Object Constraint Language Specification 6

Note – Changes based on the ISO version of UML 1.4.1 (formal/03-02-04) are in this font.

This chapter introduces and defines the Object Constraint Language (OCL), a formal language to express side-effect-free constraints.

Contents

This chapter contains the following topics.

6 Object Constraint Language Specification

6.1 Overview

This chapter introduces and defines the Object Constraint Language (OCL), a formal language used to express constraints. These typically specify invariant conditions that must hold for the system being modeled. Note that when the OCL expressions are evaluated, they do not have side effects; that is, their evaluation cannot alter the state of the corresponding executing system. In addition, to specifying invariants of the UML metamodel, UML modelers can use OCL to specify application-specific constraints in their models.

OCL is used in the UML Semantics chapter to specify the well-formedness rules of the metaclasses comprising the UML metamodel. A well-formedness rule in the static semantics chapters in the UML Semantics section normally contains an OCL expression, specifying an invariant for the associated metaclass. The grammar for OCL is specified at the end of this chapter. A parser generated from this grammar has correctly parsed all the constraints in the UML Semantics section, a process which improved the correctness of the specifications for OCL and UML.

6.1.1 Why OCL?

A UML diagram, such as a class diagram, is typically not refined enough to provide all the relevant aspects of a specification. There is, among other things, a need to describe additional constraints about the objects in the model. Such constraints are often described in natural language. Practice has shown that this will always result in ambiguities. In order to write unambiguous constraints, so-called formal languages have been developed. The disadvantage of traditional formal languages is that they are usable to persons with a strong mathematical background, but difficult for the average business or system modeler to use.

OCL has been developed to fill this gap. It is a formal language that remains easy to read and write. It has been developed as a business modeling language within the IBM Insurance division, and has its roots in the Syntropy method.

OCL is a pure expression language; therefore, an OCL expression is guaranteed to be without side effect. When an OCL expression is evaluated, it simply returns a value. It cannot change anything in the model. This means that the state of the system will never change because of the evaluation of an OCL expression, even though an OCL expression can be used to *specify* a state change (for example, in a post-condition).

OCL is not a programming language; therefore, it is not possible to write program logic or flow control in OCL. You cannot invoke processes or activate non-query operations within OCL. Because OCL is a modeling language in the first place, not everything in it is promised to be directly executable.

OCL is a typed language, so that each OCL expression has a type. To be well formed, an OCL expression must conform to the type conformance rules of the language. For example, you cannot compare an Integer with a String. Each Classifier defined within a UML model represents a distinct OCL type. In addition, OCL includes a set of supplementary predefined types (these are described in Section 6.8, “Predefined OCL Types,” on page 6-29).

As a specification language, all implementation issues are out of scope and cannot be expressed in OCL.

The evaluation of an OCL expression is instantaneous. This means that the states of objects in a model cannot change during evaluation.

6.1.2 Where to Use OCL

OCL can be used for a number of different purposes:

- To specify invariants on classes and types in the class model
- To specify type invariant for Stereotypes
- To describe pre- and post conditions on Operations and Methods
- To describe Guards
- As a navigation language
- To specify constraints on operations

Within the UML Semantics chapter, OCL is used in the well-formedness rules as invariants on the metaclasses in the abstract syntax. In several places, it is also used to define 'additional' operations which are used in the well-formedness rules. Starting with UML 1.4, these additional operations can be formally defined using «definition» constraints and let-expressions.

6.2 Introduction

6.2.1 Legend

Text written in the courier typeface as shown below is an OCL expression.

```
'This is an OCL expression'
```

The *context* keyword introduces the context for the expression. The keyword *inv*, *pre* and *post* denote the stereotypes, respectively «invariant», «precondition», and «postcondition», of the constraint. The actual OCL expression comes after the colon.

```
context TypeName inv:
'this is an OCL expression with stereotype <<invariant>> in the
context of TypeName' = 'another string'
```

In the example, the keywords of OCL are written in boldface in this document. The boldface has no formal meaning, but is used to make the expressions more readable in this document. OCL expressions in this document are written using ASCII characters only.

Words in *Italics* within the main text of the paragraphs refer to parts of OCL expressions.

6.2.2 Example Class Diagram

Figure 6-1 on page 6-4 is used in the examples in this document.

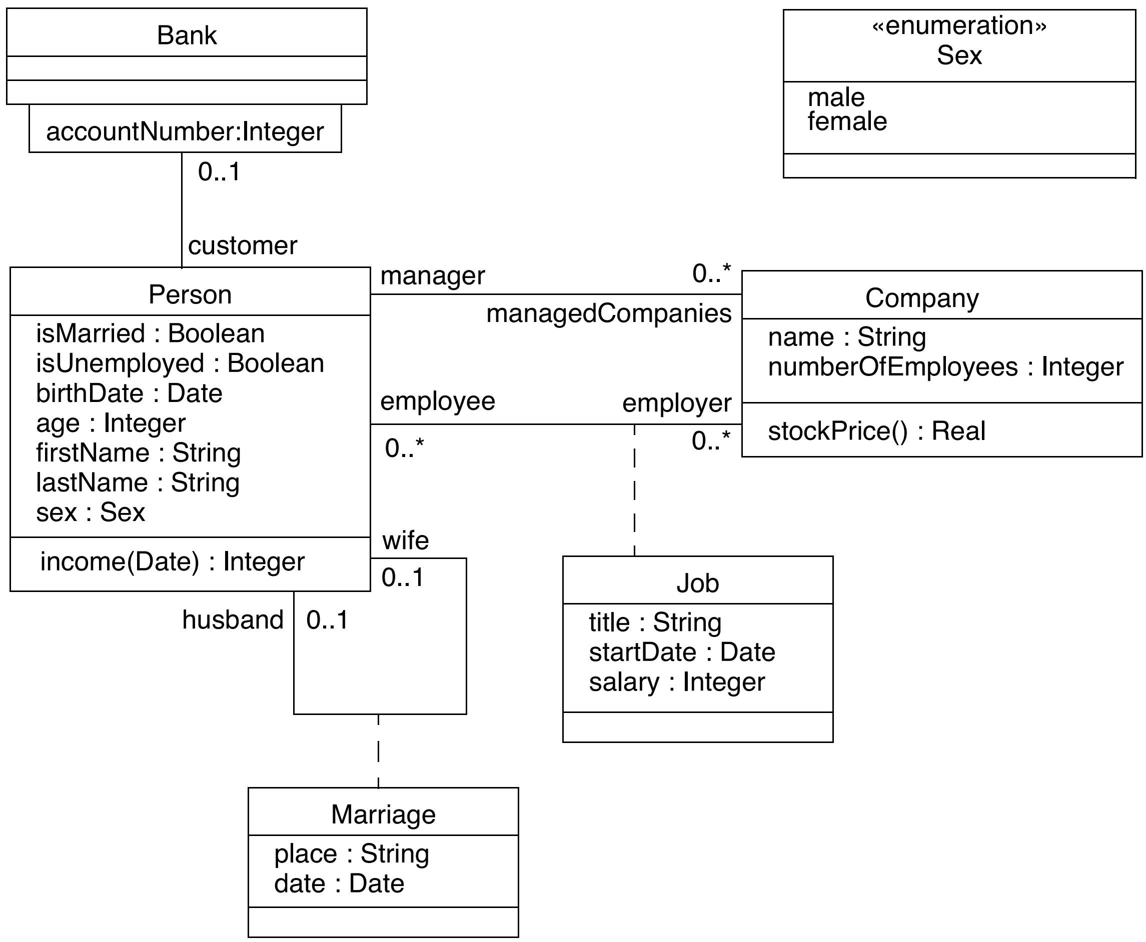


Figure 6-1 Class Diagram Example

6.3 Relation to the UML Metamodel

6.3.1 Self

Each OCL expression is written in the context of an instance of a specific type. In an OCL expression, the reserved word *self* is used to refer to the contextual instance. For instance, if the context is Company, then *self* refers to an instance of Company.

6.3.2 Specifying the UML context

The context of an OCL expression within a UML model can be specified through a so-called context declaration at the beginning of an OCL expression. The context declaration of the constraints in the following sections is shown.

If the constraint is shown in a diagram with the proper stereotype and the dashed lines to connect it to its contextual element, there is no need for an explicit context declaration in the test of the constraint. The context declaration is optional.

6.3.3 Invariants

The OCL expression can be part of an Invariant which is a Constraint stereotyped as an «invariant». When the invariant is associated with a Classifier, the latter is referred to as a “type” in this chapter. An OCL expression is an invariant of the type and must be true for all instances of that type at any time. (Note that all OCL expressions that express invariants are of the type Boolean.)

