operation such as dividing by zero; or as indicated in the OCL definition (p15) when asking for the property of an object that has been destroyed in the post-condition of an operation; or for the `@pre` property of one that has just been created; or when type casting (p6). In addition, an undefined value could be used to stand for a non-terminating computation such as an infinite loop.

Several approaches have been used in other languages to deal with undefined expressions. One approach is to regard undefined expressions as being unknown or underspecified. In this case the result of, for instance, dividing 1 by 0 is an integer but its value is unknown. This is similar to declaring a variable of a given type: the variable has a value of the declared type, but the precise value is unknown. In this approach, Boolean expressions are either true or false, resulting in a two-valued logical system. It is the approach generally adopted in classical mathematics, which admits only total functions, and in some formal specification languages, such as the Larch Shared Language [5].

Another approach is to include a special value ⊥ to denote that something is undefined. If the logical connectives are treated as boolean functions then the undefined value propagates into logical expressions. For example, `b and` ⊥ `=` ⊥. This results in a 3-valued logic, as in, for instance, VDM.

Yet another approach, adopted by Z, is to maintain the distinction between logical operators and expressions. Undefined expressions are interpreted as meaningless, that is, they do not denote anything in the interpretation domain. Since logical expressions are not treated as expressions within the language, their truth values are unknown if they involve undefined expressions.

In OCL expressions can be undefined. However, it is not clear from the documentation what is meant by being undefined. One possibility is that undefined is *not* interpreted as unknown. Let ⊥ stand for the undefined value. According to OCL, if a subexpression of an expression evaluates to undefined then the whole expression is undefined. The only exceptions to this are:

```
True or ⊥ =True
⊥ or True = True
False and ⊥ = False
⊥ and False = False
```

that is, `True` OR-ed with anything is `True`, and `False` AND-ed with anything is `False`. With other Boolean operations we deduce the following:

```
False implies ⊥ = True
⊥ implies True = True
not(⊥) = ⊥
```

The boolean operations agree with the classical logical connectives on the ordinary truth values, i.e., `True` and `False`. However, when ⊥ is involved they reflect a model of computation which is mainly strict. For example, with the operation `not`, if the argument is undefined then whole expression is undefined, that is to say `not` is strict in its argument. The operation `or`, however, is not strict in either the first or the second argument.

In addition we have the following axiom:

```
⊥ or ⊥ = ⊥
```

which implies the law of excluded middle does not always hold, that is, a Boolean expression can be true, false or undefined. (From the definition of `b implies b2`, i.e., `(not b) or (b and b2)`, given on p24 of the OCL document, we could deduce that `⊥ implies True = ⊥`, which is not consistent with either 2-valued or 3-valued logic. However, this definition is probably erroneous and should have been `(not b) or b2`.)

There is one place in OCL where undefinedness definitely is not required: when navigating over an optional association (cardinality `0..1`). By forcing the result of navigation to be a set, the equivalent of a 'null' or 'nil' reference is the empty set (and similarly for optional attributes). Thus 'null' does not correspond to an undefined value.

Both 2-valued and 3-valued logics have advantages. However, we would suggest that OCL be based on a 2-valued logic, for the following reasons. If the logic is to be used for specifying properties without reasoning about partial functions, 2-valued logic seems appropriate and simpler. In addition, reasoning with 3-valued logic is harder because of the absence of some logical laws, e.g., the law of excluded middle. We would suggest that an understanding of 3-valued logic is not required by users, so perhaps references to 3-valued logic are an unnecessary complication if practitioners are the audience.

6 Completing the set of collection operators

In its current form, the OCL contains an `includes` operation, as in `p.qualifiedFor->includes(s)`, which says that seminar `s` is an element of the set `p.qualifiedFor` (the set of seminars p is qualified to present), but there is no `p.qualifiedFor->excludes(s)`. Perhaps more importantly, there is `p.qualifiedFor->includesAll(p1.qualifiedFor)`, saying that the set `p1.qualifiedFor` is a subset of `p.qualifiedFor`, but no `p.qualifiedFor->excludesAll(p1.qualifiedFor)`. Instead the latter has to be expressed using the rather cumbersome expression:

```
(p.qualifiedFor->intersection(p1.qualifiedFor))
                              ->isEmpty
```

There is, however, an operation `p.qualifiedFor->excluding(s)`, and the set subtraction operator "-" found in traditional mathematical notation. We suggest

that the set of operations on collections could be extended so that the inclusive operators all have their exclusive counterparts.

