Interpreting the Object Constraint Language

Ali Hamie, John Howse and Stuart Kent
Division of Computing,
University of Brighton, Lewes Rd., Brighton, UK.
e-mail: A.A.Hamie@brighton.ac.uk

Abstract

The Object Constraint Language (OCL), which forms part of the UML 1.1. set of modelling notations is a precise, textual language for expressing constraints that cannot be shown in the standard diagrammatic notation used in UML. A semantics for OCL lays the foundation for building CASE tools that support integrity checking of whole UML models, not just the component expressed using OCL. This paper provides a semantics for OCL, at the same time providing a semantics for classes, associations, attributes and states.

1: Introduction

The Object Constraint Language [1][2] is a precise textual language to complement graphical languages in modelling object-oriented systems. It allows constraints on the model to be expressed, that can not be expressed using standard diagrammatic notations. Specifically, OCL supports the expression of invariants and pre/post conditions, allowing the modeller to specify precise and detailed constraints on the behaviour of a model, without getting embroiled in implementation detail.

OCL is the culmination of recent work in OO modelling [3][4] which has selected ideas from formal methods to combine with diagrammatic, object-oriented modelling resulting in a more precise, robust and expressive notation. Syntropy [3] extended OMT [5] with a Z-like textual language for adding invariants to class diagrams and annotating transitions on state diagrams with pre/post conditions. OCL adopts a simple non-symbolic syntax and restricts itself to a small set of core of concepts.

One of the most important aspects of OCL is that it is part of the Unified Modelling Language [6], which has recently become *the* global 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 [7] and Z [8], 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 provide a semantics to check that OCL is unambiguous and to improve OCL [9]. As OCL does not exist in a vacuum, but instead depends on some parts of a model to be defined already in diagrams, this necessitates a semantics for a kernel of the UML diagrammatic notation, specifically: class diagrams. Thus OCL provides a focus for integrating the semantics of the diagrammatic notations. We present one such integrated semantics, which we hope lays the foundation for building CASE tools that support integrity checking of whole UML models, not just the component expressed using OCL.

Semantics work [10][11] for OO modelling notations in widespread use, such as OMT or UML, is generally restricted to capturing the meaning of those notations, so accompanying precise textual languages have yet to be considered, as languages such as OCL have only very recently been incorporated. Exceptions to this are the work of Bicarregui et al. [12][13] which uses the Object Calculus [14] to develop a semantics for Syntropy [3], and our own work [15][16][17][18][19].

We have chosen to use Larch [20]. This choice is motivated in part by the desire not to be engaged in the design of logics and reasoning systems, but instead to focus on elaborating the meaning of the modelling notations themselves. Larch is a stable language with a well-developed supporting toolset. It uses first-order predicate logic, rather than temporal logic, so is accessible to a wider audience, which includes, hopefully, some commercial tool developers. It is also close to technologies most likely to leverage the sophisticated CASE tools that should result from increasing the precision and expressiveness of model-

ling notations. This is illustrated by the inclusion of an automated proof assistant in its accompanying toolset. We have also been using our Larch-based semantics as a starting and subsequent reference point for developing checking and animation tools in Prolog.

The paper is organised as follows. Section 2 is an informal introduction of using OCL, in combination with a kernel of the UML diagrammatic notation (class diagrams), in writing navigation expressions in object-oriented modelling. Section 3 establishes the semantic framework by giving a semantics to class diagrams. Section 4 defines the semantics of OCL expressions. A key aspect of this is a semantics for navigation expressions, including navigation over collections other than sets, such as sequences and bags, and the semantics of filters. Section 5 deals with system states. Section 6 summarises the semantics of invariants and pre/post conditions (expressed in OCL). Section 7 summarises the general mapping of the UML/OCL elements considered. Section 8 concludes with an overview of future work in semantics and elsewhere.

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.

2.1: Example model

Figure 1 presents a small, contrived example of a class diagram 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 [6][21].

