Object Constraint Language (OCL)

Perdita Stevens

School of Informatics
University of Edinburgh

OCL for this course

The following slides *summarise* what you will need to know, but are not complete notes.

Required reading from the OCL standard will give you the details.

Bottom line: you should be able to read and write straightforward OCL constraints. But you are not expected e.g. to memorise the list of reserved words!

OCL basic types

- Boolean
- String
- Integer
- Real
- (UnlimitedNatural)

With all the operations you'd expect. (Know the ones in Table 7.2 of spec.)

Integer is considered a subtype of Real.

(Remark: OCL uses the terms class and type interchangeably, which is just about OK in this context, though normally a big mistake.)

Example: class invariant

```
context Company inv:
  self.numberOfEmployees > 50
```

Note that declaring the context to be `Company` means that `self` refers to an instance of class `Company`.

Specifying that this is a class invariant (`inv`) means that the constraint has to be true of every instance of class `Company`.

Example: pre and post conditions

```
context Stove::open()
  pre: status = OVENSTATUS::off
  post: status = OVENSTATUS::off and isOpen
```

Here `status` and `isOpen` are attribute names of `Stove`. We could have written `self.status` etc.

`off` is a member of the enum type OVENSTATUS.

More features useful for pre and post conditions

Arguments and return type could be specified in the context:

`context MyClass::foo(i:Integer):String`

and then `i` can be referred to in this context.

Reserved word `result` can be used in the postcondition.

You can refer to the old value of an attribute using `pre`, e.g.

```
context MyClass::incrementCount()
post:count = count@pre + 1
```

OCL collection types

- Collection
- Set
- Bag
- Sequence
- (Tuple)
- (OrderedSet)

Set, Bag, Sequence are kinds of Collection: more specifically, Set(S) conforms to Collection(T) iff S conforms to T, etc.

Reasonable facilities for manipulating collections. E.g.

```
context Company inv:
self.employee->select(age > 50)->notEmpty()
```

Note the use of the arrow to access properties of collections...

Collections operations returning collections

(NB All these have variants that allow you to name the collection element.)

```
collection->select(boolean_expression)
collection->reject(boolean_expression)
```

(you might recognise this as filter in FP?)

```
collection->collect(expression)
```

(like map in FP) NB collect on a Set gives you a Bag.

Conversion operations, especially:

```
collection->asSet()
```

Collections operations returning boolean

Emptiness checking:

```
collection->isEmpty()
collection->notEmpty()
```

Quantifiers:

```
collection->forAll(boolean_expression)
collection->exists(boolean_expression)
```

Convenient variant:

```
self.employee->forAll( e1, e2 : Person
  | e1 <> e2 implies e1.forename <> e2.forename)
```

Collections operations returning numbers

```
collection->sum() -- type depends on element type
collection->size()
```

Navigation

As we've seen, an OCL expression in the context of one class A may refer to an associated class B.

Single (? - 1) association: straightforward, since any object of class A (Student say) determines just one object of class B (Adviser say):

- If there's a rolename use it, e.g. `self.studentadviser.name`
- If not may just use lowercased classname, e.g. `self.adviser.name`

More navigation

What if the association is not (? - 1)? E.g. consider the same association from the point of view of the Adviser – an Adviser may advise many Students.

For each Adviser the rolename advisee refers to a *set* of Students. Use OCL collection operations, e.g.

```
self.advisee->forAll (regNo <= 200000)
self.advisee->notEmpty()
```

(If you use a collection operation on something that isn't a collection it gets interpreted as a set containing one element!)

Two-stage navigation

What happens if we take more than one “hop” round the class diagram?

e.g. what is `self.student.module`?

It’s deemed to be short for

```
self.student->collect(module)
```

which is a *Bag* (not a Set) of all the modules taken by students linked to self.

Notice that putting such a constraint into a UML model creates a dependency of self on module, if there wasn’t one already.

Using operations in OCL

Consider an operation register(s:Student) of Module. Should we be able to refer to this operation in an OCL expression?

Problem: it does something – alters the state of the Module. When should this happen, if at all?

Only good way round this is to allow in OCL *only* operations that guarantee not to alter the state of any object.

Such operations are known as queries – in UML an operation has an attribute isQuery which must be true for the operation to be legal in OCL.

“Control” structures

Naturally we don’t need much in the way of control structures: OCL is a constraint language, used for defining expressions (not commands, not functions).

- `if ... then ... else ... endif`
- `let v : Sometype = someExpression in ...`
  (NB there’s no endlet! Typical use: let begins the whole constraint, and its scope extends to the end.)