7 OCL Language Description

This chapter introduces the Object Constraint Language (OCL), a formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. Note that when the OCL expressions are evaluated, they do not have side effects (i.e., their evaluation cannot alter the state of the corresponding executing system).

OCL expressions can be used to specify operations / actions that, when executed, do alter the state of the system. UML modelers can use OCL to specify application-specific constraints in their models. UML modelers can also use OCL to specify queries on the UML model, which are completely programming language independent.

Note - This chapter is informative only and not normative.

7.1 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 specification language; therefore, an OCL expression is guaranteed to be without side effects. 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 (e.g., 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, OCL expressions are not by definition 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 Chapter 11 (“The OCL Standard Library”).

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.

7.1.1 Where to Use OCL

OCL can be used for a number of different purposes:

- As a query language
- 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
- To specify target (sets) for messages and actions
- To specify constraints on operations
- To specify derivation rules for attributes for any expression over a UML model.

7.2 Introduction

7.2.1 Legend

Text written in the 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 examples 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.

7.2.2 Example Class Diagram

The diagram below is used in the examples in this chapter.

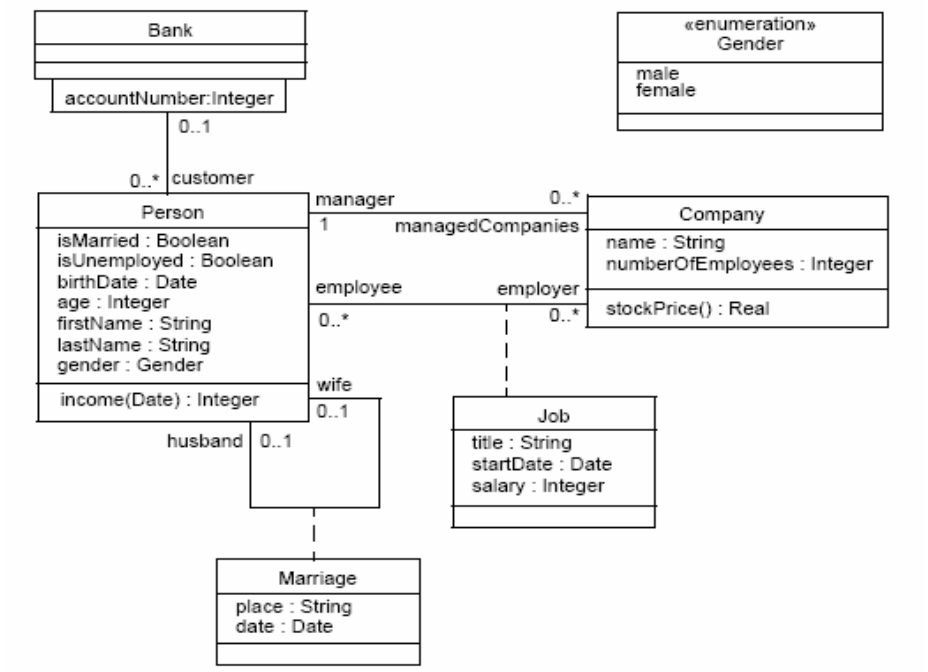

Figure 7.1 - Class Diagram Example

7.3 Relation to the UML Metamodel

7.3.1 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.

7.3.2 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.

7.3.3 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 7.1, 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 evaluating 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. For example:

```
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 1.4 metamodel, this name is a (meta-)attribute of the metaclass Constraint that is inherited from ModelElement.

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

7.3.4 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 other behavioral feature. The contextual instance *self* then is an instance of the type which 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 stereotype of constraint is shown by putting the labels ‘pre:’ and ‘post:’ before the actual Preconditions and Postconditions. For example:

```
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 the values of the attribute *name* of the metaclass Constraint that is inherited from ModelElement.

context Typename::operationName(param1 : Type1, ...): ReturnType

pre parameterOk: param1 > ...

post resultOk : result = ...

7.3.5 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 and stored in one file. This file may co-exist with a UML model as a separate entity.

