OCL 2.0 Specification June 2005

Version 2.0

ptc/2005-06-06

OCL complements

Undefined Values (p 30)

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.

Implicit conversion: element -> singleton

e.g., e -> p ⇔ Set{e} -> p

Abbreviation: s->collect(p) ⇔ s.p + flattening (but usually it's a bag)

collectNested preserves the structure

Iterate:

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

iterator accumulator initial value body, evaluated for each elem

for a sequence the order is the order of the elements in the sequence.

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.

7.5.11 Collection Literals

Sets, Sequences, and Bags can be specified by a literal in OCL. Curly brackets surround the elements of the collection, elements in the collection are written within, separated by commas. The type of the collection is written before the curly brackets:

```
Set { 1 , 2 , 5 , 88 }
Set { 'apple' , 'orange', 'strawberry' }
```

A Sequence:

```
Sequence { 1, 3, 45, 2, 3 }
Sequence { 'ape', 'nut' }
```

A bag:

```
Bag {1 , 3 , 4, 3, 5 }
```

Because of the usefulness of a Sequence of consecutive Integers, there is a separate literal to create them. The elements inside the curly brackets can be replaced by an interval specification, which consists of two expressions of type Integer, Int-expr1 and Int-expr2, separated by '..'. This denotes all the Integers between the values of Int-expr1 and Int-expr2, including the values of Int-expr1 and Int-expr2 themselves:

```
Sequence{ 1..(6 + 4) }
Sequence{ 1..10 }
-- are both identical to
Sequence{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```

The complete list of Collection operations is described in chapter 11 ("The OCL Standard Library"). Collections can be specified by a literal, as described above. The only other way to get a collection is by navigation. To be more precise, the only way to get a Set, OrderedSet, Sequence, or Bag is:

1. a literal, this will result in a Set, OrderedSet, Sequence, or Bag:

   ```
   Set {2 , 4, 1 , 5 , 7 , 13, 11, 17 }
   OrderedSet {1 , 2, 3 , 5 , 7 , 11, 13, 17 }
   Sequence {1 , 2, 3 , 5 , 7 , 11, 13, 17 }
   Bag {1, 2, 3, 2, 1}
   ```

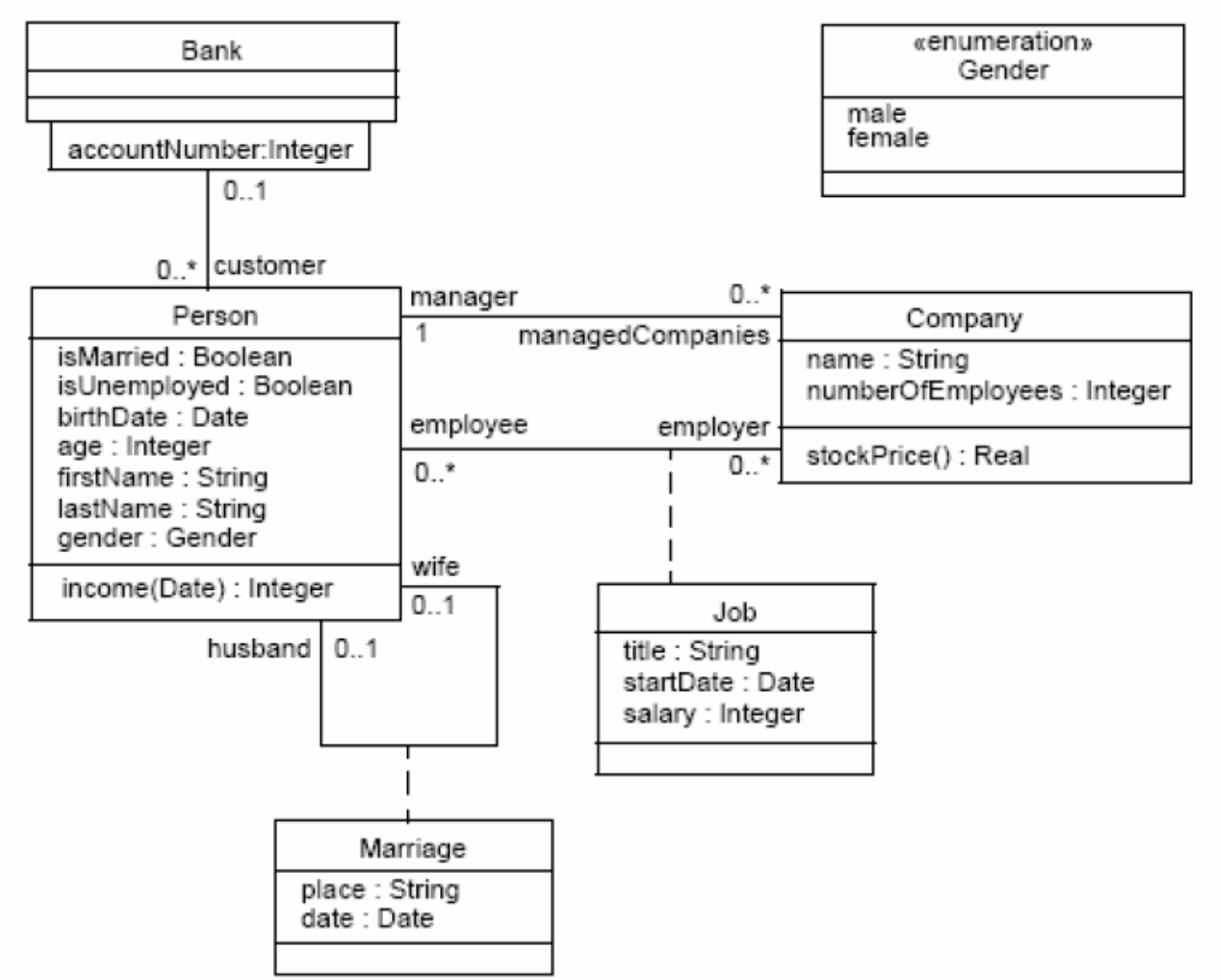
2. a navigation starting from a single object can result in a collection:

   ```
   context Company inv:
       self.employee
   ```

3. operations on collections may result in new collections:

   ```
   collection1->union(collection2)
   ```

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)
```