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 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

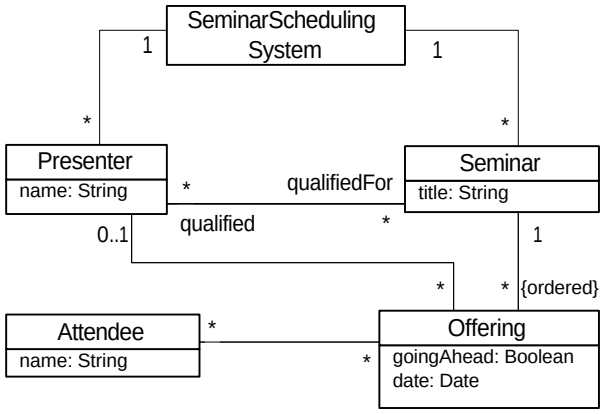

Figure 1.Class diagram for a seminar scheduling system

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. For example, given the declaration p : Presenter, the expression p.qualifiedFor results in a 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 an object to a set, to a bag, or to a sequence.

2.3: Navigating from collections

Assume we have the following declaration p:Presenter. The navigation expression p.qualifiedFor.title (also written p.qualifiedFor->collect(title) in OCL) 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 a bag obtained by applying title to each member of the set p.qualifiedFor. The OCL operation asSet can be used to convert this bag to a set. Similarly, navigating from a bag yields a bag and navigating from a sequence yields a sequence.

2.4: Invariants

Navigation expressions can be part of an invariant on a type which must be true for all instances of that type at any time. For example, an invariant for the seminar scheduling system would be:

```
Presenter
self.qualifiedFor->includesAll(self.offering.seminar)
```

which says that a presenter must be qualified for all seminars he/she is assigned to present.

2.5: Preconditions and postconditions

An OCL expression can also be used as a precondition or postcondition, which are used to specify the behaviour of an operation or method. The name self can also be used in the expression referring to the object on which the operation was called. Expressions occurring in a postcondition can refer to two sets of values for each property of an object:

- the value of a property at the start of the operation or method
- the value of a property upon completion of the operation or method

In OCL the value of a property at the start of the operation is denoted by postfixing the property name with the commercial sign *@*, followed by the keyword *pre*.

Figure 2 gives the specification of an operation markAsAbsent in terms of *pre*/*post* conditions. This operation marks a presenter as absent by cancelling his/her presentations within specific dates.

```
SeminarSchedulingSystem
markAsAbsent(p : Presenter, from, to : Date)
pre:   true
post:  p.offering@pre->forAll( o |
         o.date >= from and o.date <= to  implies
                        o.presenter = Set{})
```

Figure 2.Specification of operation markAsAbsent

3: Semantics: Class Diagrams

We shall use the *Larch Shared Language* (LSL) [20] to provide the semantics of OCL expressions. This is achieved by first providing the semantics of object types, attributes, and associations. LSL uses specification modules, called *traits*, to describe abstract data types and theories.

3.1: Object types

An object type is a description of a set of objects in terms of properties and behaviour they all share. In our formalisation, an object type is associated with an LSL basic sort consisting of elements that uniquely represent objects (instances) of the type, which can be thought of as object identifiers. The attributes of an object type are formalised as functions on the sort representing this type.

The object type Presenter in Figure 1 is interpreted as a basic sort denoted by `Presenter`, namely a sort of presenter identifiers. The attribute name is interpreted as a function `name` with signature $\texttt{name : Presenter} \rightarrow \texttt{String}$, which is added to the specification for object type Presenter. The type String is interpreted as the sort of strings `String` which is available in the Larch HandBook of specification modules [20]. In a very similar way we interpret the other object types for the seminar scheduling system.

3.2: Associations

We now extend the interpretation of object types and attributes given in the previous section to include binary associations. Associations are basically relationships between objects. Each association in a class diagram has two role names which can be used to navigate the association from a specific object to refer to other objects and their properties. For instance, the association between Presenter and Seminar (Figure 1) has two role names qualified and qualifiedFor.

