Object Constraint Language B

B.4 Overview

This sppendix introduces and defines the Object Constraint Language (OCL), a formal language to express side-effect-free constraints. Users of the Unified Modeling Language and other languages can use OCL to specify constraints and other expressions attached to their models.

OCL was used in the UML Semantics chapter to specify the well-formedness rules of the UML metamodel. Each well-formedness rule in the static semantics chapters in the UML Semantics section contains an OCL expression, which is an invariant for the involved class. 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.

B.1.1 Why OCL?

In object-oriented modeling a graphical model, like a class model, is not enough for a precise and unambiguous specification. There is 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 string 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. It cannot change anything in the model. This means that the state of the system will never change because of an OCL expression, even though an OCL expression can be used to specify a state change (e.g., in a post-condition). All values for all objects, including all links, will not change. Whenever an OCL expression is evaluated, it simply delivers a value.

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 each OCL expression has a type. In a correct OCL expression, all types used must be type conformant. For example, you cannot compare an Integer with a String. Types within OCL can be any kind of Classifier within UML.

As a modeling language, all implementation issues are out of scope and cannot be expressed in OCL. Each OCL expression is conceptually atomic. The state of the objects in the system cannot change during evaluation.

B.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:

  operation = expression

  Where operation is the name of the operation and expression the constraint. Because operations may have parameters, the constraint may also have one or more parameters, as in one of the following:

  operation(a, b) = expression

  operation(a : Type1, b : Type2) = expression

  The parameters of the operation, in this example *a* and *b*, can be used in the expression at the right-hand side of the equals sign. Operations can also be described by a recursive expression. It is the modeler's task to make sure that the recursion is well defined. An operation constraint can also be read as a definition of the operation, where the right-hand side of the equals sign determines the value that the operation will return.

Within the UML Semantics chapter, OCL is used in the well-formedness rules as invariants on the meta-classes in the abstract syntax. In several places, it is also used to define 'additional' operations which are used in the well-formedness rules.

B.5 Introduction

B.2.3 Legend

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

'This is an OCL expression'

The underlined word before an OCL expression determines the context for the expression.

TypeName

'this is an OCL expression in the context of TypeName'