11 The OCL Standard Library

This section describes the OCL Standard Library of predefined types, their operations, and predefined expression templates in the OCL. This section contains all standard types defined within OCL, including all the operations defined on those types. For each operation the signature and a description of the semantics is given. Within the description, the reserved word 'result' is used to refer to the value that results from evaluating the operation. In several places, post conditions are used to describe properties of the result. When there is more than one postcondition, all postconditions must be true. A similar thing is true for multiple preconditions. If these are used, the operation is only defined if all preconditions evaluate to true.

11.1 Introduction

The structure, syntax and semantics of the OCL is defined in chapters 8 ("Abstract Syntax"), 9 ("Concrete Syntax") and 10 ("Semantics Described using UML"). This section adds another part to the OCL definition: a library of predefined types and operations. Any implementation of OCL must include this library package. This approach has also been taken by e.g. the Java definition, where the language definition and the standard libraries are both mandatory parts of the complete language definition.

The OCL standard library defines a number of types. It includes several primitive types: Integer, Real, String and Boolean. These are familiar from many other languages. The second part of the standard library consists of the collection types. They are Bag, Set, Sequence and Collection, where Collection is an abstract type. Note that all types defined in the OCL standard library are instances of an abstract syntax class. The OCL standard library exists at the modeling level, also referred to as the M1 level, where the abstract syntax is the metalevel or M2 level.

Issue 6012: ModelPropertyCallExp renamed FeatureCallExp

Next to definitions of types the OCL standard library defines a number of template expressions. Many operations defined on collections, map not on the abstract syntax metaclass FeatureCallExp, but on the IteratorExp. For each of these a template expression that defines the name and format of the expression, is defined in Section 11.8 ("Predefined Iterator Expressions").

Issue 5972: Adding OclUndefined and updating OclVoid

11.2 The OclAny, OclVoid, OclInvalid and OclMessage types

11.2.1 OclAny

Issue 8791: Replace inheritance at M1 level by compliamce

All types in the UML model and the primitive types in the OCL standard library comply with the type OclAny. Conceptually, OclAny behaves as a supertype for all the types except for the OCL pre-defined collection types. Features of OclAny are available on each object in all OCL expressions. OclAny is itself an instance of the metatype AnyType.

Issue 8791: Removed figure showing specific inheritance links at M1 level

All classes in a UML model inherit all operations defined on OclAny. To avoid name conflicts between properties in the model and the properties inherited from OclAny, all names on the properties of OclAny start with 'ocl.' Although theoretically there may still be name conflicts, they can be avoided. One can also use the oclAsType() operation to explicitly refer to the OclAny properties.

Operations of OclAny, where the instance of OclAny is called *object*.

11.2.2 OclMessage

This section contains the definition of the standard type *OclMessage*. As defined in this section, each ocl message type is actually a template type with one parameter. 'T' denotes the parameter. A concrete ocl message type is created by substituting an operation or signal for the T.

The predefined type *OclMessage* is an instance of *MessageType*. Every *OclMessage* is fully determined by either the operation, or signal given as parameter. Note that there is conceptually an undefined (infinite) number of these types, as each is determined by a different operation or signal. These types are unnamed. Every type has as attributes the name of the operation or signal, and either all formal parameters of the operation, or all attributes of the signal. OclMessage is itself an instance of the metatype MessageType.

OclMessage has a number of predefined operations, as shown in the OCL Standard Library.

11.2.3 OclVoid

Issue 5972 : OclVoid changed and OclInvalid added

The type OclVoid is a type that conforms to all other types. It has one single instance called *null* which corresponds with the UML NullLiteral value specification. Any property call applied on *null* results in OclInvalid, except for the operation oclIsUndefined(). OclVoid is itself an instance of the metatype VoidType.

11.2.4 OclInvalid

The type OclInvalid is a type that conforms to all other types. It has one single instance called *invalid.* Any property call applied on *invalid* results in OclInvalid, except for the operations oclIsUndefined() and oclIsInvalid(). OclInvalid is itself an instance of the metatype InvalidType

11.2.5 Operations and well-formedness rules

OclAny

= (object2 : OclAny) : Boolean

True if *self* is the same object as *object2*. Infix operator.

post: result = (self = object2)

<> (object2 : OclAny) : Boolean

True if *self* is a different object from *object2*. Infix operator.

post: result = not (self = object2)

oclIsNew() : Boolean

Can only be used in a postcondition. Evaluates to true if the *self* is created during performing the operation. I.e. it didn't exist at precondition time.

post: self@pre.oclIsUndefined()

