OCL : Object Constraint Language

Reference: UML 2.0 OCL Specification, ptc/03-10-14 http://www.omg.org/docs/ptc/03-10-14.pdf

OCL is a *query-only* language. It can't modify the model in any way. It can be used to express preconditions, postconditions, invariants, guard conditions, and results of method calls.

Abbreviations (not official, for this document only).

i : Integer	c : Collection(T)	os : OrderedSet(T)	cs: constant
r : Real	st: Set(T)	t : Tuple(…)	pd : predicat
b : Boolean	bg : Bag(T)	id: identificateur	e : expression
s : String	sq : Sequence(T)	pt: property	ns: namespace

Syntactic constructs

c	e op e	ns**::** ... ns**::**id
id	e**.**id	**if** pd **then** e **else** e **endif**
self	e**.**pt (e, ... , e)	**let** id **=** e **:** T, id2 = e:T, ... **in** e2
t	c **->** pt (e, ..., e)	

Standard library

Type	Value	Operations
		`e1 = e2, e1 <> e2`
Integer	`1, -5, 34`	`i+i2, i-i2, i*i2, i.div(i2), i.mod(i), i.abs( ), i.max(i2), i.min(i2), <, >, <=, >=`
Real	`1.5, 1.34, ...`	`r+r2, r-r2, r*r2, r/r2, r.floor( ), r.round( ), r.max(r2), r.min(r2), <, >, <=, >=`
Boolean	`true, false`	`not b, b and b2, b or b2, b xor b2, b implies b2`
String	`'' a string''`	`s.size(), s.concat(s2), s.substring(i1,i2), s.toUpper(), s.toLower(), s.toInteger(), s.toReal()`
Enumeration	`Day::Monday, Day::Tuesday,`	`=, <>`
TupleType(x : T1, y : T2, z : T3 **)**	`Tuple { y : T2 = …, x = … , z = … }`	`t.x t.y t.z`
Collection(T**)**		`c->size(), c->includes(o), c->excludes(o) c->count(o), c->includesAll(c2) c->excludesAll(c2), c->isEmpty(), c->notEmpty() c->sum(), c->exists(p), c->forall(p), c->isUnique(e) c->sortedBy(e), c->iterate(e)`
Set(T**)**	`Set{1,5,10,3} Set{}`	`st->union(st2), st->union(bg) st->intersection(st2), st->intersection(bg) st - st2, st->including(e), st->excluding(e) st->symmetricDifference(st2), st->select(e) st->reject(e), st->collect(e), st->count(e) st->flatten(), st->asSequence(), st->asBag()`
Bag(T**)**	`Bag {1,5,5} Bag {}`	`bg->union(bg2), bg->union(st), bg->intersection(bg2) bg->intersection(st), bg->including(e) bg->excluding(e), bg->count(e), bg->flatten() bg->select(e), bg->reject(e), bg->collect(e) bg->asSequence(), bg->asSet()`
OrderedSet(T**)**	`OrderedSet{10,4,3} OrderedSet{}`	…
Sequence(T**)**	`Sequence{5,3,5} Sequence{}`	`sq->count(e), sq->union(sq2), sq->append(e) sq->prepend(e), sq->insertAt(i,o) sq->subSequence(i1,i2), sq->at(i), sq->first() sq->last(), sq->indexOf(o), sq->including(e) sq->excluding(e), sq->select(e), sq->reject(e) sq->collect(e), sq->iterate(e), sq->asBag, sq->asSet`

Operator precedence

- `@pre`
- dot and arrow operations: `'.'` and `'->'`
- unary `'not'` and unary minus `'-'`
- `'*'` and `'/'`
- `'+'` and binary `'-'`
- `'if-then-else-endif'`
- `'<'`, `'>'`, `'<='`, `'>='`
- `'='`, `'<>'`
- `'and'`, `'or'` and `'xor'`
- `'implies'`
-

Keywords

Keywords in OCL are reserved words. That means that the keywords cannot occur anywhere in an OCL expression as the name of a package, a type or a property. The list of keywords is shown below:
`and`, `attr`, `body`, `context`, `def`, `else`, `endif`, `endpackage`, `if`, `implies`, `in`, `inv`, `let`, `not`, `oper`, `or`, `package`, `post`, `pre`, `then`, `xor`