For example, if in the context of the Company type in Figure 6-1 on page 6-4, the following expression would specify an invariant that the number of employees must always exceed 50:

```
self.numberOfEmployees > 50
```

where *self* is an instance of type Company. (We can view *self* as the object from where we start the expression.) This invariant holds for every instance of the Company type.

The type of the contextual instance of an OCL expression, which is part of an invariant, is written with the *context* keyword, followed by the name of the type as follows. The label *inv:* declares the constraint to be an «invariant» constraint.

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

In most cases, the keyword *self* can be dropped because the context is clear, as in the above examples. As an alternative for self, a different name can be defined playing the part of self:

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

This invariant is equivalent to the previous one.

Optionally, the name of the constraint may be written after the *inv* keyword, allowing the constraint to be referenced by name. In the following example the name of the constraint is *enoughEmployees*. In the UML metamodel, this name is an attribute of the metaclass Constraint that is inherited from ModelElement.

```
context c : Company inv enoughEmployees:
    c.numberOfEmployees > 50
```

6.3.4 Pre- and Postconditions

The OCL expression can be part of a Precondition or Postcondition, corresponding to «precondition» and «postcondition» stereotypes of Constraint associated with an Operation or Method. The contextual instance *self* then is an instance of the type that owns the operation or method as a feature. The context declaration in OCL uses the *context* keyword, followed by the type and operation declaration.

The labels *pre:* and *post:* declare the constraints to be a «precondition» constraint and a «postcondition» constraint respectively.

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
    pre :  param1 > ...
    post:  result = ...
```

The name *self* can be used in the expression referring to the object on which the operation was called. The reserved word *result* denotes the result of the operation, if there is one. The names of the parameters (*param1*) can also be used in the OCL expression. In the example diagram, we can write:

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

Optionally, the name of the precondition or postcondition may be written after the *pre* or *post* keyword, allowing the constraint to be referenced by name. In the following example the name of the precondition is *parameterOk* and the name of the postcondition is *resultOk*. In the UML metamodel, these names are attributes of the metaclass Constraint that is inherited from ModelElement.

```
context Typename::operationName(param1 : Type1, ... ): ReturnType
    pre parameterOk:  param1 > ...
    post   resultOk:  result = ...
```

6.3.5 Package context

The above context declaration is precise enough when the package in which the Classifier belongs is clear from the environment. To specify explicitly in which package invariant, pre or postcondition Constraints belong, these constraints can be enclosed between 'package' and 'endpackage' statements. The package statements have the syntax:

```
package Package::SubPackage

context X inv:
    ... some invariant ...
context X::operationName(..)
    pre: ... some precondition ...

endpackage
```

An OCL file (or stream) may contain any number package statements, thus allowing all invariant, preconditions, and postconditions to be written down and stored in one file. This file may co-exist with a UML model as a separate entity.

6.3.6 General Expressions

Any OCL expression can be used as the value for an attribute of the UML metaclass Expression or one of its subtypes. In that case, the semantics section describes the meaning of the expression.

6.4 Basic Values and Types

In OCL, a number of basic types are predefined and available to the modeler at all times. These predefined value types are independent of any object model and part of the definition of OCL.

The most basic value in OCL is a value of one of the basic types. Some basic types used in the examples in this document, with corresponding examples of their values, are shown in Table 6-1.

Table 6-1 Basic Types

type	values
Boolean	true, false
Integer	1, -5, 2, 34, 26524, ...
Real	1.5, 3.14, ...
String	'To be or not to be...'

OCL defines a number of operations on the predefined types. Table 6-2 gives some examples of the operations on the predefined types. See Section 6.8, “Predefined OCL Types,” on page 6-29 for a complete list of all operations.

Table 6-2 Operations on predefined types

type	operations
Integer	*, +, -, /, abs()
Real	*, +, -, /, floor()
Boolean	and, or, xor, not, implies, if-then-else
String	toUpper(), concat()

The complete list of operations provided for each type is described at the end of this chapter. Collection, Set, Bag, and Sequence are basic types as well. Their specifics will be described in the upcoming sections.

6.4.1 Types from the UML Model

Each OCL expression is written in the context of a UML model, a number of classifiers (types/classes, ...), their features and associations, and their generalizations. All classifiers from the UML model are types in the OCL expressions that are attached to the model.

6.4.2 Enumeration Types

Enumerations are Datatypes in UML and have a name, just like any other Classifier. An enumeration defines a number of enumeration literals, that are the possible values of the enumeration. Within OCL one can refer to the value of an enumeration. When we have Datatype named Sex with values ‘female’ or ‘male’ they can be used as follows:

```
context Person inv:
       sex = Sex::male
```

6.4.3 Let Expressions and «definition» Constraints

Sometimes a sub-expression is used more than once in a constraint. The *let* expression allows one to define an attribute or operation that can be used in the constraint.

```
context Person inv:
    let income : Integer = self.job.salary->sum()
    let hasTitle(t : String) : Boolean =
                   self.job->exists(title = t) in
    if isUnemployed then
        self.income < 100
    else
        self.income >= 100 and self.hasTitle(‘manager’)
    endif
```

A let expression may be included in an invariant or pre- or postcondition. It is then only known within this specific constraint. To enable reuse of let variables/operations one can use a Constraint with the stereotype «definition», in which let variables/operations are defined. This «definition» Constraint must be attached to a Classifier and may only contain let definitions. 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. In essence, such variables and operations are psuedo-attributes and psuedo-operations of the classifier. They are used in an OCL expression in exactly the same way as attributes or operations are used. The textual notation for a «definition» Constraint uses the keyword ‘def’ as shown below:

```
context Person def:
    let income : Integer = self.job.salary->sum()
    let 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. Also, the names of all let variables and operations connected with a Classifier must be unique.

6.4.4 Type Conformance

OCL is a typed language and the basic value types are organized in a type hierarchy. This hierarchy determines conformance of the different types to each other. You cannot, for example, compare an Integer with a Boolean or a String.

An OCL expression in which all the types conform is a valid expression. An OCL expression in which the types don't conform is an invalid expression. It contains a type *conformance error*. A type *type1* conforms to a type *type2* when an instance of *type1* can be substituted at each place where an instance of *type2* is expected. The type conformance rules for types in the class diagrams are simple.

- Each type conforms to each of its supertypes.
- Type conformance is transitive: if *type1* conforms to *type2*, and *type2* conforms to *type3*, then *type1* conforms to *type3*.

The effect of this is that a type conforms to its supertype, and all the supertypes above. The type conformance rules for the value types are listed in Table 6-3.

Table 6-3 Type conformance rules

Type	Conforms to/Is a subtype of
Set(T)	Collection(T)
Sequence(T)	Collection(T)
Bag(T)	Collection(T)
Integer	Real

The conformance relation between the collection types only holds if they are collections of element types that conform to each other. See Section6.5.14, "Collection Type Hierarchy and Type Conformance Rules," on page 6-21 for the complete conformance rules for collections.

Table 6-4 provides examples of valid and invalid expressions.

Table 6-4 Valid expressions

OCL expression	valid	explanation
1 + 2 * 34	yes	
1 + 'motorcycle'	no	type String does not conform to type Integer
23 * false	no	type Boolean does not conform to Integer
12 + 13.5	yes	

6.4.5 Re-typing or Casting

In some circumstances, it is desirable to use a property of an object that is defined on a subtype of the current known type of the object. Because the property is not defined on the current known type, this results in a type conformance error.

When it is certain that the actual type of the object is the subtype, the object can be re-typed using the operation *oclAsType(OclType)*. This operation results in the same object, but the known type is the argument *OclType*. When there is an object *object* of type *Type1* and *Type2* is another type, it is allowed to write:

```
object.oclAsType(Type2) --- evaluates to object with type Type2
```

An object can only be re-typed to one of its subtypes; therefore, in the example, *Type2* must be a subtype of *Type1*.

If the actual type of the object is not a subtype of the type to which it is re-typed, the expression is undefined (see Section 6.4.10, “Undefined Values,” on page 6-11).

6.4.6 Precedence Rules

The precedence order for the operations, starting with highest precedence, in OCL is:

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

Parentheses ‘(’ and ‘)’ can be used to change precedence.

6.4.7 Use of Infix Operators

The use of infix operators is allowed in OCL. The operators ‘+’, ‘-’, ‘*’, ‘/’, ‘<‘, ‘>’, ‘<>’ ‘<=’ ‘>=’, ‘and’, ‘or’, and ‘xor’ are used as infix operators. If a type defines one of those operators with the correct signature, they will be used as infix operators. The expression:

```
a + b
```

is conceptually equal to the expression:

```
a.+(b)
```

that is, invoking the '+' operation on `a` with `b` as the parameter to the operation.

