Appendix A Object Constraint Language

The Object Constraint Language (OCL), a standard part of UML 1.1, is a specification language used in conjunction with UML models. It is an expression-based, side-effect-free language that eschews mathematical symbols (∀, ∃, and so on) for textual equivalents (forAll, exists). It uses a syntax more usual in object-oriented languages: ∀ x: T, p(x) becomes T->forAll (x | x.p).

OCL Summary

OCL uses a Smalltalk-based "block" syntax to allow you to define some kinds of functions conveniently and inline, but it does not provide corresponding type rules for this based on generic types. For our discussion here, we treat blocks as first-class functions and use the syntax (T1 X T2 X T3 -> T) for such a function. This is purely a syntactic convenience; functions can be modeled as objects. Any function f (a,b): c can be described as an object with a single method eval (a, b): c. Thus,

```
aCollection -> select ( x | p(x))        -- returns those elements for which p(x) is true
```

can be typed as follows:

```
Collection(T):: select ( T -> Boolean ): Collection(T)
-- select takes a block parameter that maps each element to a Boolean; select
   returns another collection of T
```

Most collection functions on Collection(T) use blocks that are Predicate(T) to do selection; Comparator(T) to do sorting; and Converter(T, T1) to do a general mapping from the collection elements.

```
Predicate(T) = T->Boolean
Comparator(T) = T X T -> Boolean
Converter(T,T1) = T -> T1
```

Collection types include sets (no duplicates), bags (duplicates), and sequences (duplicates, ordered). In the following description, explanations in italics indicate where we use Catalysis semantics or extensions or alternative syntax. Functions on collections also apply to sets, sequences, and bags.

Collection

Expression	Description
Collection(T)	Collections whose members all belong to type T. In the following expressions, c, c2: Collection; e: T; P: T -> Boolean; func: T ->Object.
c->size	Number of elements in the collection; for a bag or sequence, duplicates are counted as separate items.
c->sum	Sum of elements in the collection. Elements must be numbers *or have a + operation defined with the appropriate properties (described as a "provided" clause on the framework).*
c->count(e)	The number of times that e is in c.
c->isEmpty	Same as (c->size = 0).
c->notEmpty	Same as (not c->isEmpty).
c->asSet	A set corresponding to the collection (duplicates are dropped, sequencing is lost).
c->asSequence	A sequence corresponding to the collection. *More useful:* c->sortedBy (Comparator(T)).
c->asBag	A bag corresponding to the collection.
c = c2	A standard "same object" test on c and c2. The uniqueness constraints on collections are Set: no two sets have the same elements; Bag: no two bags have the same elements in the same counts; Sequence: no two sequences have the same elements in the same order. This is a specification ploy to simplify things. When dealing with mutable collections in a typical programming language, you would more likely use c.equals (c2), where equals does the comparison appropriate to that collection type.
c->includes(e)	Boolean; c->exists (x | x = e). *Catalysis alternative,* e: c.
c->excludes(e)	Boolean; not c->includes(e).
c->includesAll(c2)	Boolean; c includes all the elements in c2. *Catalysis alternative,* c2–>subsetOf(c)
c->including(e)	The collection that includes all of c as well as e. *Catalysis alternative,* c + e.
c->excluding(e)	The collection that includes all of c except e. *Catalysis alternative,* c – e.
c->exists(x | P)	Boolean; there is at least one element in c, named x, for which predicate P is true. *Catalysis alternative,* c [x | P] <> 0.
c->exists(P)	c->exists(self | P). *Catalysis alternative,* c [P] <> 0.
c->forAll(x | P)	Boolean; for every element in c, named x, predicate P is true. *In Catalysis, we use the context operator, "::", to mean for every element of this set, and write* x: c :: P.
c->forAll(P)	Same as c->forAll(self | P). *Catalysis alternative,* c :: P.
c->select(x | P)	The collection of those elements in c for which P is true. *Catalysis alternative,* c [x | P].
c->select(P)	Same as c->select(self | P). *Catalysis alternative,* c [P].
c->reject(x | P)	c->select(x | not P). *Catalysis alternative,* c [x | not P].
c->reject(P)	c->reject(self | P). *Catalysis alternative,* c [not P].
c->collect(x | E)	The bag obtained by applying E to each element of c, named x.
c->collect(E)	Same as c->collect(self | E).
c.attribute	The collection(of type of c) consisting of the attribute of each element of c.
c->iterate(x; a = E | E2)	The object obtained by applying E2 to each element of c, named x, where a is initialized to the value of the expression E.