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

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

B.2.4 Example Class Diagram

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

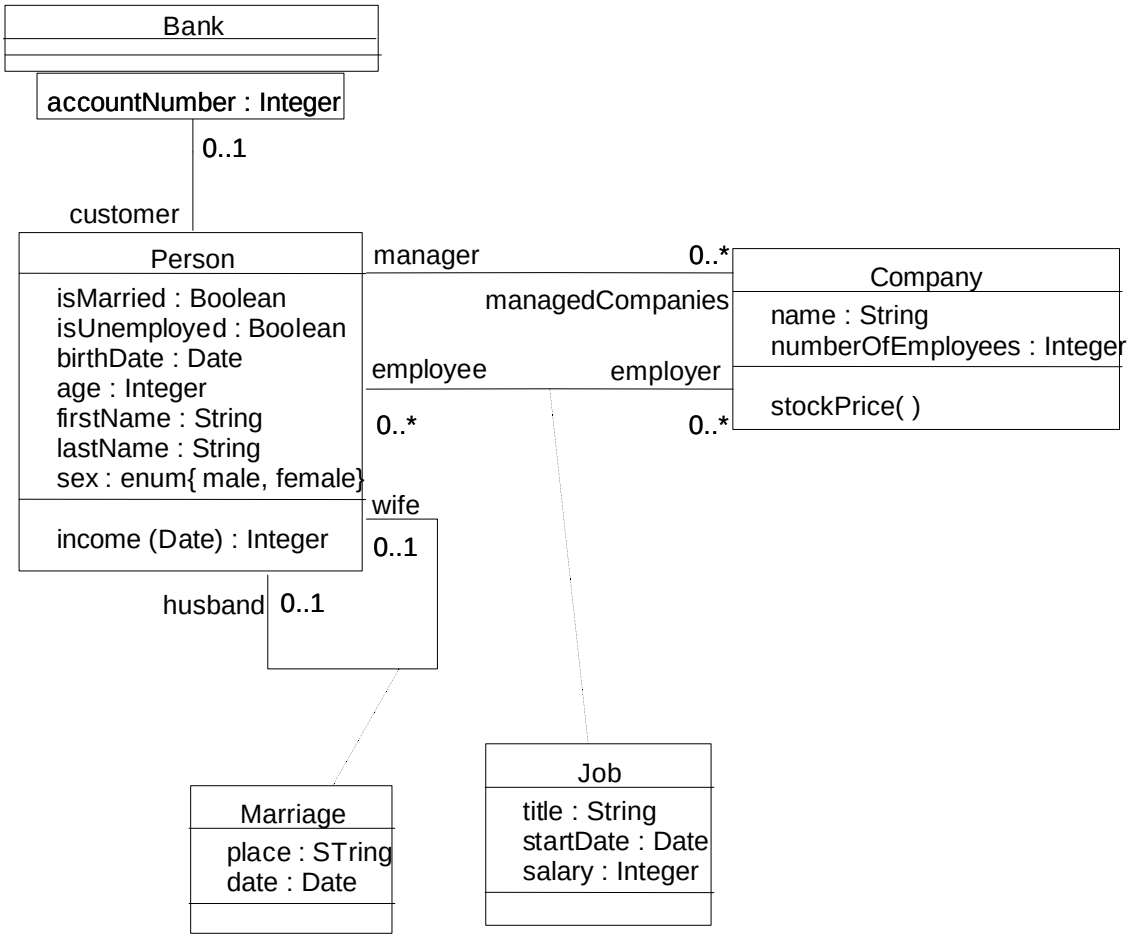

Figure 2-1 Class Diagram Example

B.6 Connection with the UML Metamodel

B.3.1 Self

Each OCL expression is written in the context of an instance of a specific type. In an OCL expression, the name *self* is used to refer to the contextual instance.

B.3.2 Invariants

The OCL expression can be part of an Invariant which is a Constraint stereotyped with «invariant». When the Invariant is associated with a Classifier, this is called the type in this document. The expression then is an invariant of the type and must be true for all instances of that type at any time. If the context is Company, then *self* refers to an instance of Company.

In the expression:

self.numberOfEmployees

self is an instance of type Company. We can see the *self* as the object from where we start the expression.

In this document, the type of the contextual instance of an OCL expression, which is part of an Invariant, is written with the name of the type underlined as follows:

Company

self.numberOfEmployees

In most cases, *self* can be left out 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:

c : Company

c.numberOfEmployees

This is identical to the previous example using self.

B.3.3 Pre and Postconditions

The OCL expression can be part of a Precondition or Postcondition, which are Constraints stereotyped with respectively «precondition» and «postcondition», the Precondition or Postcondition on Operation or Method. In this case, the expression is a pre- or postcondition on the Operation or Method. The contextual instance *self* then is of the type which owns the operation as a feature. The notation used in this document is to underline the type and operation declaration, and to put labels ‘pre:’ and ‘post:’ before Preconditions and Postconditions

Typename::operationName(parameter1 : Type1, ...): ReturnType

pre : parameter1 > …

post: result = ...

The name *self* can be used in the expression referring to the object on which the operation was called. The name *result* is the name of the returned object, if there is any. The names of the parameters (*parameter1*,) can also be used in the OCL expression. In the example diagram, we can write:

Person::income(d : Date) : Integer

post: result = ...some function of self and parameter1 ...

B.3.4 Guards

The OCL expression can be part of a Guard. In this case, *self* refers to the enclosing Classifier. No examples of guards are given in this document.

B.3.5 General Expressions

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

B.7 Basic Values and Types

In OCL, a number of basic types are predefined and available to the modeler at all time. 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 2-1.

Table 2-1 Basic Values and Types

type	values
Boolean	true, false
Integer	1, 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 2-2 gives some examples of the operations on the predefined types. See “Predefined OCL Types” on page B-23 for a complete list of all operations.

Table 2-2 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.

B.4.1 Types from the UML Model

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

B.4.2 Enumeration Types

As shown in the example diagram, new enumeration types can be defined in a model by using:

enum{ value1, value2, value3 }

The values of the enumeration (*value1*, ...) can be used within expressions.