We formalise associations between object types as two related functions that map an object of one type to the set of associated objects of another (or the same) type. These mappings are specified in a way that is independent of the structure of types they associate. Thus we have a generic Larch theory for associations that can be renamed to specify each particular association in the model. For example, the association between Presenter and Seminar would be represented as two functions `qualified` and `qualifiedFor` with the signatures:

$$\texttt{qualified} : \texttt{Set[Seminar]} \rightarrow \texttt{Set[Presenter]}$$

$$\texttt{qualifiedFor} : \texttt{Set[Presenter]} \rightarrow \texttt{Set[Seminar]}$$

where `Set[Seminar]` and `Set[Presenter]` are the power sorts of `Seminar` and `Presenter` respectively. By choosing power sorts for the domains and ranges of these mappings, we have a uniform treatment of associations which simplifies the formalisation and provides generic theory for associations. In the case of an optional association (0..1 cardinality), this is especially useful to

check whether there is an object or not when navigating the association, namely whether the resulting set is empty or not. The case where navigation is from a single object is subsumed with the general case where the set is a singleton containing that object. The corresponding functions that map single objects can be defined in terms of those that map sets of objects (see later).

The two functions `qualified` and `qualifiedFor` satisfy the axioms:

```
qualified({})= {}

qualifiedFor({})= {}

qualified(s ∪ s') ==
   qualified(s) ∪ qualified(s')

qualifiedFor(s ∪ s') ==
  qualifiedFor(s) ∪ qualifiedFor(s')
```

The operation ∪ is the union operation on sets. Note that these axioms imply that these functions are completely determined by their values at singleton sets.

In order to represent the association, these functions are related by the following axiom:

```
s ∈ qualifiedFor({p}) == p ∈ qualified({s})
```

Intuitively, this axiom asserts that if instructor p is qualified to present seminar s, then p must be included in the set of presenters qualified to present s.

The corresponding functions that operate on single objects may be constructed from those whose domains are power sorts as follows:

```
qualified(s) == qualified({s})

qualifiedFor(p) == qualifiedFor({p})
```

Semantically, navigating from a single object is equivalent to navigating from a singleton set containing that object.

For further details and for the generic traits of object types and associations the reader is referred to [15][17].

4: Collections and their operations

Collection as defined in OCL is an abstract type, with concrete collection types as its subtypes; Set, Sequence, and Bag. This type is not strictly necessary since it is defined as an abstract supertype.

The Collection type can be specified in LSL by a sort `Collection` and including the signatures of the common operations shared between its subtypes and some of their axioms. Let `Collection[T]` be the sort of collections of type `T`. For example, the size operation which common to all collection types has the signature `size:Collection[T]→Integer` and is specified in terms of the operation iterate. The collection types (Set, Bag, and Sequence) can be specified in LSL as abstract data types with the familiar operations. For example, LSL provides traits (available in the Larch HandBook) for specifying these mathematical abstractions. The additional operations provided by OCL will be dealt with in the next subsection.

The Larch Shared Language does not support subsorting. So in order to assert that `Set[T]` is a subtype of `Collection[T]` we use the function `toCollection : Set[T] → Collection[T]` that maps a set into a collection representing it. The assertion that Bag[T] and Sequence[T] are subtypes of Collection[T] can be handled in a similar way by overloading the function `toCollection`. For bags we have `toCollection : Bag[T] → Collection[T]`. The `size` operation has to be specified on `Set[T]` by including the signature `size : Set[T] → Integer` together with the axiom:

```
size(s)= size(toCollection(s))
```

similarly, we specify `size` for bags and sequences.

There are many operations defined on collection types in OCL. These operations transform existing collections into new ones. Here we consider the more interesting ones, namely select, reject, collect, forAll, exists and iterate.

4.1: Select and reject operations

The select and reject operations provide a way of specifying a subset of a collection. A select is an operation on a collection and is specified using the ->-syntax:

```
collection -> select( v : T | b-expr-v)
```

where the variable v is called the iterator and b-expr-v is a boolean expression. This expression is evaluated by using v to iterate over collection and evaluating b-expr-v for each v. The v is a reference that refers to the objects from the collection.