Issue 5972: Update OclUndefined and add oclIsInvalid

oclIsUndefined() : Boolean

Evaluates to true if the *self* is equal to OclInvalid or equal to null.

post: result = self.isTypeOf(OclVoid) or self.isTypeOf(OclInvalid)

oclIsInvalid() : Boolean

Evaluates to true if the *self* is equal to OclInvalid

post: result = self.isTypeOf(OclInvalid)

Issue 6531: Enumerations are not used anymore: typename becomes typespec

oclAsType(typespec : OclType) : T

Evaluates to *self*, where *self* is of the type identified by typespec.

post: (result = self) and result.oclIsTypeOf(typeName)

oclIsTypeOf(typespec : OclType) : Boolean

Evaluates to true if the *self* is of the type identified by typespec. .

post: -- TBD

oclIsKindOf(typespec : OclType) : Boolean

Evaluates to true if the *self* conforms to the type identified by typespec.

post: -- TBD

oclIsInState(statespec : OclState) : Boolean

Evaluates to true if the *self* is in the state indentified by statespec.

post: -- TBD

allInstances() : Set(T)

Returns all instances of self. Type T is equal to self. May only be used for classifiers that have a finite number of instances. This is the case for, for instance, user defined classes because instances need to be created explicitly. This is not the case for, for instance, the standard String, Integer, and Real types.

pre: self.isKindOf(Classifier) -- self must be a Classifier
and -- TBD -- self must have a finite number of instances
-- it depends on the UML 2.0 metamodel how this can be
-- expressed

post: -- TBD

11.2.6 OclMessage

hasReturned() : Boolean

True if type of template parameter is an operation call, and the called operation has returned a value. This implies the fact that the message has been sent. False in all other cases.

post: --

result() : <<*The return type of the called operation*>>

Returns the result of the called operation, if type of template parameter is an operation call, and the called operation has returned a value. Otherwise the undefined value is returned.

pre: hasReturned()

isSignalSent() : Boolean

Returns true if the OclMessage represents the sending of a UML Signal.

isOperationCall() : Boolean

Returns true if the OclMessage represents the sending of a UML Operation call.

Issue 5972: Remove oclIsUndefined since already defined in AnyType

Issue 6531:Special types instead of Model element types

11.3 Special types

This section defines several types that qre used to formalize the signature of pre-defined operations manipulating type and elements defined in the UML model.

11.3.1 OclElement

The singleton instance of ElementType.

11.3.2 OclType

The singleton instance of TypeType.

11.3.3 Operations and well-formedness rules

This section contains thye operatiins and well-formedness rules of the model element types.

OclElement

= (object : OclType) : Boolean

True if *self* is the same object as *object*.

<> (object : OclType) : Boolean

True if *self* is a different object from *object*.

post: result = not (self = object)

OclType

= (object : OclType) : Boolean

True if *self* is the same object as *object*.

<> (object : OclType) : Boolean

True if *self* is a different object from *object*.

post: result = not (self = object)

11.4 Primitive Types

The primitive types defined in the OCL standard library are Integer, Real, String and Boolean. They are all instance of the metaclass Primitive from the UML core package.

11.4.1 Real

The standard type Real represents the mathematical concept of real. Note that Integer is a subclass of Real, so for each parameter of type Real, you can use an integer as the actual parameter. Real is itself an instance of the metatype PrimitiveType (from UML).

11.4.2 Integer

The standard type Integer represents the mathematical concept of integer. Integer is itself an instance of the metatype PrimitiveType (from UML).

11.4.3 String

The standard type String represents strings, which can be both ASCII or Unicode. String is itself an instance of the metatype PrimitiveType (from UML).

11.4.4 Boolean

The standard type Boolean represents the common true/false values. Boolean is itself an instance of the metatype PrimitiveType (from UML).

11.4.5 UnlimitedInteger

The standard type UnlimitedInteger is used to encode the upper value of a multiplicity specification. UnlimitedInteger is itself an instance of the metatype UnlimitedIntegerType.

11.5 Operations and well-formedness rules

This section contains the operatiins and well-formedness rules of the primitive types.

11.5.1 Real

Note that Integer is a subclass of Real, so for each parameter of type Real, you can use an integer as the actual parameter.

+ (r : Real) : Real

The value of the addition of *self* and *r*.

- (r : Real) : Real

The value of the subtraction of *r* from *self*.

*** (r : Real) : Real**

The value of the multiplication of *self* and *r*.

- : Real

The negative value of *self*.

/ (r : Real) : Real

The value of *self* divided by *r*. Evaluates to *OclInvalid* if r is equal to zero.

abs() : Real

The absolute value of *self*.

post: if self < 0 then result = - self else result = self endif

floor() : Integer

The largest integer which is less than or equal to *self*.

post: (result <= self) and (result + 1 > self)

round() : Integer

The integer which is closest to *self*. When there are two such integers, the largest one.

post: ((self - result).abs() < 0.5) or ((self - result).abs() = 0.5 and (result > self))

max(r : Real) : Real

The maximum of *self* and *r*.

post: if self >= r then result = self else result = r endif

min(r : Real) : Real

The minimum of *self* and *r*.

post: if self <= r then result = self else result = r endif

< (r : Real) : Boolean

True if *self* is less than *r*.

> (r : Real) : Boolean

True if *self* is greater than *r*.

post: result = not (self <= r)

<= (r : Real) : Boolean

True if *self* is less than or equal to *r*.

post: result = ((self = r) or (self < r))

>= (r : Real) : Boolean

True if *self* is greater than or equal to *r*.

post: result = ((self = r) or (self > r))

11.5.2 Integer

- : Integer

The negative value of *self*.

