CS335: Model Driven Engineering & Object Constraint Language

Hao Wu

Computer Science Department,
National University of Ireland, Maynooth

Email: haowu@cs.nuim.ie

March 12, 2018

Basic OCL Elements

In OCL, each value, whether it is an object, a component instance, or a data value, has a certain type, which defines the operations that can be applied to the object. Types in OCL are divided into the following groups:

1. Predefined types, as defined in the standard library, including the following:
    - Basic Types
    - Collection Types
2. User-defined types

Predefined Basic Types

The predefined basic types are **Integer, Real, String**, and **Boolean**. Their definitions are similar to those in many known languages.

The predefined collection types are **Collection, Set, Bag, OrderedSet** and **Sequence**. They are used to specify the exact results of a navigation through associations in a class diagram.

Values Types and Object Types

OCL distinguishes between value types and object types. Both are types, i.e.,both specify instances, but there is one important difference: value types define instances that never change.

For example, the integer 1, will never change its value and become an integer with a value of 2.

Object types, or Classifiers, represent types that define instances that can change their value(s). An instance of the class *Person* can change the value of its attribute *name* and still remain the same instance.

Characteristcs

Another important characteristic of value types involves identity.

For value types, the value identifies the instance, hence the name. Two occurrences of a value type that have the same value are the same instance.

Tow occurrences of an object type are the same instance only if they have the same object identity.

Value types have value-based identity; object types have reference-based identity.

Both the predefined basic types and the predefined collection types of OCL are value types. The user-defined types can be either value types or object types. UML datatypes, including enumeration types, are value types. UML classes, components, and interfaces are object types.

Basic Types and Operators

The Boolean Type

A value of Boolean type can only be one of two values: true or false. A standard operation on the Boolean type that is uncommon to most programming languages. But often encountered in a more theoretical environment or in specification languages is the *implies* operation.

$$
\begin{aligned}
&a \ or \ b \\
&a \ and \ b \\
&a \ xor \ b \\
&not \ a \\
&a \ = \ b \\
&a \ <> \ b \\
&a \ implies \ b
\end{aligned}
$$

if-then-else

Another interesting operation on the Boolean type is the *if-then-else*. It is denoted in the following manner:

if $<$ *boolean OCL expression* $>$
then $<$ *OCL expression* $>$
else $<$ *OCL expression* $>$
endif

Some Examples

$$
\begin{array}{l}
not\ \ true \\
age() <> 21\ and\ age() < 65 \\
age() <= 12\ xor\ cards- > size()\ >\ 3 \\
if\ standard\ = \text{`}UML\text{'} \\
\ \ then\ \text{`}usingUMLstandard\text{'} \\
\ \ else\ \text{`}watchout:\ non\ UML\ features\ \text{'} \\
endif
\end{array}
$$

The Integer and Real Types

The Integer type in OCL represents the mathematical natural numbers. Because OCL is a modeling language, there are no restrictions on the integer values; in particular, there is no such thing as a maximum integer value.

In the same way, the Real type in OCL represents the mathematical concept of real values. As in mathematics, Integer is a subset of Real.

Operators

$$a = b$$
$$a < b$$
$$a > b$$
$$a <= b$$
$$a >= b$$
$$a + b$$
$$a - b$$
$$a * b$$
$$a / b$$
$$a.mod(b)$$
$$a.div(b)$$
$$a.abs()$$
$$a.max(b)$$
$$a.min(b)$$
$$a.round()$$
$$a.floor()$$

Some Examples

$$
\begin{aligned}
&2654 * 4.3 + 101 = 11513.2\\
&(3.2).\mathit{floor}()/3 = 1\\
&12 > 22.7 = \mathit{false}\\
&12.\mathit{max}(33) = 33\\
&13.\mathit{mod}(2) = 1\\
&-24.\mathit{abs}() = 24\\
&(-2.4).\mathit{floor}() = -3
\end{aligned}
$$

String

Strings are sequences of characters. Literal strings are written with enclosing single quotes, such as ‘apple’ or ‘weird cow’. The standard operations are as follows:

$$
\begin{array}{l}
string.concat(string) \\
string.size() \\
string.toLower() \\
string.toUpper() \\
string.substring(int, int) \\
string1 = string2 \\
string1 <> string2
\end{array}
$$

Collection Types

The Collection Types

Within OCL, there are five collection types. Four of them: the **Set, OrderedSet, Bag** and **Sequence** types are concrete types and can be used in expressions.