The meaning of select expressions can be obtained by defining two function $\texttt{select}_p$ and p with the signatures $\texttt{select}_p : \texttt{Collection[T]} \rightarrow \texttt{Collection[T]}$ and $p : T \rightarrow \texttt{Bool}$ respectively. The function p is defined as `p(v) == b-expr-v`. That is each boolean expression induces a function. The select operation applied to a set always results in a set, and the same applies for bags and sequences. Hence, the meaning of the operation has to be specified for sets, bags and sequences.

For sets we define a function $\texttt{select}_p$ with the signature $\texttt{select}_p : \texttt{Set[T]} \rightarrow \texttt{Set[T]}$ and satisfies the axioms:

$Select_p(\{\}) == \{\}$
$select_p(insert(v, s)) == \texttt{if } p(v) \texttt{ then } insert(v, select_p(s)) \texttt{ else } select_p(s)$

where p is a boolean function defined as above. In addition we have the axiom:

$toCollection(select_p(s)) = select_p(toCollection(s))$

For example, p.offering->select(goingAhead) is interpreted as $select_{goingAhead}(offering(p))$, where goingAhead is interpreted as a function with the signature $goingAhead : Offering \rightarrow Bool$.

For bags and sequences similar functions can be defined with similar axioms, the only difference is the signatures of these functions.

The reject operation is similar to the select operation, but with reject we get the subset of all the elements for which the boolean expression evaluates to False. In fact reject can be interpreted in terms of select because the expression collection->reject(v:T | b-expr-v) is equivalent to collection->select(v : T | not(b-expr-v)).

4.2: Collect operation

The select and reject operations always yield a sub-collection of the original one. However, it is often required to specify a collection which is derived from some other collection, but which contains different objects from the original collection. The collect operation provides such construct in OCL. The syntax of collect is written as follows:

collection -> collect(v : T | expr-v)

The value of the collect operation is the collection of the results of all the evaluations of expr-v.

The meaning of collect expressions can be obtained by defining two functions $collect_f$ and f with the signatures

$collect_f : Collection[T] \rightarrow Collection[S]$
$f : T \rightarrow S$

respectively. The function f is defined as $f(v)$ = expr-v .

In OCL the result of the collect operation on a set is a bag rather than a set. So we define $collect_f : Set[T] \rightarrow Bag[S]$ which satisfies the axioms:

$collect_f(\{\}) == \{\}$
$collect_f(insert(v, s)) == insert(f(v), collect_f(s))$
$toCollection(collect_p(s)) == (collect_p(toCollection(s)))$

However, if it is required that collect on a set should result in a set rather than a bag, then we can make a set from the bag by using the function $asSet : Bag[T] \rightarrow Set[T]$ which satisfies the axioms:

$asSet(\{\}) == \{\}$
$asSet(insert(v, b)) == insert(v, asSet(b))$

For bags we define $collect_f : Bag[T] \rightarrow Bag[S]$ which satisfies similar axioms as the one for sets. For sequences, we define $collect_f : Seq[T] \rightarrow Seq[S]$ which also satisfies similar axioms as the one for sets, where the only difference is that the result is a sequence.

4.3: Navigation expressions

In OO modelling navigating from a collection of objects is very common. For this reason OCL provides a shorthand notation for the operation collect. Instead of writing self.qualifiedFor->collect(title) we can write self.qualifiedFor.title. In OCL applying a property to a collection of objects is interpreted as a collect over the members of the collection with the specified property.

So, for any propertyname of objects in a collection, the following expressions are identical

collection.propertyname

collection->collect(propertyname)

In OCL, a collection of collections is automatically flattened. Such a view is easy to teach to modellers, but hard to define without falling into traps. In related work we have shown that flattening is not necessary. More information about this can be found in [9].