7.3.6 Operation Body Expression

An OCL expression may be used to indicate the result of a query operation. This can be done using the following syntax:

context Typename::operationName(param1 : Type1, ...): ReturnType

body: -- some expression

The expression must conform to the result type of the operation. Like in the pre- and postconditions, the parameters may be used in the expression. Pre-, and postconditions, and body expressions may be mixed together after one operation context. For example:

context Person::getCurrentSpouse() : Person
pre: self.isMarried = true
body: self.mariages->select(m | m.ended = false).spouse

7.3.7 Initial and Derived Values

An OCL expression may be used to indicate the initial or derived value of an attribute or association end. This can be done using the following syntax:

context Typename::attributeName: Type

init: -- some expression representing the initial value

context Typename::assocRoleName: Type

derive: -- some expression representing the derivation rule

The expression must conform to the result type of the attribute. In the case the context is an association end the expression must conform to the classifier at that end when the multiplicity is at most one, or Set, or OrderedSet when the multiplicity may be more than one. Initial and derivation expressions may be mixed together after one context. For example:

```
context Person::income : Integer
init:   parents.income->sum() * 1% -- pocket allowance
derive:  if underAge
      then parents.income->sum() * 1% -- pocket allowance
      else job.salary              -- income from regular job
      endif
```

7.3.8 Other Types of 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. A special subclass of Expression, called ExpressionInOcl is used for this purpose. See Section 12.1, “Introduction,” on page 159 for a definition.

7.4 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 are part of the definition of OCL.

The most basic value in OCL is a value of one of the basic types. The basic types of OCL, with corresponding examples of their values, are shown in the following table.

Table 7.1 - Basic OCL types and their values

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 7.2 gives some examples of the operations on the predefined types. See Section 11.4, “Primitive Types,” on page 140 for a complete list of all operations.

Table 7.2 - Examples of operations on the predefined types

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

Collection, Set, Bag, Sequence, and Tuple are basic types as well. Their specifics will be described in the upcoming sections.

7.4.1 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.

7.4.2 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 Gender in the example model with values 'female' or 'male' they can be used as follows:

```
context Person inv: gender = Gender::male
```

7.4.3 Let Expressions

Sometimes a sub-expression is used more than once in a constraint. The *let* expression allows one to define a variable that 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.

7.4.4 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.

7.4.5 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 t*ype 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 types from the OCL Standard Library are listed in Table 7.3.

Table 7.3 - Type conformance rules

Type	Conforms to/Is a subtype of	Condition
Set(T1)	Collection(T2)	if T1 conforms to T2
Sequence(T1)	Collection(T2)	if T1 conforms to T2
Bag(T1)	Collection(T2)	if T1 conforms to T2
Integer	Real	

The conformance relation between the collection types only holds if they are collections of element types that conform to each other. See Section 7.5.13, "Collection Type Hierarchy and Type Conformance Rules," on page 23 for the complete conformance rules for collections.

Table 7.4 provides examples of valid and invalid expressions.

Table 7.4 - Valid and Invalid 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	

7.4.6 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 ("Undefined Values")).

7.4.7 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.

7.4.8 Use of Infix Operators

The use of infix operators is allowed in OCL. The operators ‘+,’ ‘-,’ ‘*.’ ‘/,’ ‘<,’ ‘>,’ ‘<>’ ‘<=’ ‘>=’ 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.

7.4.9 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
context
def
else
endif
endpackage
if
implies
```

in
inv
let
not
oper
or
package
post
pre
then
xor

7.4.10 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
```

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.

7.5 Objects and Properties

OCL expressions can refer to Classifiers, e.g., 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

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

```
context Person inv:
    self.isMarried
```

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

7.5.1 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 standard 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 by the invariant above.

Attributes may have multiplicities in a UML model. Whenever the multiplicity of an attribute is greater than 1, the result type is collection of values. Collections in OCL are described later in this chapter.

7.5.2 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 result of this operation call is a value of the return type of the operation, which is Integer in this example. If the operation has out or in/out parameters, the result of this operation is a tuple containing all out, in/out parameters and the return value. For example, if the income operation would have an out parameter *bonus*, the result of the above operation call is of type *Tuple(bonus: Integer, result: Integer)*. You can access these values using the names of the out parameters, and the keyword *result*. For example:

```
aPerson.income(aDate).bonus = 300 and
aPerson.income(aDate).result = 5000
```

Note that the out parameters need not be included in the operation call. Values for all in or in/out parameters are necessary.