The infix operators defined for a type must have exactly one parameter. For the infix operators '<', '>', '<=', '>=', '<>', 'and', 'or', and 'xor' the return type must be Boolean.

6.4.8 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:

`if`	`implies`
`then`	`endpackage`
`else`	`package`
`endif`	`context`
`not`	`def`
`let`	`inv`
`or`	`pre`
`and`	`post`
`xor`	`in`

6.4.9 Comment

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
```

6.4.10 Undefined Values

Whenever an OCL expression is being evaluated, there is a possibility that one or more of the queries in the expression are undefined. If this is the case, then the complete expression will be undefined.

There are two exceptions to this for the Boolean operators:

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

The above two rules are valid irrespective of the order of the arguments and the above rules are valid whether or not the value of the other sub-expression is known.

6.5 Objects and Properties

OCL expressions can refer to Classifiers; for example, types, classes, interfaces, associations (acting as types), and datatypes. Also all attributes, association-ends, methods, and operations without side-effects that are defined on these types, etc. can be used. In a class model, an operation or method is defined to be side-effect-free if the isQuery attribute of the operations is true. For the purpose of this document, we will refer to attributes, association-ends, and side-effect-free methods and operations as being *properties*. A property is one of:

- an Attribute
- an AssociationEnd
- an Operation with *isQuery* being true
- a Method with *isQuery* being true

6.5.1 Properties

The value of a property on an object that is defined in a class diagram is specified by a dot followed by the name of the property.

```
context AType inv:
    self.property
```

If *self* is a reference to an object, then *self.property* is the value of the *property* property on *self*.

6.5.2 Properties: Attributes

For example, the age of a Person is written as *self.age*:

```
context Person inv:
    self.age > 0
```

The value of the subexpression *self.age* is the value of the *age* attribute on the particular instance of Person identified by *self*. The type of this subexpression is the type of the attribute *age*, which is the basic type Integer.

Using attributes, and operations defined on the basic value types, we can express calculations etc. over the class model. For example, a business rule might be “the age of a Person is always greater than zero.” This can be stated as shown in the invariant above.

6.5.3 Properties: Operations

Operations may have parameters. For example, as shown earlier, a Person object has an income expressed as a function of the date. This operation would be accessed as follows, for a Person *aPerson* and a date *aDate*:

```
aPerson.income(aDate)
```

The operation itself could be defined by a postcondition constraint. This is a constraint that is stereotyped as «postcondition». The object that is returned by the operation can be referred to by *result*. It takes the following form:

```
context Person::income (d: Date) : Integer
    post: result = age * 1000
```

The right-hand-side of this definition may refer to the operation being defined; that is, the definition may be recursive as long as the recursion is not infinite. The type of *result* is the return type of the operation, which is Integer in the above example.

To refer to an operation or a method that doesn't take a parameter, parentheses with an empty argument list are mandatory:

```
context Company inv:
    self.stockPrice() > 0
```

6.5.4 Properties: Association Ends 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.rolename
```

The value of this expression is the set of objects on the other side of the *rolename* 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; that is, *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 a Sequence.

Collections, like Sets, 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.

6.5.4.1 Missing Rolenames

When a rolename 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 the case with unnamed rolenames in reflexive associations. If the rolename is ambiguous, then it cannot be used in OCL.

6.5.4.2 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.

The following example shows how a property of a collection can be used.

```
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.manager* 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.sex = Sex::female
```

6.5.4.3 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
```

6.5.5 Navigation to Association Classes

To specify navigation to association classes (Job and Marriage in the example), OCL uses a dot and the name of the association class starting with a lowercase character:

```
context Person inv:
    self.job
```

The sub-expression *self.job* evaluates to a Set of all the jobs a person has with the companies that are his/her employer. In the case of an association class, there is no explicit rolename in the class diagram. The name *job* used in this navigation is the name of the association class starting with a lowercase character, similar to the way described in the section "Missing Rolenames" above.

In case of a recursive association, that is an association of a class with itself, the name of the association class alone is not enough. We need to distinguish the direction in which the association is navigated as well as the name of the association class. Take the following model as an example.

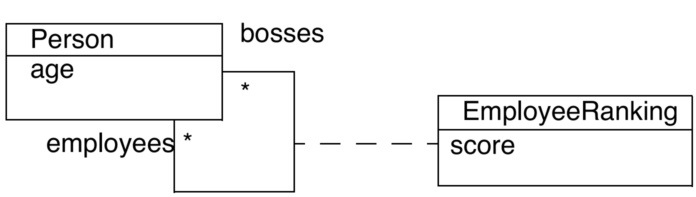


Figure 6-2 Navigating recursive association classes

When navigating to an association class such as *employeeRanking* there are two possibilities depending on the direction. For instance, in the above example, we may navigate towards the *employees* end, or the *bosses* end. By using the name of the association class alone, these two options cannot be distinguished. To make the distinction, the rolename of the direction in which we want to navigate is added to the association class name, enclosed in square brackets.

In the expression

```
context Person inv:
    self.employeeRanking[bosses]->sum() > 0
```

the *self.employeeRanking[bosses]* evaluates to the set of *EmployeeRankings* belonging to the collection of *bosses*. And in the expression

```
context Person inv:
    self.employeeRanking[employees]->sum() > 0
```

the *self.employeeRanking[employees]* evaluates to the set of *EmployeeRankings* belonging to the collection of *employees*. The unqualified use of the association class name is not allowed in such a recursive situation. Thus, the following example is invalid:

```
context Person inv:
    self.employeeRanking->sum() > 0 -- INVALID!
```

In a non-recursive situation, the association class name alone is enough, although the qualified version is allowed as well. Therefore, the examples at the start of this section could also be written as:

```
context Person inv:
    self.job[employer]
```

6.5.6 Navigation from Association Classes

We can navigate from the association class itself to the objects that participate in the association. This is done using the dot-notation and the role-names at the association-ends.

```
context Job
    inv: self.employer.numberOfEmployees >= 1
    inv: self.employee.age > 21
```

Navigation from an association class to one of the objects on the association will always deliver exactly one object. This is a result of the definition of AssociationClass. Therefore, the result of this navigation is exactly one object, although it can be used as a Set using the arrow (->).

6.5.7 Navigation through Qualified Associations

Qualified associations use one or more qualifier attributes to select the objects at the other end of the association. To navigate them, we can add the values for the qualifiers to the navigation. This is done using square brackets, following the role-name. It is permissible to leave out the qualifier values, in which case the result will be all objects at the other end of the association.

```
context Bank inv:
    self.customer
```

This results in a Set(Person) containing all customers of the Bank.

```
context Bank inv:
    self.customer[8764423]
```

This results in one Person, having accountnumber 8764423.

If there is more than one qualifier attribute, the values are separated by commas, in the order which is specified in the UML class model. It is not permissible to partially specify the qualifier attribute values.

6.5.8 Using Pathnames for Packages

Within UML, different types are organized in packages. OCL provides a way of explicitly referring to types in other packages by using a package-pathname prefix. The syntax is a package name, followed by a double colon:

```
Packagename::Typename
```

This usage of pathnames is transitive and can also be used for packages within packages:

```
Packagename1::Packagename2::Typename
```

6.5.9 Accessing overridden properties of supertypes

Whenever properties are redefined within a type, the property of the supertypes can be accessed using the *oclAsType()* operation. Whenever we have a class B as a subtype of class A, and a property p1 of both A and B, we can write:

```
context B inv:
    self.oclAsType(A).p1  -- accesses the p1 property defined in A
    self.p1              -- accesses the p1 property defined in B
```

Figure 6-3 shows an example where such a construct is needed.

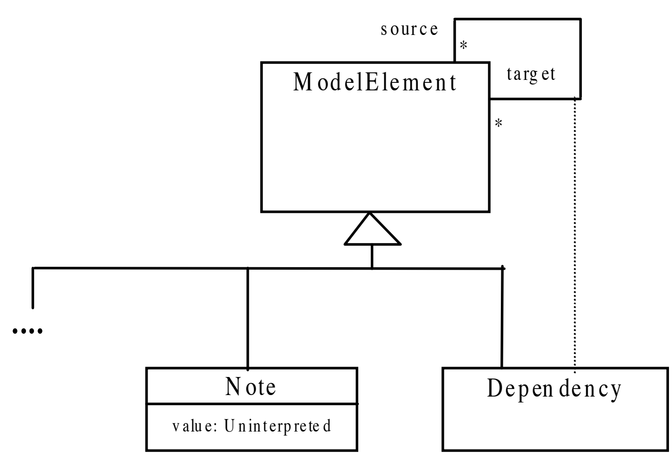


Figure 6-3 Accessing Overridden Properties Example

In this model fragment there is an ambiguity with the OCL expression on Dependency:

```
context Dependency inv:
    self.source <> self
```

This can either mean normal association navigation, which is inherited from ModelElement, or it might also mean navigation through the dotted line as an association class. Both possible navigations use the same role-name, so this is always ambiguous. Using *oclAsType()* we can distinguish between them with:

```
context Dependency
    inv: self.oclAsType(Dependency).source
    inv: self.oclAsType(ModelElement).source
```

6.5.10 Predefined properties on All Objects

There are several properties that apply to all objects, and are predefined in OCL. These are:

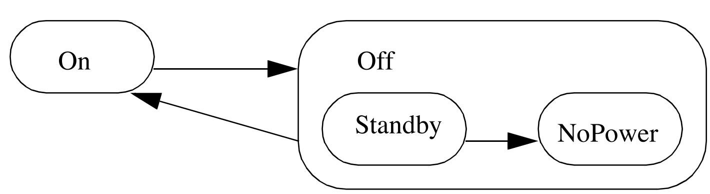
```
oclIsTypeOf(t : OclType)   : Boolean
oclIsKindOf(t : OclType)   : Boolean
oclInState(s : OclState)   : Boolean
oclIsNew()                 : Boolean
oclAsType(t : OclType) : instance of OclType
```

The operation is *oclTypeOf* results in true if the *type* of self and *t* are the same. For example:

```
context Person
    inv: self.oclIsTypeOf( Person )      -- is true
    inv: self.oclIsTypeOf( Company)      -- is false
```

The above property deals with the direct type of an object. The *oclIsKindOf* property determines whether *t* is either the direct type or one of the supertypes of an object.

The operation *oclInState(s)* results in true if the object is in the state *s*. Values for *s* are the names of the states in the statemachine(s) attached to the Classifier of *object*. For nested states the statenames can be combined using the double colon '::' .



In the example statemachine above, values for *s* can be *On*, *Off*, *Off::Standby*, *Off::NoPower*. If the classifier of *object* has the above associated statemachine valid OCL expressions are:

```
object.oclInState(On)
object.oclInState(Off)
object.oclInstate(Off::Standby)
object.oclInState(Off:NoPower)
```

If there are multiple statemachines attached to the object's classifier, then the statename can be prefixed with the name of the statemachine containing the state and the double semicolon ::, as with nested states.

The operation *oclIsNew* evaluates to true if, used in a postcondition, the object is created during performing the operation; that is, it didn't exist at precondition time.

6.5.11 Features on Classes Themselves

All properties discussed until now in OCL are properties on instances of classes. The types are either predefined in OCL or defined in the class model. In OCL, it is also possible to use features defined on the types/classes themselves. These are, for example, the *class*-scoped features defined in the class model. Furthermore, several features are predefined on each type.

A predefined feature on each type is *allInstances*, which results in the Set of all instances of the type in existence at the specific time when the expression is evaluated. If we want to make sure that all instances of Person have unique names, we can write:

```
context Person inv:
    Person.allInstances->forAll(p1, p2 |
                                  p1 <> p2 implies p1.name <> p2.name)
```

The *Person.allInstances* is the set of all persons and is of type Set(Person). It is the set of all persons that exist at the snapshot in time that the expression is evaluated.

Note – The use of *allInstances* has some problems and its use is discouraged in most cases. The first problem is best explained by looking at the types like Integer, Real and String. For these types the meaning of *allInstances* is undefined. What does it mean for an Integer to exist? The evaluation of the expression *Integer.allInstances* results in an infinite set and is therefore undefined within OCL. The second problem with *allInstances* is that the existence of objects must be considered within some overall context, like a system or a model. This overall context must be defined, which is not done within OCL. A recommended style is to model the overall contextual system explicitly as an object within the system and navigate from that object to its containing instances without using *allInstances*.

6.5.12 Collections

Single navigation results in a Set, combined navigations in a Bag, and navigation over associations adorned with {ordered} results in a Sequence. Therefore, the collection types play an important role in OCL expressions.

The type Collection is predefined in OCL. The Collection type defines a large number of predefined operations to enable the OCL expression author (the modeler) to manipulate collections. Consistent with the definition of OCL as an expression language, collection operations never change collections; *isQuery* is always true. They may result in a collection, but rather than changing the original collection they project the result into a new one.

Collection is an abstract type, with the concrete collection types as its subtypes. OCL distinguishes three different collection types: Set, Sequence, and Bag. A Set is the mathematical set. It does not contain duplicate elements. A Bag is like a set, which may contain duplicates; that is, the same element may be in a bag twice or more. A Sequence is like a Bag in which the elements are ordered. Both Bags and Sets have no order defined on them. 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 at the end of this chapter.

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, Sequence, or Bag is:

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

```
Set      {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)
```

6.5.13 Collections of Collections

Within OCL, all Collections of Collections are flattened automatically; therefore, the following two expressions have the same value:

```
Set{ Set{1, 2}, Set{3, 4}, Set{5, 6} }
Set{ 1, 2, 3, 4, 5, 6 }
```

6.5.14 Collection Type Hierarchy and Type Conformance Rules

In addition to the type conformance rules in Section 6.4.4, "Type Conformance," on page 6-9, the following rules hold for all types, including the collection types:

- The types Set (X), Bag (X) and Sequence (X) are all subtypes of Collection (X).

Type conformance rules are as follows for the collection types:

- *Type1* conforms to *Type2* when they are identical (standard rule for all types).
- *Type1* conforms to *Type2* when it is a subtype of Type2 (standard rule for all types).
- *Collection(Type1)* conforms to *Collection(Type2)*, when *Type1* conforms to *Type2*.
- Type conformance is transitive: if *Type1* conforms to *Type2*, and *Type2* conforms to *Type3*, then *Type1* conforms to *Type3* (standard rule for all types).

For example, if *Bicycle* and *Car* are two separate subtypes of *Transport:*

`Set(Bicycle)`	conforms to	`Set(Transport)`
`Set(Bicycle)`	conforms to	`Collection(Bicycle)`
`Set(Bicycle)`	conforms to	`Collection(Transport)`

Note that Set(Bicycle) does not conform to Bag(Bicycle), nor the other way around. They are both subtypes of Collection(Bicycle) at the same level in the hierarchy.

6.5.15 Previous Values in Postconditions

As stated in Section 6.3.4, "Pre- and Postconditions," on page 6-6, OCL can be used to specify pre- and post-conditions on operations and methods in UML. In a postcondition, the expression can refer to two sets of values for each property of an object:

- the value of a property at the start of the operation or method
- the value of a property upon completion of the operation or method

The value of a property in a postcondition is the value upon completion of the operation. To refer to the value of a property at the start of the operation, one has to postfix the property name with the keyword '*@pre*':

```
context Person::birthdayHappens()
    post: age = age@pre + 1
```

The property *age* refers to the property of the instance of Person on which executes the operation. The property *age@pre* refers to the value of the property *age* of the Person that executes the operation, at the start of the operation.

If the property has parameters, the ‘@pre’ is postfixed to the propertyname, before the parameters.

```
context Company::hireEmployee(p : Person)
    post: employees = employees@pre->including(p) and
        stockprice() = stockprice@pre() + 10
```

The above operation can also be specified by a postcondition and a precondition together:

```
context Company::hireEmployee(p : Person)
    pre : not employee->includes(p)
    post: employees->includes(p) and
          stockprice() = stockprice@pre() + 10
```

When the pre-value of a property evaluates to an object, all further properties that are accessed of this object are the new values (upon completion of the operation) of this object. So:

```
a.b@pre.c          -- takes the old value of property b of a, say x
                   -- and then the new value of c of x.
a.b@pre.c@pre      -- takes the old value of property b of a, say x
                   -- and then the old value of c of x.
```

The ‘@pre’ postfix is allowed only in OCL expressions that are part of a Postcondition. Asking for a current property of an object that has been destroyed during execution of the operation results in Undefined. Also, referring to the previous value of an object that has been created during execution of the operation results in Undefined.

6.6 Collection Operations

OCL defines many operations on the collection types. These operations are specifically meant to enable a flexible and powerful way of projecting new collections from existing ones. The different constructs are described in the following sections.

6.6.1 Select and Reject Operations

Sometimes an expression using operations and navigations delivers a collection, while we are interested only in a special subset of the collection. OCL has special constructs to specify a selection from a specific collection. These are the *select* and *reject* operations. The select specifies a subset of a collection. A select is an operation on a collection and is specified using the arrow-syntax:

```
collection->select( ... )
```

The parameter of select has a special syntax that enables one to specify which elements of the collection we want to select. There are three different forms, of which the simplest one is:

```
collection->select( boolean-expression )
```

This results in a collection that contains all the elements from *collection* for which the *boolean-expression* evaluates to true. To find the result of this expression, for each element in *collection* the expression *boolean-expression* is evaluated. If this evaluates to true, the element is included in the result collection, otherwise not. As an example, the following OCL expression specifies that the collection of all the employees older than 50 years is not empty:

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

The *self.employee* is of type Set(Person). The *select* takes each person from *self.employee* and evaluates *age > 50* for this person. If this results in *true*, then the person is in the result Set.

As shown in the previous example, the context for the expression in the select argument is the element of the collection on which the select is invoked. Thus the *age* property is taken in the context of a person.

In the above example, it is impossible to refer explicitly to the persons themselves; you can only refer to properties of them.To enable to refer to the persons themselves, there is a more general syntax for the select expression:

```
collection->select( v | boolean-expression-with-v )
```

The variable *v* is called the iterator. When the select is evaluated, *v* iterates over the *collection* and the *boolean-expression-with-v* is evaluated for each *v*. The *v* is a reference to the object from the collection and can be used to refer to the objects themselves from the *collection*. The two examples below are identical:

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

The result of the complete select is the collection of persons *p* for which the *p.age > 50* evaluates to True. This amounts to a subset of *self.employee*.

As a final extension to the select syntax, the expected type of the variable v can be given. The select now is written as:

```
collection->select( v : Type | boolean-expression-with-v )
```

The meaning of this is that the objects in *collection* must be of type *Type*. The next example is identical to the previous examples:

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

The compete select syntax now looks like one of:

```
collection->select( v : Type | boolean-expression-with-v )
```

```
collection->select( v | boolean-expression-with-v )
collection->select( boolean-expression )
```

The *reject* operation is identical to the select operation, but with reject we get the subset of all the elements of the collection for which the expression evaluates to False. The reject syntax is identical to the select syntax:

```
collection->reject( v : Type | boolean-expression-with-v )
collection->reject( v | boolean-expression-with-v )
collection->reject( boolean-expression )
```

As an example, specify that the collection of all the employees who are **not** married is empty:

```
context Company inv:
    self.employee->reject( isMarried )->isEmpty()
```

The reject operation is available in OCL for convenience, because each reject can be restated as a select with the negated expression. Therefore, the following two expressions are identical:

```
collection->reject( v : Type | boolean-expression-with-v )
collection->select( v : Type  | not (boolean-expression-with-v) )
```

6.6.2 Collect Operation

As shown in the previous section, the select and reject operations always result in a sub-collection of the original collection. When we want to specify a collection that is derived from some other collection, but which contains different objects from the original collection; that is, it is not a sub-collection, we can use a *collect* operation. The collect operation uses the same syntax as the select and reject and is written as one of:

```
collection->collect( v : Type | expression-with-v )
collection->collect( v | expression-with-v )
collection->collect( expression )
```

The value of the reject operation is the collection of the results of all the evaluations of *expression-with-v.*

An example: specify the collection of *birthDates* for all employees in the context of a company. This can be written in the context of a Company object as one of:

```
self.employee->collect( birthDate )
self.employee->collect( person | person.birthDate )
self.employee->collect( person : Person | person.birthDate )
```

An important issue here is that the resulting collection is not a Set, but a Bag. When more than one employee has the same value for *birthDate*, this value will be an element of the resulting Bag more than once. The Bag resulting from the *collect* operation always has the same size as the original collection.

It is possible to make a Set from the Bag, by using the asSet property on the Bag. The following expression results in the Set of different *birthDates* from all employees of a Company:

```
self.employee->collect( birthDate )->asSet()
```

6.6.2.1 Shorthand for Collect

Because navigation through many objects is very common, there is a shorthand notation for the collect that makes the OCL expressions more readable. Instead of

```
self.employee->collect(birthdate)
```

we can also write:

```
self.employee.birthdate
```

In general, when we apply a property to a collection of Objects, then it will automatically be interpreted as a *collect* over the members of the collection with the specified property.

For any *propertyname* that is defined as a property on the objects in a collection, the following two expressions are identical:

```
collection.propertyname
collection->collect(propertyname)
```

and so are these if the property is parameterized:

```
collection.propertyname(par1, par2, ...)
collection->collect(propertyname(par1, par2, ...)
```

6.6.3 ForAll Operation

Many times a constraint is needed on all elements of a collection. The forAll operation in OCL allows specifying a Boolean expression, which must hold for all objects in a collection:

```
collection->forAll( v : Type | boolean-expression-with-v )
collection->forAll( v | boolean-expression-with-v )
collection->forAll( boolean-expression )
```

This forAll expression results in a Boolean. The result is true if the *boolean-expression-with-v* is true for all elements of *collection*. If the *boolean-expression-with-v* is false for one or more *v* in *collection*, then the complete expression evaluates to false. For example, in the context of a company:

```
context Company
    inv:    self.employee->forAll( forename = 'Jack' )
    inv:    self.employee->forAll( p | p.forename = 'Jack' )
    inv:    self.employee->forAll( p : Person | p.forename = 'Jack' )
```

These invariants evaluate to true if the forename feature of each employee is equal to ‘Jack.’

The forAll operation has an extended variant in which more then one iterator is used. Both iterators will iterate over the complete collection. Effectively this is a forAll on the Cartesian product of the collection with itself.

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

This expression evaluates to true if the forenames of all employees are different. It is semantically equivalent to:

```
context Company inv:
    self.employee->forAll(e1 | self.employee->forAll (e2 |
                      e1 <> e2 implies e1.forename <> e2.forename)))
```

6.6.4 Exists Operation

Many times one needs to know whether there is at least one element in a collection for which a constraint holds. The *exists* operation in OCL allows you to specify a Boolean expression that must hold for at least one object in a collection:

```
collection->exists( v : Type | boolean-expression-with-v )
collection->exists( v | boolean-expression-with-v )
collection->exists( boolean-expression )
```

This exists operation results in a Boolean. The result is true if the *boolean-expression-with-v* is true for at least one element of *collection*. If the *boolean-expression-with-v* is false for all *v* in *collection*, then the complete expression evaluates to false. For example, in the context of a company:

```
context Company inv:
    self.employee->exists( forename = 'Jack' )
context Company inv:
    self.employee->exists( p | p.forename = 'Jack' )
context Company inv:
    self.employee->exists( p : Person | p.forename = 'Jack' )
```

These expressions evaluate to true if the forename feature of at least one employee is equal to ‘Jack.’

6.6.5 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>
   }
}
```

Although the Java pseudo code uses a ‘next element,’ the *iterate* operation is defined for each collection type and 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.

6.6.6 Iterators in Collection Operations

The collection operations that take an OclExpression as parameter may all have an optional iterator declaration. For any operation name *op*, the syntax options are:

```
collection->op( iter : Type | OclExpression )
collection->op( iter | OclExpression )
collection->op( OclExpression )
```

6.6.7 Resolving Properties

For any property (attribute, operation, or navigation), the full notation includes the object of which the property is taken. As seen in Section 6.3.3, “Invariants,” on page 6-5, *self* can be left implicit, and so can the iterator variables in collection operations. At any place in an expression, when an iterator is left out, an implicit iterator-variable is introduced. For example in:

```
context Person inv:
    employer->forAll( employee->exists( lastName = name) )
```

three implicit variables are introduced. The first is *self*, which is always the instance from which the constraint starts. Secondly an implicit iterator is introduced by the *forAll* and third by the *exists*. The implicit iterator variables are unnamed. The properties *employer*, *employee*, *lastName* and *name* all have the object on which they are applied left out. Resolving these goes as follows:

- At the place of *employer* there is one implicit variable: *self : Person*. Therefore *employer* must be a property of *self*.
- At the place of *employee* there are two implicit variables: *self : Person* and *iter1 : Company*. Therefore *employer* must be a property of either *self* or *iter1*. If *employee* is a property of both *self* and *iter1*, then this is unambiguous and the instance on which *employee* is applied must be stated explicitly. In this case only *iter1.employee* is possible.
- At the place of *lastName* and *name* there are three implicit variables: *self : Person* , *iter1 : Company* and *iter2 : Person*. Therefore *lastName* and *name* must both be a property of either *self* or *iter1* or *iter2*. Property *name* is a property of *iter1*. However, *lastName* is a property of both *self* and *iter2*. This is ambiguous and therefore the OCL expression is incorrect. The expression must state either *self.lastName* or define the *iter2* iterator variable explicit and state *iter2.lastName*.