Comments

Comments in OCL are written following two successive dashes (minus signs). Everything immediately following the two dashes up to and including the end of line is part of the comment. For example:

```
-- this is a comment
```

Undefined values

Some expressions will, when evaluated, have an undefined value. For instance, typecasting with oclAsType() to a type that the object does not support or getting the ->first() element of an empty collection will result in undefined. In general, an expression where one of the parts is undefined will itself be undefined. There are some important exceptions to this rule, however. First, there are the logical operators:

- True OR-ed with anything is True
- False AND-ed with anything is False
- False IMPLIES anything is True
- anything IMPLIES True is True

The rules for OR and AND are valid irrespective of the order of the arguments and they are valid whether the value of the other sub-expression is known or not.
The IF-expression is another exception. It will be valid as long as the chosen branch is valid, irrespective of the value of the other branch.
Finally, there is an explicit operation for testing if the value of an expression is undefined. `oclIsUndefined()` is an operation on `OclAny` that results in True if its argument is undefined and False otherwise.

Built-in Object Properties

OCL provides a set of properties on all objects in a system.
The built-in properties are:

`oclIsTypeOf(`*`t:Type`*`): Boolean`
Returns true if the tested object is exactly the same type as *t*.

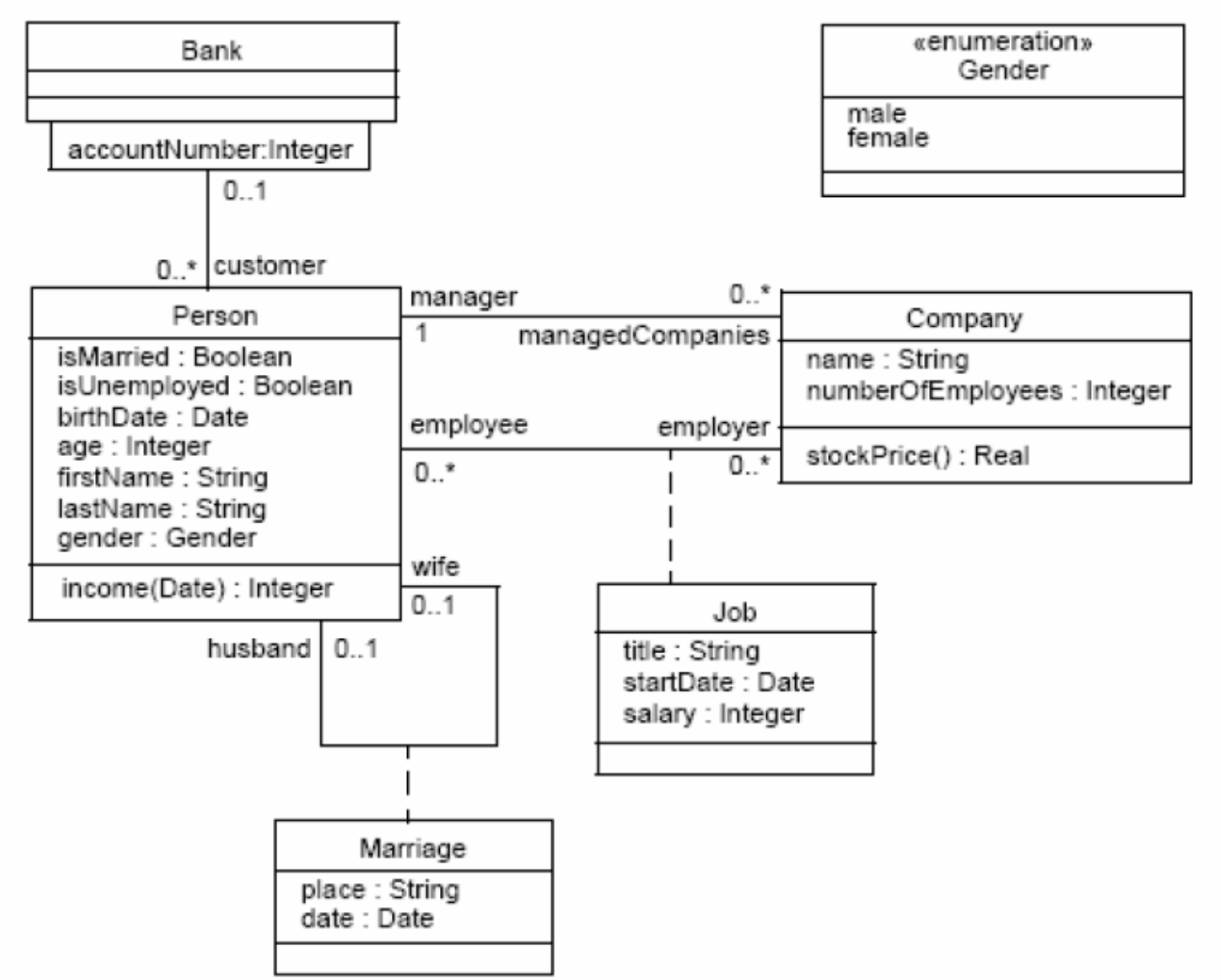
`oclIsKindOf(`*`t:Type`*`): Boolean`
Returns true if the tested object is exactly the same type or a subtype of *t*.