Defining operations

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 (i.e., the definition may be recursive) as long as the recursion is not infinite. Inside a pre- or postcondition one can also use the parameters of the operation. The type of *result*, when the operation has no out or in/out parameters, is the return type of the operation, which is Integer in the above example. When the operation does have out or in/out parameters, the return type is a Tuple as explained above. The postcondition for the income operation with out parameter bonus may take the following form:

```
context Person::income (d: Date, bonus: Integer) : Integer
    post: result = Tuple { bonus = ...,
                result = .... }
```

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
```

7.5.3 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, for example, 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.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.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
```

7.5.4 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 AssociationEnd names" 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.

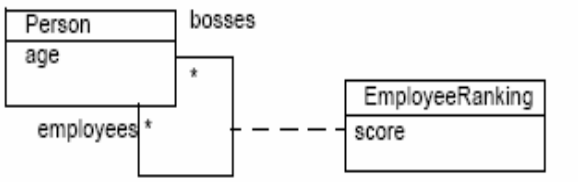

Figure 7.2 - Navigating recursive association classes

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]

7.5.5 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 (->).

7.5.6 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. The following example results in a Set(Person) containing all customers of the Bank.

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

The next example results in one Person, having account number 8764423.

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

7.5.7 Using Pathnames for Packages

Within UML, 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
```

7.5.8 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 7.3 shows an example where such a construct is needed. 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->isEmpty()
    inv:  self.oclAsType(ModelElement).source->isEmpty()
```

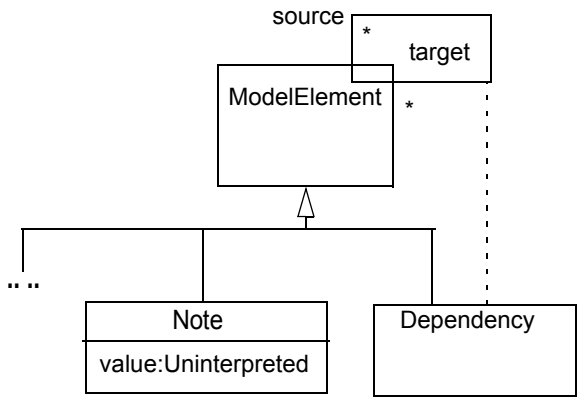

Figure 7.3 - Accessing Overridden Properties Example

7.5.9 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 *oclIsTypeOf* 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 "::".

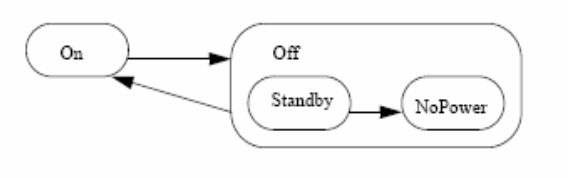

Figure 7.4 - Statemachine Example

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 colon '::,' as with nested states.

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

7.5.10 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 classes, interfaces, and enumerations 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 in the system at the time that the expression is evaluated.

7.5.11 Collections

Single navigation of an association results in a Set, combined navigations in a Bag, and navigation over associations adorned with {ordered} results in an OrderedSet. Therefore, the collection types defined in the OCL Standard Library 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 (i.e., 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.

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)
   ```

7.5.12 Collections of Collections

In UML 1.4 a collection in OCL was always flattened (i.e., a collection could never contain other collections as elements). This restriction is relieved in UML 2.0. OCL allows elements of collections to be collections themselves. The OCL Standard Library includes specific flattened operations for collections. These can be used to flatten collections of collections explicitly.

7.5.13 Collection Type Hierarchy and Type Conformance Rules

In addition to the type conformance rules in Section 7.4.5, "Type Conformance," on page 11, 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*. This is also true for *Set(Type1)/Set(Type2)*, *Sequence(Type1)/Sequence(Type2)*, *Bag(Type1)/Bag(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.

7.5.14 Previous Values in Postconditions

As stated in Section 7.3.4, "Pre- and Postconditions," on page 8, OCL can be used to specify pre- and postconditions on operations and methods in UML. In a postcondition, the expression can refer to values for each property of an object at two moments in time:

- 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 that 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
```

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 OclUndefined. Also, referring to the previous value of an object that has been created during execution of the operation results in OclUndefined.