Both of the following invariant constraints are correct:

```
context Person
    inv: employer->forAll( employee->exists( p | p.lastName = name) )
    inv: employer->forAll( employee->exists( self.lastName = name) )
```

6.7 The Standard OCL Package

Each UML model that uses OCL constraints contains a predefined standard package called “UML_OCL.” This package is used by default in all other packages in the model to evaluate OCL expressions. This package contains all predefined OCL types and their features.

To extend the predefined OCL types, a modeler should define a separate package. The standard OCL package can be imported, and each OCL type can be extended with new features.

To specify that a package used the predefined OCL types from a user defined package instead of the standard package, the using package must define a Dependency with stereotype «OCL_Types» to the package that defines the extended OCL types.

A constraint on the user defined OCL package is that as a minimum all predefined OCL types with all of their features must be defined. The user defined package must be a proper extension to the standard OCL package.

6.8 Predefined OCL Types

This section contains all standard types defined within OCL, including all the properties defined on those types. Its signature and a description of its semantics define each property. Within the description, the reserved word 'result' is used to refer to the value that results from evaluating the property. 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.

6.8.1 Basic Types

The basic types used are Integer, Real, String, and Boolean. They are supplemented with OclExpression, OclType, and OclAny.

6.8.1.1 OclType

All types defined in a UML model, or pre-defined within OCL, have a type. This type is an instance of the OCL type called OclType. Access to this type allows the modeler limited access to the meta-level of the model. This can be useful for advanced modelers.

Properties of OclType, where the instance of OclType is called t*ype*.

type.name() : String

The name of *type*.

type.attributes() : Set(String)

The set of names of the attributes of *type*, as they are defined in the model.

type.associationEnds() : Set(String)

The set of names of the navigable associationEnds of *type*, as they are defined in the model.

type.operations() : Set(String)

The set of names of the operations of *type*, as they are defined in the model.

type.supertypes() : Set(OclType)

The set of all direct supertypes of *type*.
post: type.allSupertypes()->includesAll(result)

type.allSupertypes() : Set(OclType)

The transitive closure of the set of all supertypes of *type*.

type.allInstances() : Set(type)

The set of all instances of *type* and all its subtypes in existence at the snapshot at the time that the expression is evaluated.

6.8.1.2 OclAny

Within the OCL context, the type OclAny is the supertype of all types in the model and the basic predefined OCL type. The predefined OCL Collection types are not subtypes of OclAny. Properties of OclAny are available on each object in all OCL expressions.

All classes in a UML model inherit all properties 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.

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

object = (object2 : OclAny) : Boolean

True if *object* is the same object as *object2*.

object <> (object2 : OclAny) : Boolean

True if *object* is a different object from *object2*.
post: result = not (object = object2)

object.oclIsKindOf(type : OclType) : Boolean

True if *type* is one of the types of *object*, or one of the supertypes (transitive) of the types of *object*.

object.oclIsTypeOf(type : OclType) : Boolean

True if *type* is equal to one of the types of *object*.

object.oclAsType(type : OclType) : type

Results in *object*, but of known type *type*.
Results in Undefined if the actual type of *object* is not type or one of its subtypes.
pre : object.oclIsKindOf(type)
post: result = object
post: result.oclIsKindOf(type)

6.8.1.3

object.oclInState(state : OclState) : Boolean

Results in true if *object* is in the state *state*, otherwise results in false. The argument is a name of a state in the state machine corresponding with the class of *object*.

6.8.1.4

object.oclIsNew() : Boolean

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

6.8.1.5 OclState

The type OclState is used as a parameter for the operation *oclInState*. There are no properties defined on OclState. One can only specify an OclState by using the name of the state, as it appears in a statemachine. These names can be fully qualified by the nested states and statemachine that contain them.

6.8.1.6 OclExpression

Each OCL expression itself is an object in the context of OCL. The type of the expression is OclExpression. This type and its properties are used to define the semantics of properties that take an expression as one of their parameters: select, collect, forAll, etc.

An OclExpression includes the optional iterator variable and type and the optional accumulator variable and type.

Properties of OclExpression, where the instance of OclExpression is called *expression*.

expression.evaluationType() : OclType

The type of the object that results from evaluating *expression*.

6.8.1.7 Real

The OCL 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.

Properties of Real, where the instance of Real is called *r*.

r = (r2 : Real) : Boolean

True if *r* is equal to *r2*.

r <> (r2 : Real) : Boolean

True if *r* is not equal to *r2*.
post: result = not (r = r2)

r + (r2 : Real) : Real

The value of the addition of *r* and *r2*.

r - (r2 : Real) : Real

The value of the subtraction of *r2* from *r*.

r * (r2 : Real) : Real

The value of the multiplication of *r* and *r2*.

- r : Real

The negative value of *r*.

r / (r2 : Real) : Real

The value of *r* divided by *r2*.

r.abs() : Real

The absolute value of *r*.
post: if r < 0 then result = - r else result = r endif

r.floor() : Integer

The largest integer which is less than or equal to *r*.
post: (result <= r) and (result + 1 > r)

r.round() : Integer

The integer that is closest to *r*. When there are two such integers, the largest one.
post: ((r - result) < r).abs() < 0.5) or ((r - result).abs() = 0.5 and (result > r))

r.max(r2 : Real) : Real

The maximum of *r* and *r2*.
post: if r >= r2 then result = r else result = r2 endif

r.min(r2 : Real) : Real

The minimum of *r* and *r2*.
post: if r <= r2 then result = r else result = r2 endif

r < (r2 : Real) : Boolean

True if *r1* is less than *r2*.

r > (r2 : Real) : Boolean

True if *r1* is greater than *r2*.
post: result = not (r <= r2)

r <= (r2 : Real) : Boolean

True if r1 is less than or equal to r2.
post: result = (r = r2) or (r < r2)

r >= (r2 : Real) : Boolean

True if r1 is greater than or equal to r2.
post: result = (r = r2) or (r > r2)

6.8.1.8 Integer

The OCL type Integer represents the mathematical concept of integer.

Properties of Integer, where the instance of Integer is called *i*.

i = (i2 : Integer) : Boolean

True if *i* is equal to *i2*.

- i : Integer

The negative value of *i*.

i + (i2 : Integer) : Integer

The value of the addition of *i* and *i2*.

i - (i2 : Integer) : Integer

The value of the subtraction of *i2* from *i*.

i * (i2 : Integer) : Integer

The value of the multiplication of *i* and *i2*.

i / (i2 : Integer) : Real

The value of *i* divided by *i2*.

i.abs() : Integer

The absolute value of *i*.
post: if i < 0 then result = - i else result = i endif

i.div(i2 : Integer) : Integer

The number of times that *i2* fits completely within *i*.
pre : i2 <> 0
post: if i / i2 >= 0 then result = (i / i2).floor() else result = -((-i/i2).floor()) endif

i.mod(i2 : Integer) : Integer

The result is *i* modulo *i2*.
post: result = i - (i.div(i2) * i2)

i.max(i2 : Integer) : Integer

The maximum of *i* an *i2*.
post: if i >= i2 then result = i else result = i2 endif

i.min(i2 : Integer) : Integer

The minimum of *i* an *i2*.
post: if i <= i2 then result = i else result = i2 endif

6.8.1.9 String

The OCL type String represents strings consisting of ASCII characters or multi-byte characters.

Properties of String, where the instance of String is called *string*.

string = (string2 : String) : Boolean

True if *string* and *string2* contain the same characters, in the same order.

string.size() : Integer

The number of characters in *string*.

string.concat(string2 : String) : String

The concatenation of *string* and *string2*.
post: result.size() = string.size() + string2.size()
post: result.substring(1, string.size()) = string
post: result.substring(string.size() + 1, result.size()) = string2

string.toUpper() : String

The value of *string* with all lowercase characters converted to uppercase characters.
post: result.size() = string.size()

string.toLower() : String

The value of *string* with all uppercase characters converted to lowercase characters.
post: result.size() = string.size()

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

The sub-string of *string* starting at character number *lower*, up to and including character number *upper*.

6.8.1.10 Boolean

The OCL type Boolean represents the common true/false values.

Features of Boolean, the instance of Boolean is called *b*.

b = (b2 : Boolean) : Boolean

Equal if *b* is the same as *b2*.

b or (b2 : Boolean) : Boolean

True if either *b* or *b2* is true.

b xor (b2 : Boolean) : Boolean

True if either *b* or *b2* is true, but not both.
post: (b or b2) and not (b = b2)

b and (b2 : Boolean) : Boolean

True if both *b1* and *b2* are true.

not b : Boolean

True if *b* is false.
post: if b then result = false else result = true endif

b implies (b2 : Boolean) : Boolean

True if *b* is false, or if *b* is true and *b2* is true.
post: (not b) or (b and b2)

if b then (expression1 : OclExpression)

else (expression2 : OclExpression) endif : expression1.evaluationType()

If *b* is true, the result is the value of evaluating *expression1*; otherwise, result is the value of evaluating *expression2*.