`oclInState(`*`s: State`*`): Boolean`
Returns true if the tested object is in state *s*. The states you can test must be part of a state machine attached to the classifier being tested.

`oclIsNew( ): Boolean`
Designed to be used in a postcondition for an operation, it returns true if the object being tested was created as a result of executing the operation.

`oclAsType (`*`t:Type`*`): Type`
Returns the owning object casted to *Type*. If the object is not a descendant of *t*, the operation is undefined.

Example
(Borrowed from ptc/05-06-06)



Invariants

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

context c:Company inv:
      c.numberOfEmployees > 50

context c:Company inv enoughEmployees:  -- named invariant
      c.numberOfEmployees > 50
```

Pre and Post conditions

```
context Person::income(d: Date): Integer
      post: result = 5000

context Person::income(d: Date): Integer
      pre: d > 2000
      post: result = 5000

context Person::income(d: Date): Integer  -- with optional condition names
      pre parameterOK: d > 2000
      post resultOK: result = 5000
```

Let expressions
Sometimes a sub-expression is used more than once in a constraint. The let expression allows one to define a variable which can be used in the constraint.

```
      context Person inv:
```

```
let income : Integer = self.job.salary->sum() in
if isUnemployed then
      income < 100
else
      income >= 100
endif
```

A let expression may be included in any kind of OCL expression. It is only known within this specific expression.

Additional operations/attributes through «definition» expressions

The Let expression allows a variable to be used in one Ocl expression. To enable reuse of variables/operations over multiple OCL expressions one can use a Constraint with the stereotype «definition», in which helper variables/operations are defined.
This «definition» Constraint must be attached to a Classifier and may only contain variable and/or operation definitions, nothing else. All variables and operations defined in the «definition» constraint are known in the same context as where any property of the Classifier can be used. Such variables and operations are attributes and operations with stereotype «OclHelper» of the classifier. They are used in an OCL expression in exactly the same way as normal attributes or operations are used. The syntax of the attribute or operation definitions is similar to the Let expression, but each attribute and operation definition is prefixed with the keyword **'def'** as shown below.

context Person
def: income : Integer = self.job.salary->sum()
def: nickname : String = 'Little Red Rooster'
def: hasTitle(t : String) : Boolean = self.job->exists(title = t)

The names of the attributes / operations in a let expression may not conflict with the names of respective attributes/ associationEnds and operations of the Classifier.
Using this definition syntax is identical to defining an attribute/operation in the UML with stereotype «OclHelper» with an attached OCL constraint for its derivation.

Properties: AssociationEnds and Navigation
Starting from a specific object, we can navigate an association on the class diagram to refer to other objects and their properties.
To do so, we navigate the association by using the opposite association-end:

```
object.associationEndName
```

The value of this expression is the set of objects on the other side of the associationEndName association. If the multiplicity of the association-end has a maximum of one ("0..1" or "1"), then the value of this expression is an object. In the example class diagram, when we start in the context of a Company (i.e., self is an instance of Company), we can write:

```
context Company
      inv: self.manager.isUnemployed = false
      inv: self.employee->notEmpty()
```

In the first invariant *self.manager* is a Person, because the multiplicity of the association is one. In the second invariant *self.employee* will evaluate in a Set of Persons. By default, navigation will result in a Set. When the association on the Class Diagram is adorned with {ordered}, the navigation results in an OrderedSet.

Collections, like Sets, OrderedSets, Bags, and Sequences are predefined types in OCL. They have a large number of predefined operations on them. A property of the collection itself is accessed by using an arrow '->' followed by the name of the property. The following example is in the context of a person:

```
context Person inv:
      self.employer->size() < 3
```

This applies the size property on the Set *self.employer*, which results in the number of employers of the Person self.

```
context Person inv:
      self.employer->isEmpty()
```

This applies the *isEmpty* property on the Set self.employer. This evaluates to true if the set of employers is empty and false otherwise.

Missing AssociationEnd names
When the name of an association-end is missing at one of the ends of an association, the name of the type at the association end starting with a lowercase character is used as the rolename. If this results in an ambiguity, the rolename is mandatory. This is e.g. the case with unnamed rolenames in reflexive associations. If the rolename is ambiguous, then it cannot be used in OCL.

Navigation over Associations with Multiplicity Zero or One

Because the multiplicity of the role manager is one, *self.manager* is an object of type Person. Such a single object can be used as a Set as well. It then behaves as if it is a Set containing the single object. The usage as a set is done through the arrow followed by a property of Set. This is shown in the following example:

```
context Company inv:
      self.manager->size() = 1
```

The sub-expression *self.manager* is used as a Set, because the arrow is used to access the size property on Set. This expression evaluates to true.

```
context Company inv:
      self.manager->foo
```

The sub-expression *self.manager* is used as Set, because the arrow is used to access the foo property on the Set. This expression is incorrect, because foo is not a defined property of Set.

```
context Company inv:
      self.manager.age > 40
```

The sub-expression *self.manage*r is used as a Person, because the dot is used to access the age property of Person.
In the case of an optional (0..1 multiplicity) association, this is especially useful to check whether there is an object or not when navigating the association. In the example we can write:

```
context Person inv:
      self.wife->notEmpty() implies self.wife.gender = Gender::female
```

Combining Properties

Properties can be combined to make more complicated expressions. An important rule is that an OCL expression always evaluates to a specific object of a specific type. After obtaining a result, one can always apply another property to the result to get a new result value. Therefore, each OCL expression can be read and evaluated left-to-right.
Following are some invariants that use combined properties on the example class diagram:

[1] Married people are of age >= 18

```
context Person inv:
      self.wife->notEmpty() implies self.wife.age >= 18 and
      self.husband->notEmpty() implies self.husband.age >= 18
```

[2] a company has at most 50 employees

```
context Company inv:
      self.employee->size() <= 50
```

Iterate Operation

The iterate operation is slightly more complicated, but is very generic. The operations reject, select, forAll, exists, collect, can all be described in terms of iterate. An accumulation builds one value by iterating over a collection.

```
collection->iterate( elem : Type; acc : Type = <expression> |
                                        expression-with-elem-and-acc )
```

The variable elem is the iterator, as in the definition of select, forAll, etc. The variable acc is the accumulator. The accumulator gets an initial value <expression>. When the iterate is evaluated, elem iterates over the collection and the expression-with-elem-and-acc is evaluated for each elem. After each evaluation of expression-with-elem-and-acc, its value is assigned to acc. In this way, the value of acc is built up during the iteration of the collection. The collect operation described in terms of iterate will look like:

```
collection->collect(x : T | x.property)
-- is identical to:
collection->iterate(x : T; acc : T2 = Bag{} | acc->including(x.property))
```

Or written in Java-like pseudocode the result of the iterate can be calculated as:

```
iterate(elem : T; acc : T2 = value)
{
      acc = value;
      for(Enumeration e = collection.elements() ; e.hasMoreElements(); ){
            elem = e.nextElement();
            acc = <expression-with-elem-and-acc>
      }
      return acc;
}
```

Although the Java pseudo code uses a ‘next element’, the iterate operation is defined not only for Sequqnce, but for each collection type. The order of the iteration through the elements in the collection is not defined for Set and Bag. For a Sequence the order is the order of the elements in the sequence.

Other Examples : (from J-M Favre, http://www-adele.imag.fr/~jmfavre)

```
context Personne
  inv :  enfants->forall( e | e.age < self.age - 7)
  inv :  enfants->forall( e : Personne | e.age < self.age - 7)
  inv i3 :  enfants->forall( e1,e2 : Personne |  e1 <> e2 implies e1.prénom <> e2.prénom)
  inv i4 :  self.enfants -> isUnique ( prénom )
  def cousins : Set(Personne) = parents.parents.enfants.enfants->excluding(
parents.enfants )->asSet()

context Personne::age : Integer
  init : 0

context Personne::estMarrié : Boolean
  derive : conjoint->notEmpty()

context Personne::salaire() : integer
   post : return > 5000

context Compagnie::embaucheEmployé( p : Personne)
  pre pasPrésent : not (employés->includes(p))
  post embauché : employés->includes(p)

context Personne::grandsParents() : Set(Personne)
   body : parents.parents->asSet()

(age<40 implies salaire>1000) and (age>=40 implies salaire>2000)
if age<40 then salaire > 1000 else salaire > 2000 endif
salaire > (if age<40 then 1000 else 2000 endif)
nom= nom.substring(1,1).toUpper().concat(
          nom.substring(2,nom.size()).toLower())
épouse->notEmpty() implies épouse.sexe = Sexe::Feminin
Set { 3, 5, 2, 45, 5 }->size()
Sequence { 1, 2, 45, 9, 3, 9 } ->count(9)
Sequence { 1, 2, 45, 2, 3, 9 } ->includes(45)
Bag { 1, 9, 9, 1 } -> count(9)
c->asSet()->size() = c->size()
c->count(x) = 0
Bag { 1, 9, 0, 1, 2, 9, 1 } -> includesAll( Bag{ 9,1,9} )
self.enfants ->select( age>10 and sexe = Sexe::Masculin)
self.enfants ->reject(enfants->isEmpty())->notEmpty()
membres->any(titre='président')

self.employé->select(age > 50)
self.employé->select( p | p.age>50 )
self.employé->select( p : Personne | p.age>50)
self.enfants->forall(age<10)
self.enfants->exists(sexe=Sexe::Masculin)
self.enfants->one(age>=18)self.enfants->forall( age < self.age )
self.enfants->forall( e | e.age < self.age - 7)
self.enfants->forall( e : Personne | e.age < self.age - 7)
self.enfants->exists( e1,e2 | e1.age = e2.age )
self.enfants->forall( e1,e2 : Personne |
                                e1 <> e2 implies e1.prénom <> e2.prénom)
self.enfants -> isUnique ( prénom )
self.enfants->collect(age) = Bag{10,5,10,7}
self.employés->collect(salaire/10)->sum()
self.enfants.enfants.voitures
enfants.enfants.prénom = Bag{ 'pierre', 'paul', 'marie', 'paul' }

enfants->collectNested(enfants.prénom) =
          Bag { Bag{'pierre', 'paul'}, Bag{'marie','paul'}

Sequence{1..s->size()-1} -> forall(i | s.at(i) < s.at(i+1) )
enfants->sortedBy( age )
```

```
enfants->sortedBy( enfants->size() )->last()
let ages = enfants.age->sortedBy(a | a) in ages.last() - ages.first()
s.Emploi
p.Emploi
s.Emploi->collect(salaire)->sum()

s.Emploi.salaire->forall(x | x>500)
p.Evaluation[chefs]
p.Evaluation[employés]
p.Evaluation[chefs].note -> sum()s.Emploi-> select(salaire<1000).employé
p.enfants->select(oclIsKindOf(Femme)).asTypeOf(Set(Femme)) ->select(nomDeJF <> nom)
Personne.allInstances->size() < 500
Personne.allInstances->forall(p1,p2 | p1<>p2 implies p1.numsecu <> p2.numsecu)
Personne.allInstances->isUnique(numsecu)
```