7.5.15 Tuples

It is possible to compose several values into a *tuple*. A tuple consists of named parts, each of which can have a distinct type. Some examples of tuples are:

```
Tuple {name: String = 'John,'  age: Integer = 10}
Tuple {a: Collection(Integer) = Set{1, 3, 4}, b: String = 'foo,' c: String = 'bar'}
```

This is also the way to write tuple literals in OCL; they are enclosed in curly brackets, and the parts are separated by commas. The type names are optional, and the order of the parts is unimportant. Thus:

```
Tuple {name: String = 'John,' age: Integer = 10} is equivalent to
Tuple {name = 'John,' age = 10} and to
Tuple {age = 10, name = 'John'}
```

Also, note that the values of the parts may be given by arbitrary OCL expressions, so for example we may write:

```
context Person def:
attr statistics : Set(TupleType(company: Company, numEmployees: Integer,
                        wellpaidEmployees: Set(Person), totalSalary: Integer)) =
    managedCompanies->collect(c |
     Tuple { company: Company = c,
        numEmployees: Integer = c.employee->size(),
        wellpaidEmployees: Set(Person) = c.job->select(salary>10000).employee->asSet(),
        totalSalary: Integer = c.job.salary->sum()
       }
    )
```

This results in a bag of tuples summarizing the company, number of employees, the best paid employees, and total salary costs of each company a person manages.

The parts of a tuple are accessed by their names, using the same dot notation that is used for accessing attributes. Thus:

```
Tuple {x: Integer = 5, y: String = 'hi'}.x = 5
```

is a true, if somewhat pointless, expression. Using the definition of statistics above, we can write:

```
context Person inv:
statistics->sortedBy(totalSalary)->last().wellpaidEmployees->includes(self)
```

This asserts that a person is one of the best-paid employees of the company with the highest total salary that he manages. In this expression, both ‘totalSalary’ and ‘wellpaidEmployees’ are accessing tuple parts.

7.6 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.

7.6.1 Select and Reject Operations

Sometimes an expression using operations and navigations results in 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) )
```

7.6.2 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 which is derived from some other collection, but which contains different objects from the original collection (i.e., 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 when the source collection is a Set the resulting collection is not a Set but a Bag. Moreover, if the source collection is a Sequence or an OrderedSet, the resulting collection is a Sequence. 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()
```

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, ...))
```

7.6.3 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( age <= 65 )
    inv:    self.employee->forAll( p | p.age <= 65 )
    inv:    self.employee->forAll( p : Person | p.age <= 65 )
```

These invariants evaluate to true if the *age* property of each employee is less or equal to 65.

The *forAll* operation has an extended variant in which more than 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 : 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))
```

7.6.4 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* property of at least one employee is equal to ‘Jack.’

7.6.5 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>
```

```
    }
    return acc;
}
```

Although the Java pseudo code uses a ‘next element,’ the *iterate* operation is defined not only for Sequence, 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.

7.7 Messages in OCL

This section contains some examples of the concrete syntax and explains the finer details of the message expression. In earlier versions the phrase “actions in OCL” was used, but message was found to capture the meaning more precisely.

7.7.1 Calling operations and sending signals

To specify that communication has taken place, the hasSent (‘^’) operator is used:

```
context Subject::hasChanged()
post:  observer^update(12, 14)
```

The *observer^update(12, 14)* results in true if an update message with arguments 12 and 14 was sent to *observer* during the execution of the operation. *Update()* is either an *Operation* that is defined in the class of *observer*, or it is a *Signal* specified in the UML model. The argument(s) of the message expression (12 and 14 in this example) must conform to the parameters of the operation/signal definition.

If the actual arguments of the operation/signal are not known, or not restricted in any way, it can be left unspecified. This is shown by using a question mark. Following the question mark is an optional type, which may be needed to find the correct operation when the same operation exists with different parameter types.

```
context Subject::hasChanged()
post:  observer^update(? : Integer, ? : Integer)
```

This example states that the message update has been sent to *observer*, but that the values of the parameters are not known.

OCL also defines a special *OclMessage* type. One can get the actual OclMessages through the message operator: ^^.

```
context Subject::hasChanged()
post:  observer^^update(12, 14)
```

This results in the Sequence of messages sent. Each element of the collection is an instance of *OclMessage*. In the remainder of the constraint one can refer to the parameters of the operation using their formal parameter name from the operation definition. If the operation update has been defined with formal parameters named *i* and *j*, then we can write:

```
context Subject::hasChanged()
post: let messages : Sequence(OclMessage) = observer^^update(? : Integer, ? : Integer) in
    messages->notEmpty() and
    messages->exists( m | m.i > 0 and m.j >= m.i )
```

The value of the parameter *i* is not known, but it must be greater than zero and the value of parameter j must be larger or equal to i.