Sequence

Expression	Description
Seq(T)	Sequence of elements of type T. In these expressions, s: Seq(T); e, x, y, z: T, i,j: Integer.
s->union (c)	The sequence obtained by appending c to s. *Catalysis alternative,* s + c.
s->append (e)	The sequence obtained by appending e to s. *Catalysis alternative,* s + e.
s->prepend (e)	The sequence obtained by prepending e to s. result = Seq { e, s }.
s->at (i)	The ith element of the sequence.
s->first	s->first = s->at(1).
s->last	s->last = s->at(s->size).
s->subSequence (i, j)	The sequence from positions i to j, inclusive (element positions start at 1). *In Catalysis you can define more convenient functions as extensions.*
Seq { x, y, z, x }	The sequence containing x, y, z, x, in that order.

Bag

Expression	Description
Bag(T)	Collection of elements of type T, with duplicates. In the following, b: Bag (T); e, x, y, z: T.
b->union (c)	The bag with all elements from b and c. *Catalysis alternative,* b + c.
b->intersection (c)	The bag with elements common to both b and c. *Catalysis alternative,* b * c.
Bag { x, y, z, x }	The bag containing two occurrences of x, one occurrence of y, and one occurrence of z.

Set

Expression	Description
Set(T)	The type of unordered collections of objects of type T with no duplicates. In the following, c: Collection; s1, s2: Set (T); x, y, z: T.
s1->union (c)	The set with all elements from s1 and c. *Catalysis alternative,* s1 + c.
s1->intersection(c)	The set with elements common to both s1 and c. *Catalysis alternative,* s1 * c.
s1->symmetric Difference (s2)	The set containing all the elements that are in s1 or in s2 but not in both. *Catalysis alternative,* s1 – s2.
Set { x, y, z }	The set containing x, y, z.

Object (Called OclAny)

Expression	Description
OclAny	The type that includes all others. In the following, x, y : OclAny, T is an OCL type.
x = y	x and y are the same object.
x <> y	not (x = y).
x.oclType	The type of x.
x.isKindOf (T)	True if T is a supertype (transitive) of the type of x. *Types coerce to sets, so the Catalysis alternative is* x : T.
x.isTypeOf (T)	True if T is equal to the type of x. *Don't use this one.*
x.asType (T)	Results in x, but of type T. Undefined if T is not the actual type of x or one of its subtypes.

OclType: A Metatype

Expression	Description
T: OclType	T is an OCL type.
T.new	The set of new instances of type T; also T*new. Not defined in OCL.
T.allInstances	All the instances of type T. *In Catalysis a type is a set, so this is not used.*

Boolean

Expression	Description
Boolean	Expressions yielding true, false (or unknown). In the following, b, b2: Boolean; e1, e2: Object.
b and b2, b or b2, b xor b2, not b	The standard operators. If any part of a Boolean expression fully determines the result, then it does not matter if some other parts of that expression have unknown or undefined results. *Catalysis alternatives:* b & b2, b | b2.
b implies b2	True if b is false or if b is true and b2 is true. *Catalysis alternative,* b ==> b2.
if b then e1 else e2 endif	If b is true the result is the value of e1; otherwise, the result is the value of e2.

String

Expression	Description
String	A sequence of ASCII characters. In the following, s, s2: String; l,u: Integer.
s = s2	s and s2 have the same characters in the same order.
s.size	The number of characters in s.
s.concat(s2)	The concatenation of s and s2.
s.substring(l, u)	The string from positions l to u, inclusive (positions start at 1).
s.toUpper	The value of s with all characters converted to uppercase characters.
s.toLower	The value of s with all characters converted to lowercase characters.