4.4: Quantifications

OCL provides two operations for quantifications forAll and exists operations. The forAll operation in OCL allows the specification of a boolean expression, which must hold for all objects in a collection. Its syntax is given by:

collection -> forAll(v : T | b-expr-v)

The value of a forAll expression is boolean. The result is true if the boolean expression b-expr-v is true for all elements of collection. The result is false if b-expr-v evaluates to false for one or more v in collection.

The semantics of forAll can be given by using LSL universal quantification denoted by $\forall$. So the forAll expression is interpreted as:

$(\forall v : T)(v \in collection \Rightarrow b\text{-}expr\text{-}v)$

The semantics of exists can be given in a similar way.

4.5: Iterate operation

OCL also has the iterate operation which is very generic in the sense that the operations select, reject, forall, exists, and collect can all be described in terms of iterate. The syntax of iterate is:

collection->iterate(v : T; acc:S = expr | expr-v-acc)

The iterate operation is evaluated by using v to iterate over the collection and the expr-v-acc is evaluated for each v. After each evaluation of expr-v-acc, its value is assigned to acc. In this way the value of acc is built up during the iteration of the collection.

The meaning of iterate expressions can be obtained by defining $\texttt{iterate}_{\texttt{f,expr}} : \texttt{Collection[T]} \to \texttt{S}$ and $\texttt{f} : \texttt{T,S} \to \texttt{S}$, where $\texttt{f(v, acc)} == \texttt{expr-v-acc}$. For sets the function $\texttt{iterate}_{\texttt{f,expr}}$ satisfies the axioms:

$$\texttt{iterate}_{\texttt{f,expr}}(\{\}) == \texttt{expr}$$
$$\texttt{iterate}_{\texttt{f,expr}}(\texttt{insert(v, s)}) == \texttt{f(v, iterate}_{\texttt{f,expr}}\texttt{(delete(v, s)))}$$

where delete is the operation for removing an element from a set. These axioms are only valid for functions f that satisfy the properties:

$$\texttt{f(x, f(y, z))} = \texttt{f(y, f(x, z))}$$

Without this axiom we can have two equal sets s1 and s2 where iterate(s1) is not equal to iterate(s2). This is clearly not consistent with the notion of substituting equals for equals. So if f does not satisfy these properties, the operation iterate is not deterministic. For bags and sequences we can define similar functions which satisfy similar axioms.

5: System state

So far we have ignored system state, which is required in the presence of dynamic behaviour, as specified for example through preconditions and postconditions on operations. Thus we enrich the semantic model with a sort of system state Σ. Given this, we introduce, for each object type T, a function $\texttt{T} : \Sigma \to \texttt{Set[T]}$ which returns the set of existing objects of type T (i.e. those that have been created and not destroyed) in a given state σ. This function is used to interpret the allInstances feature of an object type, which returns the set of all instances of the type. For example, T.allInstances is interpreted as the set $\overline{\texttt{T}}(\sigma)$.

Attributes of a given object type are interpreted as functions with additional argument for the system states. For example, the attribute title is now interpreted as a function $\texttt{title} : \texttt{Seminar}, \Sigma \to \texttt{String}$. And similarly for associations.

6: Invariants, preconditions and postconditions

We interpret invariants by interpreting each expression occurring in it and adding universal quantifications. For example, for the invariant given earlier, the self.qualifiedFor is interpreted as $\texttt{qualifiedFor(self}, \sigma)$, and self.offering.seminar is interpreted as $\texttt{seminar(offering(self}, \sigma), \sigma)$. So the invariant is interpreted as the following assertion:

$$\forall \texttt{p} : \texttt{Presenter}, \sigma : \Sigma \; (\texttt{seminar(offering(self}, \sigma), \sigma)) \subseteq \texttt{qualifiedFor(self}, \sigma)$$

That is, the invariant must hold in every system state σ.