The **Collection** type is the abstract supertype of the other four and is used to define operations common to all collection types.

Definitions

1. A **Set** is a collection that contains instances of a valid OCL type. A set does not contain duplicate elements; any instance can be present only once. Elements in the set are not ordered.
2. An **OrderedSet** is a set whose element are ordered.
3. A **Bag** is a collection that may contain duplicate elements; that is, the same instance may occur in a bag more than once. A bag is typically the result of combining navigations.
4. A **Sequence** is a bag whose elements are ordered.

Note that a value of type **Sequence** and **OrderedSet** is ordered and not sorted. Each element has a sequence number, like array elements in programming languages. This does not mean that the element at certain sequence number is in any sense less than or greater than the element before it.

Collection Constants

Constant sets, ordered sets, sequences, and bags can be specified by enumerating their elements. Curly brackets should surround the elements of the collection, and the elements are separated by commas. The type of the collection is written before the curly brackets, as shown in the following examples:

$$
\begin{aligned}
&Set\{1, 2, 5, 88\}\\
&Set\{`apple', `orange', `stawberry'\}\\
&OrderedSet\{`apple', `orange', `stawberry', `pear'\}\\
&Sequence\{1, 3, 45, 2, 3\}\\
&Bag\{1, 3, 4, 3, 5\}\\
&Sequence\{1..(6+4)\}
\end{aligned}
$$

Collection Operations

All operations on collections are denoted in OCL expressions using an arrow; the operation following the arrow is applied to the collection before the arrow. All collection types are defined as value types; that is, the value of an instance cannot be changed. Therefore, collection operations do not change a collection, but they may result in a new collection.

```
context LoyaltyProgam
inv: self.participants->size() < 1000
```

Treating Instances as Collections

Because the OCL syntax for applying collection operation is different from that for user-defined type operations, we can use a single instance as a collection. This collection is considered to be a set with the instances as the only element.

```
context Membership
inv: account.isEmpty()
inv: account->isEmpty()
```

The second class invariant uses the Set operation isEmpty, where account is used as a collection.

Collections of Collections

A special feature of OCL collection is that in most cases, collections are automatically flattened; that is, a collection does not contain collections but only simple objects.

$$\begin{array}{l} Set\{Set\{1,2\}, Set\{3,4\}, Set\{5,6\}\} \\ Set\{1,2,3,4,5,6\} \end{array}$$

When a collection is inserted into another collection, the resulting collection is automatically flattened.

Operations on Collection Types

The standard operations on all collection types are as follows:

$$
\begin{array}{l}
count(object)\\
excludes(object)\\
excludesAll(collection)\\
includes(object)\\
includesAll(collection)\\
isEmpty()\\
notEmpty()\\
size()\\
sum()
\end{array}
$$

The equals and notEquals Operations

The **equals** operator (denoted by $=$) evaluates to true if all elements in two collections are the same.

For **sets**, this means that all elements present in the first set must be present in the second set and vice versa.

For **ordered sets**, an extra restriction specifies that the order in which the elements appear must also be the same.

For two **bags** to be equal, not only must all elements be present in both, but the number of times an element is present must also be the same.

For two **sequences**, the rules for bags apply, plus the extra restriction that the order of elements must be equal.

The **notEquals** operator (denoted by $<>$) evaluates to true if all elements in two collections are not the same. The opposite rules as for the **equals** operator apply.

The including and excluding Operations

The **including** operation results in a new collection with one element added to the original collection.

For a bag, this description is completely true. If the collection is a set or ordered set, then the element is added only if it is not already present in the set; otherwise the result is the equal to the original collection.

If the collection is a sequence or an ordered set, the element is added after all elements in the original collection.

The **excluding** operation results in a new collection with an element removed from original collection. From a set or ordered set, it removes only **one element**. From a bag or sequence, it removes **all occurrences** of the given object.