+ (i : Integer) : Integer

The value of the addition of *self* and *i*.

- (i : Integer) : Integer

The value of the subtraction of *i* from *self*.

*** (i : Integer) : Integer**

The value of the multiplication of *self* and *i*.

/ (i : Integer) : Real

The value of *self* divided by *i*.Evaluates to *OclInvalid* if r is equal to zero

abs() : Integer

The absolute value of *self*.

post: if self < 0 then result = - self else result = self endif

div(i : Integer) : Integer

The number of times that *i* fits completely within *self*.

```
pre : i <> 0
post: if self / i >= 0 then result = (self / i).floor()
                else result = -((-self/i).floor())
    endif
```

mod(i : Integer) : Integer

The result is *self* modulo *i*.

post: result = self - (self.div(i) * i)

max(i : Integer) : Integer

The maximum of *self* an *i*.

post: if self >= i then result = self else result = i endif

min(i : Integer) : Integer

The minimum of *self* an *i*.

post: if self <= i then result = self else result = i endif

11.5.3 String

size() : Integer

The number of characters in *self*.

concat(s : String) : String

The concatenation of *self* and *s*.

```
post: result.size() = self.size() + string.size()
post: result.substring(1, self.size() ) = self
```

post: result.substring(self.size() + 1, result.size()) = s

substring(lower : Integer, upper : Integer) : String

The sub-string of *self* starting at character number *lower*, up to and including character number *upper*. Character numbers run from 1 to *self.size()*.

pre: 1 <= lower

pre: lower <= upper

pre: upper <= self.size()

toInteger() : Integer

Converts *self* to an Integer value.

toReal() : Real

Converts *self* to a Real value.

11.5.4 Boolean

or (b : Boolean) : Boolean

True if either *self* or *b* is true.

xor (b : Boolean) : Boolean

True if either *self* or *b* is true, but not both.

post: (self or b) and not (self = b)

and (b : Boolean) : Boolean

True if both *b1* and *b* are true.

not : Boolean

True if *self* is false.

post: if self then result = false else result = true endif

implies (b : Boolean) : Boolean

True if *self* is false, or if *self* is true and *b* is true.

post: (not self) or (self and b)

11.6 Collection-Related Types

This section defines the collection types and their operations. As defined in this section, each collection type is actually a template type with one parameter. 'T' denotes the parameter. A concrete collection type is created by substituting a type for the T. So Set (Integer) and Bag (Person) are collection types.

11.6.1 Collection

Collection is the abstract supertype of all collection types in the OCL Standard Library. Each occurrence of an object in a collection is called an *element*. If an object occurs twice in a collection, there are two elements. This section defines the properties on Collections that have identical semantics for all collection subtypes. Some operations may be defined within the

subtype as well, which means that there is an additional postcondition or a more specialized return value. Collection is itself an instance of the metatype CollectionType.

The definition of several common operations is different for each subtype. These operations are not mentioned in this section.

The semantics of the collection operations is given in the form of a postcondtion that uses the *IterateExp* ot the *IteratorExp* construct. The semantics of those constructs is defined in chapter 10 ("Semantics Described using UML"). In several cases the postcondtion refers to other collection operations, which in turn are defined in terms of the *IterateExp* or *IteratorExp* constructs.

11.6.2 Set

The Set is the mathematical set. It contains elements without duplicates. Set is itself an instance of the metatype SetType.

11.6.3 OrderedSet

The OrderedSet is a Set the elements of which are ordered. It contains no duplicates. OrderedSet is itself an instance of the metatype OrderedSetType.

11.6.4 Bag

A bag is a collection with duplicates allowed. That is, one object can be an element of a bag many times. There is no ordering defined on the elements in a bag. Bag is itself an instance of the metatype BagType.

11.6.5 Sequence

A sequence is a collection where the elements are ordered. An element may be part of a sequence more than once. Sequence is itself an instance of the metatype SequenceType.

11.7 Operations and well-formedness rules

This section contains the operations and well-formedness rules of the collection types.

11.7.1 Collection

size() : Integer

The number of elements in the collection *self*.

post: result = self->iterate(elem; acc : Integer = 0 | acc + 1)

includes(object : T) : Boolean

True if *object* is an element of *self*, false otherwise.

post: result = (self->count(object) > 0)

excludes(object : T) : Boolean

True if *object* is not an element of *self*, false otherwise.

post: result = (self->count(object) = 0)

count(object : T) : Integer

The number of times that *object* occurs in the collection *self*.

post: result = self->iterate(elem; acc : Integer = 0 |
if elem = object then acc + 1 else acc endif)

includesAll(c2 : Collection(T)) : Boolean

Does *self* contain all the elements of *c2* ?

post: result = c2->forAll(elem | self->includes(elem))

excludesAll(c2 : Collection(T)) : Boolean

Does *self* contain none of the elements of *c2* ?

post: result = c2->forAll(elem | self->excludes(elem))

isEmpty() : Boolean

Is *self* the empty collection?

post: result = (self->size() = 0)

notEmpty() : Boolean

Is *self* not the empty collection?

post: result = (self->size() <> 0)

sum() : T

The addition of all elements in *self*. Elements must be of a type supporting the + operation. The + operation must take one parameter of type T and be both associative: (a+b)+c = a+(b+c), and commutative: a+b = b+a. Integer and Real fulfill this condition.