6.8.1.11 Enumeration

The OCL type Enumeration represents the enumerations defined in a UML model.

Features of Enumeration, the instance of Enumeration is called *enumeration*.

enumeration = (enumeration2 : Boolean) : Boolean

Equal if *enumeration* is the same as *enumeration2*.

enumeration <> (enumeration2 : Boolean) : Boolean

Equal if *enumeration* is not the same as *enumeration2*.
post: result = not (enumeration = enumeration2)

6.8.2 Collection-Related Types

The following sections define the properties on collections; that is, these properties are available on Set, Bag, and Sequence. As defined in this section, each collection type is actually a template with one parameter. 'T' denotes the parameter. A real collection type is created by substituting a type for the T. So Set (Integer) and Bag (Person) are collection types.

All collection operations with an OclExpression as parameter can have an iterator declarator.

6.8.2.1 Collection

Collection is the abstract supertype of all collection types in OCL. 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 properties may be defined with the subtype as well, which means that there is an additional postcondition or a more specialized return value.

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

Properties of Collection, where the instance of Collection is called *collection*.

collection->size() : Integer

The number of elements in the collection *collection*.
post: result = collection->iterate(elem; acc : Integer = 0 | acc + 1)

collection->includes(object : OclAny) : Boolean

True if *object* is an element of *collection*, false otherwise.
post: result = (collection->count(object) > 0)

collection->excludes(object : OclAny) : Boolean

True if *object* is not an element of *collection*, false otherwise.
post: result = (collection->count(object) = 0)

collection->count(object : OclAny) : Integer

The number of times that *object* occurs in the collection *collection*.
post: result = collection->iterate(elem; acc : Integer = 0 |
    if elem = object then acc + 1 else acc endif)

collection->includesAll(c2 : Collection(T)) : Boolean

Does *collection* contain all the elements of *c2* ?
post: result = c2->forAll(elem | collection->includes(elem))

collection->excludesAll(c2 : Collection(T)) : Boolean

Does *collection* contain none of the elements of *c2* ?
post: result = c2->forAll(elem | collection->excludes(elem))

collection->isEmpty() : Boolean

Is *collection* the empty collection?
post: result = (collection->size() = 0)

collection->notEmpty() : Boolean

Is *collection* not the empty collection?
post: result = (collection->size() <> 0)

collection->sum() : T

The addition of all elements in *collection*. 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 = collection->iterate(elem; acc : T = 0 |
    acc + elem)

collection->exists(expr : OclExpression) : Boolean

Results in true if *expr* evaluates to true for at least one element in *collection*.

post: result = collection->iterate(elem; acc : Boolean = false |
    acc or expr)

collection->forAll(expr : OclExpression) : Boolean

Results in true if *expr* evaluates to true for each element in *collection*; otherwise, result is false.

post: result = collection->iterate(elem; acc : Boolean = true |
    acc and expr)

collection->isUnique(expr : OclExpression) : Boolean

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

post: let values = collection->collect(expr) in
result = res->forAll(e | values->count(e) = 1)

collection->sortedBy(expr : OclExpression) : Sequence(T)

Results in the Sequence containing all elements of *collection*. The element for which *expr* has the lowest value comes first, and so on. The type of the *expr* 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).

pre: expr.evaluationType().operations()->includes('<')
post: result->includesAll(collection) and collection->includesAll(result)

collection->iterate(expr : OclExpression) : expr.evaluationType()

Iterates over the collection. See Section 6.6.5, "Iterate Operation," on page 6-26 for a complete description. This is the basic collection operation with which the other collection operations can be described.

collection->any(expr : OclExpression) : T

Returns any element in the *collection* for which *expr* evaluates to true. If there is more than one element for which *expr* is true, one of them is returned. The precondition states that there must be at least one element fulfilling *expr*; otherwise, the result of this operation is Undefined.

pre: collection->exists(expr)
post collection->select(expr)->includes(result)

collection->one(expr : OclExpression) : Boolean

Results in true if there is exactly one element in the *collection* for which *expr* is true.
post: collection->select(expr)->size() = 1

6.8.2.2 Set

The Set is the mathematical set. It contains elements without duplicates. Features of Set, the instance of Set is called *set*.

set->union(set2 : Set(T)) : Set(T)

The union of *set* and *set2*.

post: result->forAll(elem | set->includes(elem) or set2->includes(elem))
post: set->forAll(elem | result->includes(elem))
post: set2->forAll(elem | result->includes(elem))

set->union(bag : Bag(T)) : Bag(T)

The union of *set* and *bag*.

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

set = (set2 : Set(T)) : Boolean

Evaluates to true if *set* and *set2* contain the same elements.

post: result = (set->forAll(elem | set2->includes(elem)) and
set2->forAll(elem | set->includes(elem)))

set->intersection(set2 : Set(T)) : Set(T)

The intersection of *set* and *set2*; that is, the set of all elements that are in both *set* and *set2*.

post: result->forAll(elem | set->includes(elem) and set2->includes(elem))
post: set->forAll(elem | set2->includes(elem) = result->includes(elem))
post: set2->forAll(elem | set->includes(elem) = result->includes(elem))

set->intersection(bag : Bag(T)) : Set(T)

The intersection of *set* and *bag*.
post: result = set->intersection(bag->asSet)

set – (set2 : Set(T)) : Set(T)

The elements of *set*, which are not in *set2*.

post: result->forAll(elem | set->includes(elem) and set2->excludes(elem))
post: set->forAll(elem | result->includes(elem) = set2->excludes(elem))

set->including(object : T) : Set(T)

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

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

set->excluding(object : T) : Set(T)

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

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

set->symmetricDifference(set2 : Set(T)) : Set(T)

The sets containing all the elements that are in *set* or *set2*, but not in both.

post: result->forAll(elem | set->includes(elem) xor set2->includes(elem))
post: set->forAll(elem | result->includes(elem) = set2->excludes(elem))
post: set2->forAll(elem | result->includes(elem) = set->excludes(elem))

set->select(expr : OclExpression) : Set(T)

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

post: result = set->iterate(elem; acc : Set(T) = Set{} |
    if expr then acc->including(elem) else acc endif)

set->reject(expr : OclExpression) : Set(T)

The subset of *set* for which *expr* is false.
post: result = set->select(not expr)

set->collect(expr : OclExpression) : Bag(expr.evaluationType())

The Bag of elements that results from applying *expr* to every member of *set*.

post: result = set->iterate(elem; acc : Bag(expr.evaluationType()) = Bag{} |
    acc->including(expr))

set->count(object : T) : Integer

The number of occurrences of *object* in *set*.
post: result <= 1

set->asSequence() : Sequence(T)

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

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

set->asBag() : Bag(T)

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

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

6.8.2.3 *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.

Properties of Bag, where the instance of Bag is called *bag*.

bag = (bag2 : Bag(T)) : Boolean

True if *bag* and *bag2* contain the same elements, the same number of times.

```
post: result = (bag->forAll(elem | bag->count(elem) = bag2->count(elem)) and
bag2->forAll(elem | bag2->count(elem) = bag->count(elem)) )
```

bag->union(bag2 : Bag(T)) : Bag(T)

The union of *bag* and *bag2*.

```
post: result->forAll( elem | result->count(elem) = bag->count(elem) + bag2->count(elem))
post: bag->forAll( elem | result->count(elem) = bag->count(elem) + bag2->count(elem))
post: bag2->forAll( elem | result->count(elem) = bag->count(elem) + bag2->count(elem))
```

bag->union(set : Set(T)) : Bag(T)

The union of *bag* and *set*.

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

bag->intersection(bag2 : Bag(T)) : Bag(T)

The intersection of *bag* and *bag2*.

```
post: result->forAll(elem | result->count(elem) = bag->count(elem).min(bag2->count(elem)) )
post: bag->forAll(elem |result->count(elem) = bag->count(elem).min(bag2->count(elem)) )
post: bag2->forAll(elem |result->count(elem) = bag->count(elem).min(bag2->count(elem)) )
```

bag->intersection(set : Set(T)) : Set(T)

The intersection of *bag* and *set*.