As there might be a name conflict with attribute names being equal to enumeration values, the usage of an enumeration value is expressed syntactically with an additional # symbol in front of the value:

#value1

The type of an enumeration attribute is Enumeration, with restrictions on the values for the attribute.

B.4.3 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 its supertype.
- 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 2-3.

Table 2-3 Type Conformance Rules for Value Types

Type	Conforms to/Is subtype of
Set	Collection
Sequence	Collection
Bag	Collection
Integer	Real

The conformance relation between the collection types only holds if they are collections of element types that conform to each other. See "Collection Type Hierarchy and Type Conformance Rules" on page B-17 for the complete conformance rules for collections.

Table 2-4 provides examples of valid and invalid expressions.

Table 2-4 Valid and Invalid Expression Examples

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

B.4.4 Re-typing or Casing

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 subtype; therefore, in the example, *Type2* must be a subtype of *Type1*.

If the actual type of the object is not equal to the type to which it is re-typed, the expression is undefined (see “Undefined Values” on page B-9).

B.4.5 Precedence Rules

The precedence order for the operations in OCL is:

- dot and arrow operations have highest precedence
- unary ‘not’ and unary minus ‘-’
- ‘*’ and ‘/’
- ‘+’ and binary ‘-’
- ‘and’, ‘or’ and ‘xor’
- ‘implies’
- ‘if-then-else-endif’
- ‘<’, ‘>’, ‘<=’, ‘>=’ and ‘=’

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

B.4.6 Comment

Comments in OCL are written after two dashes. Everything after the two dashes up to and including the end of line is comment. For example:

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

B.4.7 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.

B.8 Objects and Properties

OCL expressions can refer to 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

B.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.

AType

```
self.property
```

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

B.5.2 Properties: Attributes

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

Person

self.age

The value of this expression is the value of the *age* attribute on the Person *self*. The type of this expression is the type of the attribute *age*, which is the basic type Integer.

With of attributes, and the 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 or equal to zero." This can be stated as the invariant:

Person

self.age >= 0

B.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:

Person::income (d: Date) : Integer

post: result = - - some function of d and other properties of person

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 well defined. 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 used:

Company

self.stockPrice()

B.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 (i.e., *self* is an instance of Company), we can write:

Company

```
self.manager      -- is of type      Person
self.employee     -- is of type      Set(Person)
```

The evaluation of the first expression will result in an object of type Person, because the multiplicity of the association is one. The evaluation of the second expression will result 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:

Person

```
self.employer->size
```

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

Person

```
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 Rolenames

Whenever 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.

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:

Company

```
self.manager->size -- 'self.manager' is used as Set, because the
                   -- arrow
```

```
                                -- is used to access the 'size' property on
                                      Set
                                -- This expresin result in 1

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

self.manager.age                -- 'self.manager' is used as 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:

Company

self.wife->notEmpty **implies** self.wife.sex = 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. Upon this result, one can always apply another property. 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

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

self.employee->size <= 50

[3] A marriage is between a female (wife) and male (husband)

self.wife.sex = #female **and**

self.husband.sex = #male

[4] A person cannot both have a wife and a husband

not ((self.wife->size = 1) **and** (self.husband->size = 1))

B.5.5 Navigation to Association Types

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:

Person

```
self.job
```

This 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.

B.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.

Job

```
self.employer
self.employee
```

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 (->).

B.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.

Bank

```
self.customer                -- results in a Set(Person) containing
                             -- all customers of the Bank
self.customer[8764423]       -- results in one Person, having account
                             -- number 8764423
```

If there is more than one qualifier attribute, the values are separated by commas. It is not permissible to partially specify the qualifier attribute values.