post: result = self->iterate(elem; acc : T = 0 | acc + elem)

product(c2: Collection(T2)) : Set(Tuple(first: T, second: T2))

The cartesian product operation of *self* and *c2*.

```
post: result = self->iterate (e1; acc: Set(Tuple(first: T, second: T2)) = Set{} |
                  c2->iterate (e2; acc2: Set(Tuple(first: T, second: T2)) = acc |
                    acc2->including (Tuple{first = e1, second = e2}) ) )
```

11.7.2 Set

union(s : Set(T)) : Set(T)

The union of *self* and *s*.

post: result->forAll(elem | self->includes(elem) or s->includes(elem))
post: self ->forAll(elem | result->includes(elem))
post: s ->forAll(elem | result->includes(elem))

union(bag : Bag(T)) : Bag(T)

The union of *self* and *bag*.

post: result->forAll(elem | result->count(elem) = self->count(elem) + bag->count(elem))
post: self->forAll(elem | result->includes(elem))
post: bag ->forAll(elem | result->includes(elem))

= (s : Set(T)) : Boolean

Evaluates to true if *self* and *s* contain the same elements.

post: result = (self->forAll(elem | s->includes(elem)) and
s->forAll(elem | self->includes(elem)))

intersection(s : Set(T)) : Set(T)

The intersection of *self* and *s* (i.e, the set of all elements that are in both *self* and *s*).

post: result->forAll(elem | self->includes(elem) and s->includes(elem))
post: self->forAll(elem | s ->includes(elem) = result->includes(elem))
post: s ->forAll(elem | self->includes(elem) = result->includes(elem))

intersection(bag : Bag(T)) : Set(T)

The intersection of *self* and *bag*.

post: result = self->intersection(bag->asSet)

– (s : Set(T)) : Set(T)

The elements of *self*, which are not in *s*.

post: result->forAll(elem | self->includes(elem) and s->excludes(elem))
post: self ->forAll(elem | result->includes(elem) = s->excludes(elem))

including(object : T) : Set(T)

The set containing all elements of *self* plus *object*.

post: result->forAll(elem | self->includes(elem) or (elem = object))
post: self- >forAll(elem | result->includes(elem))
post: result->includes(object)

excluding(object : T) : Set(T)

The set containing all elements of *self* without *object*.

post: result->forAll(elem | self->includes(elem) and (elem <> object))
post: self- >forAll(elem | result->includes(elem) = (object <> elem))
post: result->excludes(object)

symmetricDifference(s : Set(T)) : Set(T)

The sets containing all the elements that are in *self* or *s*, but not in both.

post: result->forAll(elem | self->includes(elem) xor s->includes(elem))
post: self->forAll(elem | result->includes(elem) = s ->excludes(elem))
post: s ->forAll(elem | result->includes(elem) = self->excludes(elem))

count(object : T) : Integer

The number of occurrences of *object* in *self*.

post: result <= 1

flatten() : Set(T2)

If the element type is not a collection type this result in the same *self*. If the element type is a collection type, the result is the set

containing all the elements of all the elements of *self*.

```
post: result = if self.type.elementType.oclIsKindOf(CollectionType) then
                 self->iterate(c; acc : Set() = Set{} |
                     acc->union(c->asSet() ) )
               else
                 self
               endif
```

asSet() : Set(T)

A Set identical to self. This operation exists for convenience reasons.

```
post: result = self
```

asOrderedSet() : OrderedSet(T)

An OrderedSet that contains all the elements from *self,* in undefined order.

```
post: result->forAll(elem | self->includes(elem))
```

asSequence() : Sequence(T)

A Sequence that contains all the elements from *self,* in undefined order.

```
post: result->forAll(elem | self->includes(elem))
post: self->forAll(elem | result->count(elem) = 1)
```

asBag() : Bag(T)

The Bag that contains all the elements from *self*.

```
post: result->forAll(elem | self->includes(elem))
post: self->forAll(elem | result->count(elem) = 1)
```

11.7.3 OrderedSet

append (object: T) : OrderedSet(T)

The set of elements, consisting of all elements of *self*, followed by *object*.

```
post: result->size() = self->size() + 1
post: result->at(result->size() ) = object
post:   Sequence{1..self->size() }->forAll(index : Integer |
        result->at(index) = self ->at(index))
```

prepend(object : T) : OrderedSet(T)

The sequence consisting of *object*, followed by all elements in *self*.

```
post: result->size = self->size() + 1
post: result->at(1) = object
post:   Sequence{1..self->size()}->forAll(index : Integer |
       self->at(index) = result->at(index + 1))
```

insertAt(index : Integer, object : T) : OrderedSet(T)

The set consisting of *self* with *object* inserted at position *index*.

```
post: result->size = self->size() + 1
```

```
post: result->at(index) = object
post: Sequence{1..(index - 1)}->forAll(i : Integer |
      self->at(i) = result->at(i))
post: Sequence{(index + 1)..self->size()}->forAll(i : Integer |
      self->at(i) = result->at(i + 1))
```

subOrderedSet(lower : Integer, upper : Integer) : OrderedSet(T)