We interpret pre/post conditions by interpreting each expression occurring in them. For example, the postcondition of the operation markAsAbsent is interpreted by interpreting p.seminar@pre as $\texttt{seminar(p}, \sigma)$, and the predicate part of forAll as $\texttt{pred(o}, \sigma) == \texttt{date(o}, \sigma) \geq \texttt{from} \wedge \texttt{date(o}, \sigma) \leq \texttt{to}$. The whole postcondition is interpreted as:

$$(\forall(\texttt{o} : \texttt{Offering}, \sigma, \sigma' : \Sigma))(\texttt{o} \in \overline{\texttt{Offering}}(\sigma) \wedge \texttt{pred(o}, \sigma) \Rightarrow \texttt{presenter(p}, \sigma') = \{\})$$

where σ and σ' are the states before and after the operation is executed respectively.

7: Summary of mapping

In this section we summarise the mappings between OCL types and expressions and the sorts and expressions of LSL.

For each type A in a class diagram we associate with it a sort of all possible object identities that conform to the type, denoted by A. We define a mapping $\tau : \texttt{OclType} \to \texttt{lslSort}$ by $\tau(\texttt{A}) = \texttt{A}$. That is, τ(A) is the sort associated with the type A. For example, we have τ(Seminar) $=_{\text{def}}$ Seminar.

The basic value types in OCL are mapped directly to predefined sorts in LSL. The type Boolean is mapped to the sort Bool, i.e. τ(Boolean) $=_{\text{def}}$ Bool, which is specified as a trait in the Larch HandBook of specifications. The type Integer is also mapped to a predefined sort in LSL namely Integer, i.e τ(Integer) $=_{\text{def}}$ Integer. Similarly we have τ(String) $=_{\text{def}}$ String. Enumerated types are also mapped directly to LSL enumerated sorts.

For collection types such as sets, bags and sequences, LSL provides basic traits that specify basic operations on these structures. However, these traits need to be extended to deal with new operations available in OCL. These traits can be constructed by including the traits that specify operations like select, reject, iterate, etc.. For the moment we map the types Set(T), Bag(T), and Sequence(T) to the sorts `Set[T]`, `Bag[T]`, and `Seq[T]` respectively.

For each attribute att of type T of an object type A we associate with it a function symbol denoted by `att`. For this we define a mapping α_A which maps an attribute symbol to a function symbol in LSL, by $\alpha_A(\text{att:T}) =_{\text{def}}$ $\texttt{att} : A, \Sigma \rightarrow T$. Operations or queries on type A such as op(S):T of type T, are mapped as: $\alpha_A(\text{op(S):T}) =_{\text{def}}$ $\texttt{op} : A, S, \Sigma \rightarrow T$.

OCL	LSL
A (object type)	`A` (sort of object identities)
T (value type)	`T` (sort of values)
Boolean	`Bool`
String	`String`
Integer	`Integer`
Collection(T)	`Collection[T]`
Set(T)	`Set[T]`
Bag(T)	`Bag[T]`
Sequence(T)	`Seq[T]`
att:T (attribute)	$\texttt{att} : A, \Sigma \rightarrow T$
op(S) :T	$\texttt{op} : A, S, \Sigma \rightarrow T$
r : set[B] (role)	$\texttt{r} : \texttt{Set[A]}, \Sigma \rightarrow \texttt{Set[B]}$
r (S) : set[B] (role)	$\texttt{r} : \texttt{Set[A]}, S, \Sigma \rightarrow \texttt{Set[B]}$
r : B (role)	$\texttt{r} : \texttt{Set[A]}, \Sigma \rightarrow \texttt{Set[B]}$
r (S) : B (role)	$\texttt{r} : \texttt{Set[A]}, S, \Sigma \rightarrow \texttt{Set[B]}$

Figure 3. Mappings of types, attributes and associations

For each association role r (at the right) of an association between two types A and B we associate a function symbol in LSL. For this we define a mapping ρ_A by $\rho_A(\text{r:set[B]}) =_{\text{def}}$ $\texttt{r} : \texttt{set[A]}, \Sigma \rightarrow \texttt{set[B]}$. Parameterised (qualified) association roles are dealt with in a similar way. The table in Figure 3 summarizes the above mappings.