7 Local definitions

In VDM [9], "let" expressions have the following syntax:

```
let(x = expr : oclExpression) in
       (expr1 : oclExpression) end
                                  :expr1.evaluationType
```

The value of a let expression is evaluated by evaluating expression `expr` and then using the result in the evaluation of expr1. This is equivalent to `expr1[expr/x]` (the expression `expr1` with `x` substituted for `expr`).

Let expressions are useful when the same expression needs to be used a number of times in the same assertion. This is particularly true when long navigation expressions are combined with operators on collections to identify particular sets of objects. Then having to repeat such expressions several times is cumbersome, and can obscure the meaning of the overall assertion. We therefore recommend that some form of local definition mechanism be included.

8 Further work

In this paper we have considered some issues related to the OCL language. We believe that the ideas we have presented about navigation should be tested by including them in a proper formal semantics for OCL.

With regard to object states, we have commented on the fact that there is a problem in UML with the integration of state and class diagrams, and no attempt has been made to resolve this in OCL. We have sketched some approaches to providing an integrated semantics. However, there is semantic work to be done here, too. For instance, the approach based on dynamic subtypes is at odds with the (informally described) semantics provided as part of the UML 1.1. In particular, it takes no account of events and requires the restriction that all transitions must be atomic and at the same level of granularity to be lifted. We believe that work in this area is crucial if UML is to proceed any further, especially when one considers that UML-RT (Real Time) is likely to provide us with yet another possible semantics for state diagrams and, at least initially, seems to be taking a "bolt on" rather than "integrative" approach.

In general, the integration of the UML notation set, including OCL, needs attention.

We have highlighted a range of approaches in the formal methods literature for dealing with undefinedness. We do not believe this issue can be resolved without providing a formal semantics for OCL, and the way it is resolved will depend on the semantics approach taken. We believe that a semantics should be built for a purpose,

which in our view should be to support CASE tools for reasoning about and checking the integrity of models specified using UML and OCL.

Acknowledgements

This work was supported by funds from the UK EPSRC under grant number GR/K67304.

References

1. Cook, S., Daniels, J.: Designing Object Systems: Object-Oriented Modelling with Syntropy. Prentice Hall, UK (1994)
2. D'Souza, D., Wills, A.: Extending Fusion: practical rigor and refinement. R. Malan et al., OO Development at Work, Prentice Hall (1996)
3. D'Souza, D., Wills, A.: Objects, Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, to appear 1998. Draft and other related material available http://www.trireme.com/catalysis
4. Fowler, M., Scott, K.: UML Distilled. Addison-Wesley (1997)
5. Guttag, J., Horning, J.: Larch: Languages and Tools for Formal Specifications. Springer-Verlag (1993)
6. Hamie, A., Howse, J., Kent, S.: Navigation Expressions in Object-Oriented Modelling. Lecture Notes in Computer Science, Vol. 1382. Springer-Verlag (1998) 123-137
7. Hamie, A., Howse, J., Kent, S.: Compositional Semantics for Object-Oriented Models. In Duke, D. and Evans A., editors, 3rd Northern Formal Methods Workshop, electronic Workshops in Computing, UK, Springer-Verlag (1998)
8. Meyer, B.: Eiffel the Language. Prentice Hall (1992)
9. Jones, B. C.: Systematic Software Development using VDM. Prentice Hall (1990)
10. Kent, S.: UML: What does it all mean? 1 day tutorial at ETAPS'98, Lisbon, Portugal. Notes available from http://www.it.brighton.ac.uk/staff/Stuart.Kent (1998)
11. Rational Software Corporation: The Unified Modeling Language Version 1.1. Available from http://www.rational.com (1997)
12. Rational Software Corporation: The Object Constraint Language Specification, Version 1.1. Available from http://www.rational.com (1997)
13. Rumbaugh, J., Blaha, M., Premerali, W., Eddy, F, Lorensen, W.: Object-Oriented Modelling and Design. Prentice Hall (1991)
14. Spivey, M.: The Z Notation. 2nd ed. Prentice Hall, UK (1992)