The sub-set of *self* starting at number *lower*, up to and including element number *upper*.

```
pre : 1 <= lower
pre : lower <= upper
pre : upper <= self->size()
post: result->size() = upper -lower + 1
post: Sequence{lower..upper}->forAll( index |
      result->at(index - lower + 1) =
            self->at(index))
```

at(i : Integer) : T

The *i-th* element of *self*.

```
pre : i >= 1 and i <= self->size()
```

indexOf(obj : T) : Integer

The index of object *obj* in the sequence.

```
pre  : self->includes(obj)
post : self->at(i) = obj
```

first() : T

The first element in *self*.

```
post: result = self->at(1)
```

last() : T

The last element in *self*.

```
post: result = self->at(self->size() )
```

11.7.4 Bag

= (bag : Bag(T)) : Boolean

True if *self* and *bag* contain the same elements, the same number of times.

```
post: result = (self->forAll(elem | self->count(elem) = bag->count(elem)) and
            bag->forAll(elem | bag->count(elem) = self->count(elem)) )
```

union(bag : Bag(T)) : Bag(T)

The union of *self* and *bag*.

```
post: result->forAll( elem | result->count(elem) = self->count(elem) + bag->count(elem))
post: self  ->forAll( elem | result->count(elem) = self->count(elem) + bag->count(elem))
```

post: bag ->forAll(elem | result->count(elem) = self->count(elem) + bag->count(elem))

union(set : Set(T)) : Bag(T)

The union of *self* and *set*.

post: result->forAll(elem | result->count(elem) = self->count(elem) + set->count(elem))
post: self ->forAll(elem | result->count(elem) = self->count(elem) + set->count(elem))
post: set ->forAll(elem | result->count(elem) = self->count(elem) + set->count(elem))

intersection(bag : Bag(T)) : Bag(T)

The intersection of *self* and *bag*.

```
post: result->forAll(elem |
    result->count(elem) = self->count(elem).min(bag->count(elem)) )
post: self->forAll(elem |
    result->count(elem) = self->count(elem).min(bag->count(elem)) )
post: bag->forAll(elem |
    result->count(elem) = self->count(elem).min(bag->count(elem)) )
```

intersection(set : Set(T)) : Set(T)

The intersection of *self* and *set.*

post: result->forAll(elem|result->count(elem) = self->count(elem).min(set->count(elem)))
post: self ->forAll(elem|result->count(elem) = self->count(elem).min(set->count(elem)))
post: set ->forAll(elem|result->count(elem) = self->count(elem).min(set->count(elem)))

including(object : T) : Bag(T)

The bag containing all elements of *self* plus *object*.

```
post: result->forAll(elem |
      if elem = object then
        result->count(elem) = self->count(elem) + 1
      else
        result->count(elem) = self->count(elem)
      endif)
post: self->forAll(elem |
      if elem = object then
        result->count(elem) = self->count(elem) + 1
      else
        result->count(elem) = self->count(elem)
      endif)
```

excluding(object : T) : Bag(T)

The bag containing all elements of self apart from all occurrences of *object*.

```
post: result->forAll(elem |
      if elem = object then
        result->count(elem) = 0
      else
        result->count(elem) = self->count(elem)
```

```
        endif)
    post: self->forAll(elem |
        if elem = object then
          result->count(elem) = 0
        else
          result->count(elem) = self->count(elem)
        endif)
```

count(object : T) : Integer

The number of occurrences of *object* in *self*.

flatten() : Bag(T2)

If the element type is not a collection type this result in the same bag. If the element type is a collection type, the result is the bag containing all the elements of all the elements of *self*.

```
post: result = if self.type.elementType.oclIsKindOf(CollectionType) then
            self->iterate(c; acc : Bag() = Bag{} |
                acc->union(c->asBag() ) )
          else
            self
          endif
```

asBag() : Bag(T)

A Bag identical to *self*. This operation exists for convenience reasons.

```
post: result = self
```

asSequence() : Sequence(T)

A Sequence that contains all the elements from *self*, in undefined order.

```
post: result->forAll(elem | self->count(elem) = result->count(elem))
post: self  ->forAll(elem | self->count(elem) = result->count(elem))
```

asSet() : Set(T)

The Set containing all the elements from *self*, with duplicates removed.

```
post: result->forAll(elem | self  ->includes(elem))
post: self  ->forAll(elem | result->includes(elem))
```

asOrderedSet() : OrderedSet(T)

An OrderedSet that contains all the elements from *self,* in undefined order, with duplicates removed.

```
post: result->forAll(elem | self  ->includes(elem))
post: self  ->forAll(elem | result->includes(elem))
post: self  ->forAll(elem | result->count(elem) = 1)
```

11.7.5 Sequence

count(object : T) : Integer

The number of occurrences of *object* in *self*.

= (s : Sequence(T)) : Boolean

True if *self* contains the same elements as *s* in the same order.

```
post: result = Sequence{1..self->size()}->forAll(index : Integer |
                    self->at(index) = s->at(index))
                    and
                    self->size() = s->size()
```

union (s : Sequence(T)) : Sequence(T)

The sequence consisting of all elements in *self*, followed by all elements in *s*.

```
post: result->size() = self->size() + s->size()
post: Sequence{1..self->size()}->forAll(index : Integer |
                                    self->at(index) = result->at(index))
post: Sequence{1..s->size()}->forAll(index : Integer |
                                s->at(index) =  result->at(index + self->size() )))
```

flatten() : Sequence(T2)