Loop Operations or Iterators

Iterator Vairables

Every iterator operation may have an extra parameter, an **iterator variable**. An iterator variable is a variable that is used within the body parameter to indicate the element of the collection for which the body parameter is being calculated. The type of this iterator variable is always the type of the elements in the collection. Because the type is known, it may be omitted in the declaration of the iterate variable.

```
context LoyaltyProgam
inv: self.Membership.account->isUnique(acc | acc.number)

context LoyaltyProgam
inv: self.Membership.account->isUnique(acc:LoyaltyAccount
         | acc.number)
```

The *sortedBy* Operation

We can demand an ordering on the elements of any collection using the *sortedBy* operation. The parameter of this operation is a property of the type of elements in the collection.

The operation will loop over all elements in the original collection and will order all elements according to the value derived from calculating the parameter property.

```
context LoyaltyProgram
def: sortedAccounts: Sequence (LoyaltyAccount) =
        self.Membership.account->sortedBy(number)
```

The *select* Operation

Sometimes an expression using operations and navigations results in a collection, but we are interested only in a special subset of the collection. The *select* operation enables us to specify a selection from the original collection.

The parameter of the *select* operation is a boolean expression that specifies which elements we want to select from the collection. The result of *select* is the collection that contains all elements for which the boolean expression is true.

```
context CustomerCard
        inv: self.transcations->select(points > 100)
                ->notEmpty()
```

The *reject* Operation

The *reject* operation is analogous to *select*, with the distinction that *reject* selects all elements from the collection for which the expression evaluates to false. The existence of *reject* is merely a convenience. The following two invariants are semantically equivalent:

```
context CustomerCard
inv: Membership.account->select(points>0)
inv: Membership.account->reject(not(points >0))
```

The *any* Operation

To obtain any element from a collection for which a certain condition holds, we can use the *any* operation. The body parameter of this operation is a boolean expression. The result is a **single** element of the original collection.

If the condition holds for more than one element, one of them is randomly chosen. If the condition does not hold for any element in the source collection, the result is undefined.

```
context CustomerCard
inv: Membership.account->any(number<10000)
```

The *forAll* Operation

We often want to specify that a certain condition must hold for all elements of a collection. The *forAll* operation on collections can be used for this purpose. The result of the *forAll* operation is a boolean value. It is true if the expression is true for **all** elements of the collection. If the expression is false for **one** or **more** elements in the collection, the *forAll* results in false.

```
context LoyaltyProgram
inv: participants->forAll(age() <= 70)
inv: self.participants->forAll(c1,c2 |
        c1<>c2 implies c1.name<>c2.name)
inv: self.participants->forAll(c1 | self.participants->
        forAll(c2 | c1<>c2
                implies c1.name<>c2.name) )
```

The *exists* Operation

We want to specify that there is **at least one** object in a collection for which a certain condition holds. The *exists* operation on collections can be used for this purpose. The result is true if the expression is true for at least one element of the collection. If the expression is false for all elements in the collection, then the *exists* operation results in false.

```
context LoyaltyAccount
inv: points > 0 implies
        transactions->exists (t | t.points > 0)
```

The following two expressions are equivalent.

```
collection->exists (<expression>)
not collection->forAll(not <expression>)
```

The *one* Operation

The *one* operation gives a boolean result stating whether there is **exactly** one element in the collection for which a condition holds. If there is exactly one such element, then the result is true; otherwise, the result is false.

```
context LoyaltyProgrram
inv: self.Membership.account->one (number<1000)
```

Note the different between the *any* and *one* operations. The *any* operation can be seen as a variant of the *select* operation: its result is an element selected from the source collection. The *one* operation is a variant of the *exists* operation: its result is either true or false depending on whether or not a certain element exists in the source collection.

The *collect* Operation

The *collect* operation iterates over the collection, computes a value for each element of the collection, and gathers the evaluated values into a new collection. The type of the elements in the resulting collection is usually different from the type of the elements in the collection on which the operation is applied.

```
context LoyaltyAccount
inv: transcations->collect(points) ->
         exists (p:Integer | p =500)
```

The *iterate* Operation

The *iterate* operation is the most fundamental and complex of the loop operations. The syntax of the *iterate* operation is as follows:

```
collection->iterate(element : Type1; result: Type2 =
        <expression> |
        <expression-with-element-and-result>)
```

```
Set{1,2,3}->iterate(i:Integer;
        sum:Integer=0 | sum + i)
```