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

bag->including(object : T) : Bag(T)

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

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

bag->excluding(object : T) : Bag(T)

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

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

bag->select(expr : OclExpression) : Bag(T)

The sub-bag of *bag* for which *expr* is true.

```
post: result = bag->iterate(elem; acc : Bag(T) = Bag{} |
         if expr then acc->including(elem) else acc endif)
```

bag->reject(expr : OclExpression) : Bag(T)

The sub-bag of *bag* for which *expr* is false.

```
post: result = bag->select(not expr)
```

bag->collect(expr: OclExpression) : Bag(expr.evaluationType())

The Bag of elements that results from applying *expr* to every member of *bag*.

```
post: result = bag->iterate(elem; acc : Bag(expr.evaluationType() ) = Bag{} |
              acc->including(expr) )
```

bag->count(object : T) : Integer

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

bag->asSequence() : Sequence(T)

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

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

bag->asSet() : Set(T)

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

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

6.8.2.4 Sequence

A sequence is a collection where the elements are ordered. An element may be part of a sequence more than once.

Properties of Sequence(T), where the instance of Sequence is called *sequence*.

sequence->count(object : T) : Integer

> The number of occurrences of *object* in *sequence*.

sequence = (sequence2 : Sequence(T)) : Boolean

> True if *sequence* contains the same elements as *sequence2* in the same order.

```
post: result = Sequence{1..sequence->size()}->forAll(index : Integer |
          sequence->at(index) = sequence2->at(index))
        and
        sequence->size() = sequence2->size()
```

sequence->union (sequence2 : Sequence(T)) : Sequence(T)

> The sequence consisting of all elements in *sequence*, followed by all elements in *sequence2*.

```
post: result->size() = sequence->size() + sequence2->size()
post: Sequence{1..sequence->size()}->forAll(index : Integer |
        sequence->at(index) = result->at(index))
post: Sequence{1..sequence2->size()}->forAll(index : Integer |
        sequence2->at(index) =
                result->at(index + sequence->size() )))
```

sequence->append (object: T) : Sequence(T)

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

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

sequence->prepend(object : T) : Sequence(T)

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

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

sequence->subSequence(lower : Integer, upper : Integer) : Sequence(T)

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

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

sequence->at(i : Integer) : T

The *i-th* element of sequence.
pre : i >= 1 and i <= sequence->size()

sequence->first() : T

The first element in *sequence*.
post: result = sequence->at(1)

sequence->last() : T

The last element in *sequence*.
post: result = sequence->at(sequence->size())

sequence->including(object : T) : Sequence(T)

The sequence containing all elements of *sequence* plus *object* added as the last element.
post: result = sequence.append(object)

sequence->excluding(object : T) : Sequence(T)

The sequence containing all elements of *sequence* apart from all occurrences of *object*. The order of the remaining elements is not changed.

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

sequence->select(expression : OclExpression) : Sequence(T)

The subsequence of *sequence* for which *expression* is *true*.

```
post: result = sequence->iterate(elem; acc : Sequence(T) = Sequence{} |
        if expr then acc->including(elem) else acc endif)
```

sequence->reject(expression : OclExpression) : Sequence(T)

The subsequence of *sequence* for which *expression* is false.
post: result = sequence->select(not expr)

sequence->collect(expression : OclExpression) : Sequence(expression.evaluationType())

The Sequence of elements that results from applying *expression* to every member of *sequence*.

sequence->iterate(expr : OclExpression) : expr.evaluationType()

Iterates over the sequence. Iteration will be done from element at position 1 up until the element at the last position following the order of the sequence.

sequence->asBag() : Bag(T)

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

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

sequence->asSet() : Set(T)

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

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

6.9 Grammar

This section describes the grammar for OCL expressions. An executable LL(1) version of this grammar is available on the OCL web site. (See http://www.software.ibm.com/ad/ocl).

The grammar description uses the EBNF syntax, where “|” means a choice, “?” optionality, and “*” means zero or more times, “+” means one or more times, and expressions delimited with “/*” and “*/” are definitions described with English words or sentences. In the description of *string,* the syntax for lexical tokens from the JavaCC parser generator is used. The “~” symbol denotes that none of the symbols following may be matched. It means “everything except the following.”

```
oclFile              := ( "package" packageName
                          oclExpressions
                          "endpackage"
                        )+
packageName          := pathName
oclExpressions       := ( constraint )*
constraint           := contextDeclaration
                        ( ( "def" name? ":" letExpression* )
                          |
```

```
                              ( stereotype name? ":" oclExpression)
                            )+
contextDeclaration     := "context"
                            ( operationContext | classifierContext )
classifierContext      := (  name ":" name )
                            | name
operationContext       := name "::" operationName
                            "(" formalParameterList ")"
                            ( ":" returnType )?
stereotype             := ( "pre" | "post" | "inv" )
operationName          := name | "=" | "+" | "-" | "<"  | "<=" |
                            ">="   | ">" | "/" | "*" | "<>" |
                            "implies" | "not" | "or" | "xor" | "and"
formalParameterList    := ( name ":" typeSpecifier
                              ("," name ":" typeSpecifier )*
                            )?
typeSpecifier          := simpleTypeSpecifier
                            | collectionType
collectionType         := collectionKind
                            "(" simpleTypeSpecifier ")"
oclExpression          := (letExpression* "in")? expression
returnType             := typeSpecifier
expression             := logicalExpression
letExpression          := "let" name
                            ( "(" formalParameterList ")" )?
                            ( ":" typeSpecifier )?
                            "=" expression
ifExpression           := "if" expression
                            "then" expression
                            "else" expression
                            "endif"
logicalExpression      := relationalExpression
                            ( logicalOperator
                              relationalExpression
                            )*
relationalExpression   := additiveExpression
                            ( relationalOperator
                              additiveExpression
                            )?
additiveExpression     := multiplicativeExpression
                            ( addOperator
```

```
                                   multiplicativeExpression
                                 )*
multiplicativeExpression:= unaryExpression
                                  ( multiplyOperator
                                    unaryExpression
                                  )*
unaryExpression          := ( unaryOperator
                                   postfixExpression
                                 )
                                 | postfixExpression
postfixExpression        := primaryExpression
                                 ( ("." | "->")propertyCall )*
primaryExpression        := literalCollection
                                 | literal
                                 | propertyCall
                                 | "("  expression  ")"
                                 | ifExpression
propertyCallParameters := "(" ( declarator )?
                                 ( actualParameterList )? ")"
literal                  := string
                                 | number
                                 | enumLiteral
enumLiteral              := name "::" name ( "::" name )*
simpleTypeSpecifier      := pathName
literalCollection        := collectionKind "{"
                                 ( collectionItem
                                   ("," collectionItem )*
                                 )?
                                 "}"
collectionItem           := expression (".." expression )?
propertyCall             := pathName
                                 ( timeExpression )?
                                 ( qualifiers )?
                                 ( propertyCallParameters )?
qualifiers               := "[" actualParameterList "]"
declarator               := name ( "," name )*
                                 ( ":" simpleTypeSpecifier )?
                                 ( ";" name ":" typeSpecifier "="
                                   expression
                                 )?
                                 "|"
```

```
pathName                := name ( "::" name )*
timeExpression          := "@" "pre"
actualParameterList     := expression ("," expression)*
logicalOperator         := "and" | "or" | "xor" | "implies"
collectionKind          := "Set" | "Bag" | "Sequence" | "Collection"
relationalOperator      := "=" | ">" | "<" | ">=" | "<=" | "<>"
addOperator             := "+" |  "-"
multiplyOperator        := "*" | "/"
unaryOperator           := "-" | "not"
typeName                :=charForNameTop charForName*
name                    := charForNameTop charForName*
charForNameTop          := /* Characters except inhibitedChar
                             and ["0"-"9"]; the available
                             characters shall be determined by
                             the tool implementers ultimately.*/
charForName             := /* Characters except inhibitedChar; the
                             available characters shall be determined
                             by the tool implementers ultimately.*/
inhibitedChar      := " " | "\"" | "#" | "\'" | "(" | ")" |
                      "*" | "+" | "," | "-" | "." | "/" |
                      ":" | ";" | "<" | "=" | ">" | "@" |
                      "[" | "\\" | "]" | "{" | "|" | "}"
number                  := ["0"-"9"] (["0"-"9"])*
                           ( "." ["0"-"9"] (["0"-"9"])* )?
                           ( ("e" | "E") ( "+" | "-" )? ["0"-"9"]
                             (["0"-"9"])*
                           )?
string                  := "'"
                           (( ~["'","\\","\n","\r"] )
                            |("\\"
                              ( ["n","t","b","r","f","\\","'","\""]
                                | ["0"-"7"]
                                  ( ["0"-"7"] ( ["0"-"7"] )? )?
                              )
                            )
                           )*
                           "'"
```