If the element type is not a collection type this result in the same *self*. If the element type is a collection type, the result is the seuqnce containing all the elements of all the elements of *self*. The order of the elements is partial.

```
post: result = if self.type.elementType.oclIsKindOf(CollectionType) then
          self->iterate(c; acc : Sequence() = Sequence{} |
              acc->union(c->asSequence() ) )
        else
          self
        endif
```

append (object: T) : Sequence(T)

The sequence of elements, consisting of all elements of *self*, followed by *object*.

```
post: result->size() = self->size() + 1
post: result->at(result->size() ) = object
post:   Sequence{1..self->size() }->forAll(index : Integer |
      result->at(index) = self ->at(index))
```

prepend(object : T) : Sequence(T)

The sequence consisting of *object*, followed by all elements in *self*.

```
post: result->size = self->size() + 1
post: result->at(1) = object
post:   Sequence{1..self->size()}->forAll(index : Integer |
     self->at(index) = result->at(index + 1))
```

insertAt(index : Integer, object : T) : Sequence(T)

The sequence consisting of *self* with *object* inserted at position *index*.

```
post: result->size = self->size() + 1
post: result->at(index) = object
post: Sequence{1..(index - 1)}->forAll(i : Integer |
     self->at(i) = result->at(i))
```

```
post: Sequence{(index + 1)..self->size()}->forAll(i : Integer |
      self->at(i) = result->at(i + 1))
```

subSequence(lower : Integer, upper : Integer) : Sequence(T)

The sub-sequence of *self* starting at number *lower*, up to and including element number *upper*.

```
pre : 1 <= lower
pre : lower <= upper
pre : upper <= self->size()
post: result->size() = upper -lower + 1
post: Sequence{lower..upper}->forAll( index |
      result->at(index - lower + 1) =
                  self->at(index))
```

at(i : Integer) : T

The *i-th* element of sequence.

```
pre : i >= 1 and i <= self->size()
```

indexOf(obj : T) : Integer

The index of object *obj* in the sequence.

```
pre  : self->includes(obj)
post : self->at(i) = obj
```

first() : T

The first element in *self*.

```
post: result = self->at(1)
```

last() : T

The last element in *self*.

```
post: result = self->at(self->size() )
```

including(object : T) : Sequence(T)

The sequence containing all elements of *self* plus *object* added as the last element.

```
post: result = self.append(object)
```

excluding(object : T) : Sequence(T)

The sequence containing all elements of *self* apart from all occurrences of *object*.

The order of the remaining elements is not changed.

```
post:result->includes(object) = false
post: result->size() = self->size() - self->count(object)
post: result = self->iterate(elem; acc : Sequence(T)
   = Sequence{}|
      if elem = object then acc else acc->append(elem) endif )
```

asBag() : Bag(T)

The Bag containing all the elements from *self*, including duplicates.

post: result->forAll(elem | self->count(elem) = result->count(elem))

post: self->forAll(elem | self->count(elem) = result->count(elem))

asSequence() : Sequence(T)

The Sequence identical to the object itself. This operation exists for convenience reasons.

post: result = self

asSet() : Set(T)

The Set containing all the elements from *self*, with duplicated removed.

post: result->forAll(elem | self ->includes(elem))

post: self ->forAll(elem | result->includes(elem))

asOrderedSet() : OrderedSet(T)

An OrderedSet that contains all the elements from *self,* in the same order, with duplicates removed.

post: result->forAll(elem | self ->includes(elem))

post: self ->forAll(elem | result->includes(elem))

post: self ->forAll(elem | result->count(elem) = 1)

post: self ->forAll(elem1, elem2 |

self->indexOf(elem1) < self->indexOf(elem2)

implies result->indexOf(elem1) < result->indexOf(elem2))

11.8 Predefined Iterator Expressions

This section defines the standard OCL iterator expressions. In the abstract syntax these are all instances of IteratorExp. These iterator expressions always have a collection expression as their source, as is defined in the well-formedness rules in Chapter 8 (“Abstract Syntax”). The defined iterator expressions are shown per source collection type. The semantics of each iterator expression is defined through a mapping from the iterator to the ’*iterate*’ construct. this means that the semantics of the iterator expressions does not need to be defined seperately in the sementics sections.

Whenever a new itertor is added to the library, the mapping to the *iterate* expression must be defined. If this is not done, the semantics of the new iterator is undefined.

In all of the following OCL expressions, the lefthand side of the equals sign is the *IteratorExp* to be defined, and the righthand side of the equals sign is the equivalent as an *IterateExp*. The names *source*, *body* and *iterator* refer to the role names in the abstract syntax:

- source — The source expression of the IteratorExp
- body — The body expression of the IteratorExp
- iterator — The iterator variable of the IteratorExp
- result — The result variable of the IterateExp

11.8.1 Extending the standard library with iterator expressions

When new iterator expressions are added to the standard library, there mapping to existing constructs should be fully defines. If this is done, the semantics of the new iterator expression will be defined.

11.9 Mapping rules for predefined iterator expressions

This section contains the operations and well-formedness rules of the collection types.