We now define a mapping μ with signature

$$\mu : \texttt{OclExpression} \rightarrow \texttt{lslExpression},$$

that maps OCL expressions to LSL expressions based on the above mappings. The definition of μ is given in Figure 4. The interpretation of an OCL expression as given by μ is given at a moment in time corresponding to a system state σ. In this definition, variables in OCL are mapped into variables in LSL, i.e. $\mu(v) = v$. Expressions of the form v.att are mapped to $\texttt{att}(a, \sigma)$. Expressions of the form c->collect(v : T | expr-v), where c is a collection and expr-v is an expression involving v, are mapped to $\texttt{collect}_f(c, \sigma)$, where $\mu(c) = c$ is the interpretation of c, and $\texttt{f}(v) = \mu(\text{expr-v})$. The only exception is where the expression is a role name, in which case c->collect(r) is interpreted as $\texttt{r}(c, \sigma)$.

Value expressions true and false are mapped to `true` and `false` respectively. Set expressions such as Set{}, Set{1,2} are mapped to `{}` and `{1,2}`, syntactic sugar for `insert(1,insert(2,{}))`, respectively. Other value expressions are mapped in a similar way.

OCL expressions	LSL expressions
v (variable)	`v` (variable)
v.att	$\texttt{att}(v, \sigma)$
v.op(v')	$\texttt{op}(v, v', \sigma)$
v.r (r role name)	$\texttt{r}(v, \sigma)$
c->select(v | b-expr-v)	$\texttt{select}_p(c, \sigma)$, where $\texttt{p}(v, \sigma) = \mu(\text{b-expr-v})$
c->reject(v | b-expr-v)	$\texttt{reject}_p(c, \sigma)$, where $\texttt{p}(v, \sigma) = \mu(\text{b-exp-v})$
c->collect(v | exp-v)	$\texttt{collect}_f(c, \sigma)$, where $\texttt{f}(v, \sigma) = \mu(\text{exp-v})$
c->iteate(v;acc=exp | expr-v-acc)	$\texttt{iterate}_{f,\text{exp}}(c, \sigma)$, where $\texttt{f}(v,\text{acc},\sigma) = \mu(\text{expr-v-acc})$
c.r (c collection, r role)	$\texttt{r}(c, \sigma)$, $(\mu(c) = c)$

Figure 4. Definition of the mapping μ

8: Conclusions

A precise semantics for a subset of OCL expressions together with the semantics for a kernel of the UML diagrammatic notation – class (type) diagrams, has been defined in terms of Larch. We have achieved nearly complete coverage of the OCL, although details have been omitted in some cases. Through this semantics, we have established that there is no need for flattening collection of collections when navigating from collections.

We have not considered meta level features in OCL, such as type casting and interrogation queries on objects. However, it is relatively a simple matter to formalise these in Larch.

Future semantics work includes:

- Semantics for constraint diagrams [18][19], a diagrammatic notation that allows most, if not all, OCL expressions to be given a diagrammatic characterisation.
- The use of this kernel to give the semantics of other aspects of UML. In particular state diagrams may be mapped to class diagrams with additional constraints expressed in OCL.
- Also being worked on is the semantics of extensions to UML suggested by Catalysis [4]. We are using the Larch trait inclusion mechanism to define the semantics of framework composition; and are working out the proof obligations for establishing conformance relationships between models, in particular between the specification and design in a refinement.

Apart from establishing precise, core concepts and checking the integrity and well-definedness of modelling notations, the semantics effort is also aimed at establishing a foundation for building CASE tools. We are currently experimenting with checking and animation tools written in Prolog, where the mapping from model to Prolog has benefited considerably from the work in Larch.

Acknowledgements

Thanks for comments are due to colleagues on the BIRO project at Brighton, in particular Franco Civello and Richard Mitchell. This research is partially funded by the UK EPSRC under grant number GR/K67304.

References

[1] Rational Software Corporation, *Object Constraint Language Specification*, Version 1.1, http://www.rational.com, 1997.

[2] A. Kleppe, J. Warmer, and S. Cook, “Informal Formality? The Object Constraint Language and its application in the metamodel”, *Proc. of UML’98 International Workshop*, P. Muller and J. Bezivin, ed., Mulhouse, France, June 3-4, 1998, pp. 127-136.