Because the ^^ operator results in an instance of *OclMessage*, the message expression can also be used to specify collections of messages sent to different targets. For an observer pattern we can write:

```
context Subject::hasChanged()
post:  let messages : Sequence(OclMessage) =
            observers->collect(o | o^^update(? : Integer, ? : Integer) ) in
    messages->forAll(m | m.i <= m.j )
```

Messages is now a set of *OclMessage* instances, where every *OclMessage* instance has one of the *observers* as a *target*.

7.7.2 Accessing result values

A signal sent message is by definition asynchronous, so there never is a return value. If there is a logical return value it must be modeled as a separate signal message. Yet, for an operation call there is a potential return value. This is only available if the operation has already returned (not necessary if the operation call is asynchronous), and it specifies a return type in its definition. The standard operation *result()* of OclMessage contains the return value of the called operation. If getMoney(...) is an operation on Company that returns a boolean, as in *Company::getMoney(amount : Integer) : Boolean*, we can write:

```
context Person::giveSalary(amount : Integer)
post: let message : OclMessage = company^getMoney(amount) in
    message.hasReturned()              -- getMoney was sent and returned
    and
    message.result() = true            -- the getMoney call returned true
```

As with the previous example we can also access a collection of return values from a collection of OclMessages. If *message.hasReturned()* is false, then *message.result()* will be undefined.

7.7.3 An example

This section shows an example of using the OCL message expression.

The Example and Problem

Suppose we have build a component, which takes any form of input and transforms it into garbage (aka encrypts it). The component *GarbageCan* uses an interface *UsefulInformationProvider* that must be implemented by users of the component to provide the input. The operation *getNextPieceOfGarbage* of *GarbageCan* can then be used to retrieve the garbled data. Figure 7.5 shows the component's class diagram. Note that none of the operations are marked as queries.

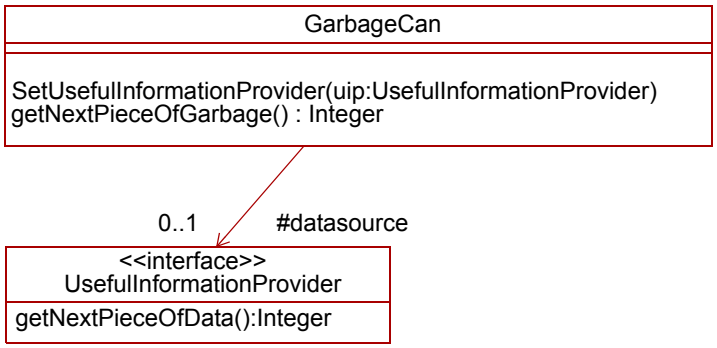

Figure 7.5 - OclMessageExample

When selling the component, we do not want to give the source code to our customers. However, we want to specify the component's behavior as precisely as possible. So, for example, we want to specify, what *getNextPieceOfGarbage* does. Note that we cannot write:

```
context GarbageCan::getNextPieceOfGarbage() : Integer
post: result = (datasource.getNextPieceOfData() * .7683425 + 10000) / 20 + 3
```

because *UsefulInformationProvider::getNextPieceOfData()* is not a query (e.g., it may increase some internal pointer so that it can return the next piece of data at the next call). Still we would like to say something about how the garbage is derived from the original data.

The solution

To solve this problem, we can use an OclMessage to represent the call to *getNextPieceOfData*. This allows us to check for the result. Note that we need to demand that the call has returned before accessing the result:

```
context GarbageCan::getNextPieceOfGarbage() : Integer
post: let message : OclMessage = datasource^^getNextPieceOfData()->first() in
    message.hasReturned()
    and
    result = (message.result() * .7683425 + 10000) / 20 + 3
```

7.8 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 7.3.3, "Invariants," on page 7, *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 it is defined to belong to the variable in the most inner scope, which is *iter1*.
- 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*. In the UML model property *name* is a property of *iter1*. However, *lastName* is a property of both *self* and *iter2*. This is ambiguous and therefore the *lastName* refers to the variable in the most inner scope, which is *iter2*.

Both of the following invariant constraints are correct, but have a different meaning:

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