11.9.1 Collection

exists

Results in true if *body* evaluates to true for at least one element in the *source* collection.

```
source->exists(iterators | body) =
        source->iterate(iterators; result : Boolean = false | result or body)
```

forAll

Results in true if the *body* expression evaluates to true for each element in the *source* collection; otherwise, result is false.

```
source->forAll(iterators | body ) =
        source->iterate(iterators; result : Boolean = true | result and body)
```

isUnique

Results in true if *body* evaluates to a different value for each element in the *source* collection; otherwise, result is false.

```
source->isUnique (iterators | body) =
  source->collect (iterators | Tuple{iter = Tuple{iterators}, value = body})
    ->forAll (x, y | (x.iter <> y.iter) implies (x.value <> y.value))
```

isUnique may have at most one iterator variable.

any

Returns any element in the *source* collection for which *body* evaluates to true. If there is more than one element for which *body* is true, one of them is returned. There must be at least one element fulfilling *body*, otherwise the result of this IteratorExp is null.

```
source->any(iterator | body) =
    source->select(iterator | body)->asSequence()->first()
```

any may have at most one iterator variable.

one

Results in true if there is exactly one element in the *source* collection for which *body* is true.

```
source->one(iterator | body) =
    source->select(iterator | body)->size() = 1
```

one may have at most one iterator variable.

collect

The Collection of elements which results from applying *body* to every member of the *source* set. The result is flattened. Notice that this is based on *collectNested*, which can be of different type depending on the type of *source*. *collectNested* is defined individually for each subclass of *CollectionType*.

```
source->collect (iterators | body) = source->collectNested (iterators | body)->flatten()
```

collect may have at most one iterator variable.

11.9.2 Set

The standard iterator expression with source of type Set(T) are:

select

The subset of *set* for which *expr* is true.

```
source->select(iterator | body) =
      source->iterate(iterator; result : Set(T) = Set{} |
                  if body then result->including(iterator)
                       else result
                  endif)
```

select may have at most one iterator variable.

reject

The subset of the *source* set for which *body* is false.

```
source->reject(iterator | body) =
      source->select(iterator | not body)
```

reject may have at most one iterator variable.

collectNested

The Bag of elements which results from applying *body* to every member of the *source* set.

```
source->collect(iterators | body) =
      source->iterate(iterators; result : Bag(body.type) = Bag{} |
                  result->including(body ) )
```

collectNested may have at most one iterator variable.

sortedBy

Results in the OrderedSet containing all elements of the *source* collection. The element for which *body* has the lowest value comes first, and so on. The type of the *body* expression must have the < operation defined. The < operation must return a Boolean value and must be transitive i.e. if a < b and b < c then a < c.

```
source->sortedBy(iterator | body) =
   iterate( iterator ; result : OrderedSet(T) : OrderedSet {} |
      if result->isEmpty() then
        result.append(iterator)
      else
        let position : Integer = result->indexOf (
                  result->select (item | body (item) > body (iterator)) ->first() )
        in
          result.insertAt(position, iterator)
      endif
```

sortedBy may have at most one iterator variable.

11.9.3 Bag

The standard iterator expression with source of type Bag(T) are:

select

The sub-bag of the *source* bag for which *body* is true.

```
source->select(iterator | body) =
    source->iterate(iterator; result : Bag(T) = Bag{} |
            if body then result->including(iterator)
                else result
            endif)
```

select may have at most one iterator variable.

reject

The sub-bag of the *source* bag for which *body* is false.

```
source->reject(iterator | body) =
    source->select(iterator | not body)
```

reject may have at most one iterator variable.

collectNested

The Bag of elements which results from applying *body* to every member of the *source* bag.

```
source->collect(iterators | body) =
    source->iterate(iterators; result : Bag(body.type) = Bag{} |
            result->including(body ) )
```

collectNested may have at most one iterator variable.

sortedBy

Results in the Sequence containing all elements of the *source* collection. The element for which *body* has the lowest value comes first, and so on. The type of the *body* expression must have the < operation defined. The < operation must return a Boolean value and must be transitive i.e. if a < b and b < c then a < c.

```
source->sortedBy(iterator | body) =
  iterate( iterator ; result : Sequence(T) : Sequence {} |
    if result->isEmpty() then
      result.append(iterator)
    else
      let position : Integer = result->indexOf (
            result->select (item | body (item) > body (iterator)) ->first() )
      in
        result.insertAt(position, iterator)
    endif
```

sortedBy may have at most one iterator variable.

11.9.4 Sequence

The standard iterator expressions with source of type Sequence(T) are:

select(expression : OclExpression) : Sequence(T)

The subsequence of the *source* sequence for which *body* is *true*.

```
source->select(iterator | body) =
    source->iterate(iterator; result : Sequence(T) = Sequence{} |
            if body then result->including(iterator)
                else result
            endif)
```

select may have at most one iterator variable.

reject

The subsequence of the *source* sequence for which *body* is false.

```
source->reject(iterator | body) =
    source->select(iterator | not body)
```

reject may have at most one iterator variable.

collectNested

The Sequence of elements which results from applying *body* to every member of the *source* sequence.

```
source->collect(iterators | body) =
    source->iterate(iterators; result : Sequence(body.type) = Sequence{} |
            result->append(body ) )
```

collectNested may have at most one iterator variable.

sortedBy

Results in the Sequence containing all elements of the *source* collection. The element for which *body* has the lowest value comes first, and so on. The type of the *body* expression must have the < operation defined. The < operation must return a Boolean value and must be transitive i.e. if a < b and b < c then a < c.

```
source->sortedBy(iterator | body) =
  iterate( iterator ; result : Sequence(T) : Sequence {} |
    if result->isEmpty() then
      result.append(iterator)
    else
      let position : Integer = result->indexOf (
            result->select (item | body (item) > body (iterator)) ->first() )
      in
        result.insertAt(position, iterator)
    endif
```

sortedBy may have at most one iterator variable.