OCL Quick Reference

Eoin Woods, Zuhlke Engineering, July 2005.

This document provides a quick reference summary of the Object Constraint Language, as I understand it to be as of the UML 1.5 standard. It should not be taken as a definitive statement of OCL and does not attempt to provide formal semantic definitions. Refer to the *Resources* section at the end of the document for sources of further information on OCL. Reports of errors and omissions are gratefully received by email to `eoin@copse.org.uk`. This document is not a product or service provided by Zuhlke Engineering and has no implied warranty or endorsement from Zuhlke.

Uses of OCL

- Class model **invariants**; operation **pre and post conditions**; query **bodies** (definitions).
- Definition of **guards** in statechart models.

Basic Points

- All OCL expressions are side-effect free.
- No statement terminators ("`;`" or similar).
- Comments can be "`-- rest of line`" or "`/* block */`" (latter is OCLE only).
- Naming rules are implementation specific, the reference manual simply stating that "`::`" is used as a scope separator (e.g. `package::class::method()`).
- The term "operation" refers to a method with side-effects, the term "query" for a method that does not change the state (`state = state@pre` holds in the post-condition).

Setting Context

To define the model, package and scope of an expression:

```
model my_model -- OCLE specific
      package my_package
            context Class1
                  inv ... (invariant or whatever)
            context Class1::operation1(v1: Integer)
                  inv ...
      endpackage
endmodel
```

The context can be set to any model element (package, class, interface, component) or some sub-elements such as operation, attribute and in some cases (e.g. interaction diagram) an instance.

The reserved name "`self`" refers to the current object (a la "`this`" in C++ or Java).

OCLE extends this slightly by allowing context on inner classes to be specified using the syntax "`context Outer.Inner inv inv1 ...`".

Constraints

Can define invariants over model state or sets or pre/post conditions for operations or bodies (i.e. definitions) for queries.

Invariants:

```
context Class1
      inv attr1 > 100
context Class2
      inv secondInvariant: attr2 < 10
```

Can have as many "`inv`" statements as required, optionally named, and the resulting invariant is their conjunction ("`i1 and i2 and ...`").

Pre and Post Conditions:

```
context Class1::method1(v1: Integer) : Integer
pre valueIsLargeEnough: v1 >= 100
post: attr1 >= attr1@pre + 100 and result > v/10
```

The "`@pre`" notation refers to the "before" state (VDM's "hook" notation) and "`result`" is a reserved word for the result of the operation (if it has one).

Can have as many "`pre`" and "`post`" statements as required (optionally named) and the resulting pre/post conditions are their conjunction ("`p1 and p2 and ...`").

Can also use the notation "`item^method(val)`" in the post-condition to indicate that a method must have been called on a particular object reference by the operation.

Query Definitions

Queries don't change state and so pre/post-condition form isn't used to define them. Instead, they use a single expression in a "`body`" statement.

```
context Class1::query1(v: Integer) : Integer
body: v + 100 + attr1
```

The body statement defines the value for the query.

Definitions

Definitions, indicated by the "`def`", "`init`" or "`derive`" statements, are used to introduce new elements to the model or define further structural information about the model.

Introduce a Query

```
context Class1
def: getTotal() : Integer = items.value->sum()
```

Define an Initial Value

```
context Class1::attr1
init: 100
```

Define a Derived Attribute

```
context Class1::attr2
derive: attr1/100
```

Introduce a New Derived Value

```
context Class1
def: attr2 : Integer = attr1/100
```

Basic Types

The following atomic types (and their operations) are provided as part of the language:

- `Integer`, `Real`: =, <>, <=, >=, +, -, *, /, `x.mod(y)`, `x.div(y)` (div is integer division)
- `String`: `s.concat(t)`, `s.size()`, `s.toLower()`, `s.toUpper()`, `s.subString(start, end)` (indexing is "1-based").
    - OCLE also offers `contains(subStr : String):Boolean`, `pos(subStr : String):Integer` and `split(separators : String):Sequence(String)`.
- `Boolean`: `and`, `or`, `not`, `xor`, =, <>, `implies`, "`if b1 then ... else ... endif`".

Collection Types

The following types are provided for collections, all of which are subtypes of an abstract base called `Collection`.

- `Set` – no duplicates, no order.
- `Bag` – duplicates, no order.
- `OrderedSet` – no duplicates, ordered.
- `Sequence` – duplicates, ordered.

Collection type objects can be converted between types using built-in cast-like operators such as `seq1->asSet()` (see below).

An important rule in navigation expressions (e.g. `item1.subItems.value`) is that navigation through a ***single*** 1:m relationship (e.g. `item.subItem`) returns a ***set***, while navigation through ***more than one*** such relationship (e.g. `item1.subItems.value`) returns a ***bag***, while navigation through an `{ordered}` relationship results in a ***sequence***.

A related rule is that as collections (e.g. sets) are merged, they are *flattened* into a single collection of times, rather than forming a structured collection. The `collectNested()` operation can be used to avoid this (see below).

Collection Expressions

```
Type{initialiser}

Set{1, 2, 3}

Bag{'one', 'two', 'three', 'two'}

OrderedSet{true, false}

Sequence{1..30} – Special case for integers

Set{Set{1,2}, Set{2,3}} = Set{1,2,3} -- flattening
```

Collection Manipulation Operations

Operations are applied to collections using the "`->`" operator (e.g. `items->isEmpty()`, where "`items`" is a collection).

Note that all indexing is "1-based" rather than "0-based".