[3] S. Cook, and J. Daniels, *Designing Object Systems: Object-Oriented Modelling with Syntropy*, Prentice-Hall, Hemel Hempstead, UK, 1994, p. 389.

[4] D. D'Souza, and A. Wills, *Objects, Components and Frameworks with UML: The Catalysis Approach*, book submitted for publication by Addison-Wesley, UK, 1998, also available at http://www.trireme.com/catalysis.

[5] J. Rumbaugh, M. Blaha, W. Premerali, F. Eddy, and W. Lorensen, *Object-Oriented Modelling and Design*, Prentice-Hall, Emglewood Cliffs, New Jersy, 1991, p. 500.

[6] Rational Software Corporation, *The Unified Modeling Language*, Version 1.1, http://www.rational.com, 1997.

[7] C. Jones, *Systematic Software Development using VDM* (2nd edition), Prentice-Hall, Hemel Hempstead, UK, 1990, p. 333.

[8] M. Spivey, *The Z notation* (2nd ed.), Prentice Hall, UK, 1992.

[9] A. Hamie, F. Civello, J. Howse, S. Kent, and R. Mitchell, “Reflections on the Object Constraint Language”, *Proc. of UML’98 International Workshop*, P. Muller and J. Bezivin, ed., Mulhouse, France, June 3-4, 1998, pp. 137-145.

[10] H. Bourdeau, and B. Cheng, “A Formal Semantics for Object Model Diagrams”, *IEEE Transactions on Software Engineering*, Vol. 21, No. 10, 1995, pp. 799-821.

[11] R. France, J. Bruel, M. Larrondo-Petrie, and M. Shroff, “Exploring The Semantics of UML Type Structures with Z”, *Proc. Int’l Workshop on Formal Methods for Object-Based Distributed Systems* (*FMOODS’97*), Chapman and Hall, London, 1997, pp. 247-260.

[12] J. Bicarregui, K. Lano, and T. Maibaum, “Towards a Compositional Interpretation of Object Diagrams”, *Proc. IFIP TC2 Working conference on Algorithmic Languages and Calculi*, Chapman and Hall, 1997.

[13] J. Bicarregui, K. Lano, and T.S.E Maibaum, “Objects, Associations and Subsystems: a hierarchical approach to encapsulation”, *Proc. European Conf. of Object-Oriented Programming (ECOOP'97)*, LNCS 1241, Springer-Verlag, 1997, pp. 324-343.

[14] J. Fiadeiro, and T. Maibaum, “Temporal Theories and Modularisation Units for Concurrent System Specification”, *Formal Aspects of Computing*, Springer-Verlag, Vol. 4, No. 3, 1992, pp. 239-272.

[15] A. Hamie, and J. Howse, “Interpreting Syntropy in Larch”, Tech. Report ITCM97/C1, Computing Division, University of Brighton, Brighton, UK, 1997.

[16] A. Hamie, J. Howse, and S. Kent, “Navigation Expressions in Object-Oriented Modelling”, *Proc. of FASE in ETAPS98*, LNCS, 1382, Springer-Verlag, 1998, pp. 123-137.

[17] A. Hamie, J. Howse, and S. Kent, “Modular Semantics of Object-Oriented Models”, to be published in the proceedings of the *Third Northern Formal Methods WorkShop*, UK, 1998.

[18] S. Kent, “Constraint Diagrams: Visualising Invariants in Object-Oriented Models”, *Proc. of OOPSLA97*, ACM Press, 1997.

[19] S. Kent, “ Visualising Contracts in Object-Oriented Models”, *Proc. VISUAL98 in ETAPS’98*, Lisbon, Portugal, 1998.

[20] J. Guttag, and J. Horning, *Larch: Languages and Tools for Formal Specifications*, Springer-Verlag, 1993.

[21] M. Fowler, and K. Scott, *UML Distilled*, Addison-Wesley, 1997, p. 179.