Real Numbers

Expression	Description
Real	Real numbers. In the following, r, r2: Real.
r = r2	r and r2 have the same value.
< > >= <=	Usual meaning for numbers.
+ –* /	Usual meaning for numbers.
r.abs	The absolute value of r. (result >= 0) and (result – r = 0).
r.floor	The largest integer, which is less than or equal to r. (result <= r) and (result + 1 > r).
r.ceiling	The smallest integer, which is greater than or equal to r. (result >= r) and (result –1 < r).
r.min(r2)	The minimum of r and r2. result = if r <= r2 then r else r2 endif.
r. max(r2)	The maximum of r and r2. result = if r >= r2 then r else r2 endif.

Integers

Expression	Description
Integer	Integers. In the following, i, i2: Integer.
i div i2	integer division. result * i2 <= i and result * (i2 + 1) > i.
i mod i2	i modulo i2. result = i – (i.div(i2) * i2).

OCL in Catalysis

In OCL the built-in objects are immutable and have fixed relations with other objects. In Catalysis we model all these relations as attributes or parameterized attributes. Thus, a < b is syntactical sugar for a.<(b), or a.isLessThan(b). Because Catalysis also offers package scope, this could also be described as a top-level query within a package: isLessThan (a, b).

We treat types as sets, so we can use type expressions such as Performer = Dancer + Singer. Note that this is different from inheriting from a common supertype Performer: In the former, Dancer and Singer are not defined in terms of Performer; instead, Performer is a type defined in terms of Dancer and Singer.

Catalysis collections are generic types in a template package that is predefined for basic UML. All collections are immutable; the many collection functions return other collections. A later section, Extending OCL, shows how extensions are readily accommodated in Catalysis.

In Catalysis, navigating through a collection yields a like collection: Bag->Bag, Sequence->Sequence, and so on. The asSet, asSeq, and asBag operators (or asType (T) equivalents) provide conversions. Thus, the following yields a bag of last names containing as many (possibly duplicated) entries as joe has friends.

```
joe.friends ->asBag.lastName
```

OCL does not provide a simple means to refer to the creation of a new object in a postcondition; Catalysis provides Type*new and the more conventional Type.new.

The Choice of "->"

We have found the OCL "->" symbol counterintuitive to use and to teach to others. Popular languages such as C and C++ use -> to mean "dereference," that is, to move through a level of indirection. In contrast, OCL uses it to mean *do not move through a level of indirection*.

```
joe.cars              -- the set of Joe's cars
joe.cars.size         -- navigate through the set, collecting each car's size into
                         the result
joe.cars -> size      -- do not navigate through; just give the size of the set
```

It would be more consistent to have the "." operator have the same meaning for collections and noncollection types and to use -> to apply to each element in the collection. In particular, Catalysis frameworks could be used to define the meaning of -> as no more than syntactic sugar for the following:

```
package Collections (X);
type Sequence(T)
    map ( T->X ) : Sequence(X)          -- sequence resulting from applying block to
                                           each element
    inv length = length(map(f)) & i: [0...length] map(f) [i] = f (self [i])
```

Now, c->a is just syntactic sugar for c.map (e | e.a).

Now, writing joe.cars -> size translates into joe.cars.map (e | e.size).

Extending OCL in Catalysis

OCL provides a fixed set of basic types and operations on numbers, sets, sequences, and so on. No such set will suffice for real applications unless they can be extended; many common constraints would be awkward if limited to the OCL set. The Catalysis semantics of packages lets us extend these basic types and operations without requiring any change to OCL. Thus, Catalysis sequences could support something like this:

```
Sequence(T):: shortestInitialSubsequence ( Predicate(Sequence(T)) )
```

This statement evaluates to the shortest initial subsequence that satisfies the test argument, assuming that such a subsequence exists.