OCL provides a rich set of operations for use in collection expressions, including:

`= / <>`	Are the collections identical (not identical).
`-`	Return the value of the set difference of the arguments (`Set` and `OrderedSet` only).
`append(obj)`	Append `obj` to an ordered collection.
`asBag(), asSet(), asOrderedSet(), asSequence()`	Type conversion operations (available to/from all collection types).
`at(idx)`	Return object at index of ordered collection.
`count(obj)`	Number of times that `obj` appears in a collection.
`excludes(obj)`	Does `count(obj) = 0` ?
`excludesAll(coll)`	Does `count(obj) = 0` hold for all items in collection `coll`?
`excluding(obj)`	Value of collection with object `obj` removed.
`first()`	The first item in the ordered collection.
`includes(obj)`	Is `count(obj) > 0` ?
`includesAll(coll)`	Does `count(obj) > 0` hold for all items in collection `coll`?
`including(obj)`	Value of collection with object `obj` added.
`indexOf(obj)`	The index value of the (first) occurrence of an object in an ordered collection.
`insertAt(idx,obj)`	Value of the collection with the specified object inserted at the index of the ordered collection.
`intersection(coll)`	Value of the intersection of the unordered collection and the unordered collection `coll`.
`isEmpty()`	Is collection's `size() = 0` ?
`last()`	The last item in the ordered collection.
`notEmpty()`	Is collection's `size() > 0` ?
`prepend(obj)`	Value of the ordered collection with the object `obj` pre-pended to it.
`size()`	Number of items in the collection.
`subOrderedSet(start,end)`	Value of a subset of an ordered set based on indexing.
`subSequence(start,end)`	Value of part of a sequence, based on indexing.
`sum()`	Sum all items in the collection (Integer/Real only).

`symmetricDifference(coll)`	A collection containing the symmetric difference (items in either, not both – XOR) between a set and `coll` (defined on set only).
`union(coll)`	Return a collection which contains the combination of collection and `coll`.

Loop Operations

A another important set of operations on collections are the "loop" operators like `select()`, `collect()` and `forAll()`, which are used for applying predicates to collections.

`one(expr)`	Returns an item from the collection, for which the expression `expr` holds. If this applies to more than one, the particular item returned is non-deterministic. If `expr` does not hold for any items, the result is undefined.
`collect(expr)`	Returns a *bag* containing the value of the expression for each of the items in the collection (e.g. `items->collect(value)`). A simpler synonym for this operation is the period (".") operator (e.g. `items.value`).
`collectNested(expr)`	Behaves as `collect()` but does not flatten the collections, so resulting in a structured collection if a number of collections are merged (e.g. set of sets).
`exists(expr)`	Does expression `expr` hold for any items in the collection?
`forAll(expr)`	Does expression `expr` hold for *all* items in the collection?
`isUnique(expr)`	Returns `true` if the expression evaluates to a different value for each item in the collection, returns `false` otherwise.
`iterate(i : Type; a : Type | expr)`	Base iteration operation, from which others are defined. The "`i`" variable is the iterator, while "`a`" is the accumulator, assigned the value of the expression after each evaluation of it.
`one(expr)`	Returns the value of the expression `coll->select(expr)->size()=1`
`reject(expr)`	Returns the sub-collection of items in a collection for which expression expr does not hold.
`select(expr)`	Returns the sub-collection of items in a collection for which expression expr holds.
`sortedBy(expr)`	Returns a sequence containing all of the items from the collection, ordered by the value of the expression `expr` (the type of which much have the "<" operator defined for it).

The operations that select collection elements based on a condition can be used with two syntaxes, as follows. These two examples are equivalent.

```
set1->select(attr1 > 10)
set1->select(i | i.attr1 > 10)
```

In the second case the "`i`" is an "iterator" variable and can be thought of as being set to each of the elements of `set1` in turn.

Other Features

Let Expressions

Used to define temporary attributes or operations to allow them to be used in constraints to simplify the constraint or avoid repetition. For example:

```
context Class1
      inv abc:
      let value1:Boolean = att1 > 100 and att2 < 25
      let largeEnough(v :Integer):Boolean = v > 100
      in (val1 and attr3.mod(5)=0) or
            (val1 and attr4/5 > 10) or
            (largeEnough(val3))
```

OCLE Extensions

Some of the useful OCLE extensions are:

- `model` ... `endmodel`
- Block comments `/* ... */`
- Inner class contexts like "`Outer.Inner`".
- Static access to members of a class.
- Support for enumerations specified via the "`<<enumeration>>`" stereotype instead of using the more specific meta-class.
- Print operations called `dump()`, which always returns true and `dumpi()`, which returns the value it was called with.
    - Parameter is a string containing "`%n`" placeholders, "`%0`" is expression on which the operation is called, `%1`, `%2`, ... are values after the message.
    - Example: `attr1.dump("Attr1: %0 other=%1", "other value")`.
- The `pos()`, `contains()` and `split()` string operations.

Resources

- Annke Kleppe's company and OCL centre: `http://www.klasse.nl/ocl`.
- *The Object Constraint Language*, 2nd Edition, Jos Warmer and Anneke Kleppe, Addison Wesley, 2003.
- *OCLE*, a freely available OCL tool: http://lci.cs.ubbcluj.ro/ocle.
- *OCL 1.5 Specification*: `http://www.omg.org/cgi-bin/apps/doc?formal/03-03-13`.
- *OCL 2.0 Specification*: `http://www.omg.org/cgi-bin/apps/doc?ptc/2003-10-14`.