A user-defined package can introduce additional convenient terms on existing OCL types as needed, without the need to modify the predefined OCL package in any way. According to the Catalysis package rules, they can even continue to use existing packages (see Figure A.1). Here are some expressions you might find useful as extensions to OCL:

```
Set(T)::
    any (Pred): T          -- returns any element that satisfies Pred
    one (Pred): Boolean    -- Boolean; exactly one element satisfies Pred
    theOne (Pred): T       -- returns the single element that satisfies Pred

Map(A,B)::                 -- a mapping from A to B
    domain                 -- standard definitions of domain
```

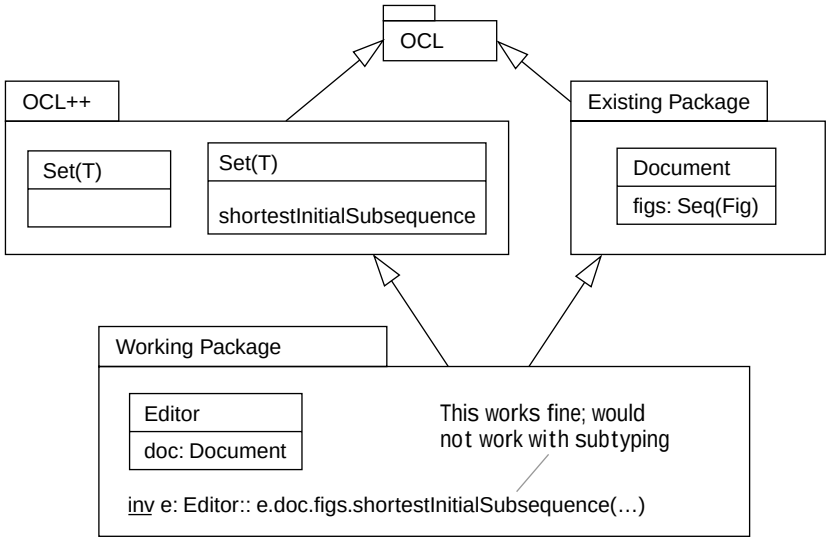


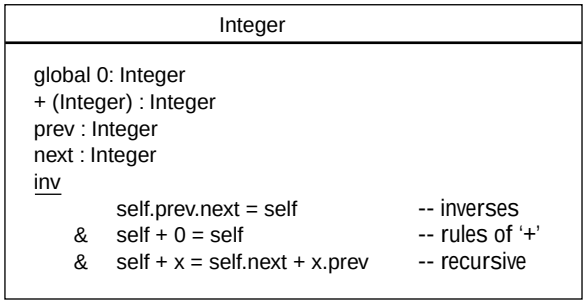
Figure A.1 Using Catalysis extension to customize OCL.

```
range                -- . . . and range
value (A)            -- the corresponding B
keys (B)             -- the set of A's that are mapped to this B
```

You will probably find Time, Date, DateRange, and Duration to be useful types. In reality, we would write these as frameworks on any total-ordered type (T, Range(T), and Delta(T)). For more details, see www.catalysis.org.

Defining Basic Types Using OCL

There are no "primitives" in Catalysis, only a set of basic object types that are defined in standard packages. Using exactly the same mechanisms, you can define your own basic types and packages of basic types suitable for your problem domain (see Figure A.2). Remember to add uniqueness constraints, which help define the meaning of object identity for your object type. There are frameworks, algebraic in nature, that capture even these basic recurring patterns.



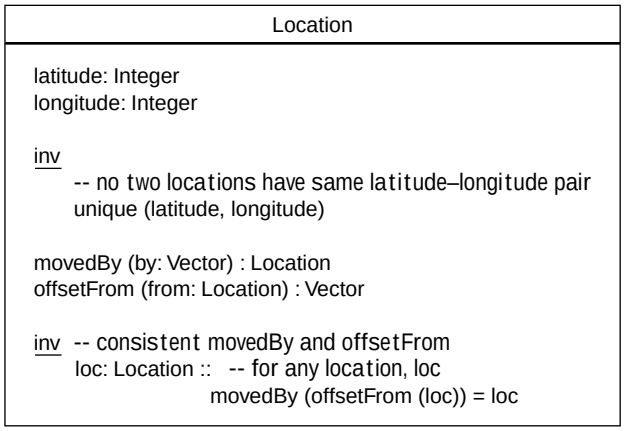


Figure A.2 Examples of defining basic types.