B.5.8 Using Pathnames for Packages and Properties

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
```

Whenever properties are redefined within a type, the property of the supertypes can be accessed using the same path syntax. Whenever we have a class B as a subtype of class A, and a property p1 of both A and B, we can write:

B

```
self.A::p1    -- accesses the p1 property defined in A
self.B::p1    -- accesses the p1 property defined in B
```

Figure 2-2 shows an example where such a pathname is needed.

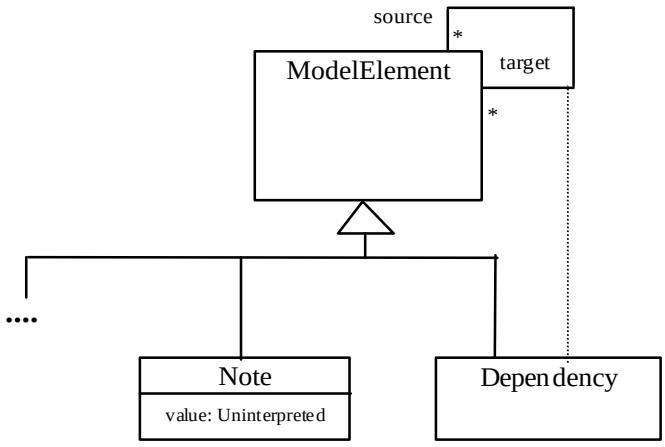

Figure 2-2 Pathname Example

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

Dependency

```
self.source
```

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 the pathname we can distinguish between them with:

Dependency

```
self.Dependency::source
self.ModelElement::source
```

B.5.9 Predefined Features on All Objects

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

```
oclType              : OclType
oclIsTypeOf(t : OclType) : boolean
oclIsKindOf(t : OclType) : boolean
```

The feature *oclType* results in the type of an object. For example, the expression

```
Person
    self.oclType
```

results in Person. The type of this is OclType, a predefined type within the OCL language.

Note – not Person, which is the type of self.

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

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

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

B.5.10 Features on Types 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.

The most important predefined feature on each type is *allInstances*, which results in the Set of all instances of the type. If we want to make sure that all instances of Person have unique names, we can write:

```
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).

B.5.11 Collections

Navigation will most often result in a collection; 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 (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. 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:

```
Company
```

self.employee

3. operations on collections may result in new collections:

collection1->union(collection2)

B.5.12 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 }

B.5.13 Collection Type Hierarchy and Type Conformance Rules

In addition to the type conformance rules in "Type Conformance" on page B-7, the following rules hold for all types, including the collection types:

- Every type Collection (X) is a subtype of OclAny. 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.

B.5.14 Previous Values in Postconditions

As stated in "Pre and Postconditions" on page B-5, 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 commercial at sign '@' followed by the keyword 'pre':

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.

Company::hireEmployee(p : Person)

post: employees = employees@pre->including(p) and
    stockprice() = stockprice@pre() + 10

The above operation can also be specified by a post and pre condition together:

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 is takes and this 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.

B.9 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.

B.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 next OCL expression specifies all the employees older than 50 years:

Company

```
self.employee->select(age > 50)
```

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:

Company

```
self.employee->select(age > 50)
```

Company

```
self.employee->select(p | p.age > 50)
```

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:

Company

self.employee.select(p : Person | p.age > 50)

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 all the employees who are **not** married:

Company

self.employee->reject(isMarried)

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))

B.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 as one of:

Company

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:

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, …)
```

B.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:

Company

```
self.employee->forAll( forename = 'Jack' )
```

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

These expressions 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.

Company

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

Company

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

B.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 which 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 forAll 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:

Company

```
self.employee->exists( forename = 'Jack' )
self.employee->exists( p | p.forename = 'Jack' )
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.’

B.6.5 Iterate Operation

The *iterate* operation is slightly more complicated, but is very generic. The operations *reject, select, forAll, exists, collect, elect* 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>
  }
}
```

B.10 Predefined OCL Types

This section contains all standard types defined within OCL, including all the features defined on those types. Its signature and a description of its semantics define each feature. Within the description, the name ‘result’ is used to refer to the value that results from evaluating the feature. 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.

B.7.1 Basic Types

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

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 access to the meta-level of the model. This can be useful for advanced modelers.

Features of OclType, 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.

OclAny

Within the OCL context, the type OclAny is the supertype of all types in the model. Features on OclAny are available on each object in all OCL expressions.

All classes in a UML model inherit all features defined on OclAny. To avoid name conflicts between features in the model and the features inherited from OclAny, all names on the features of OclAny start with ‘ocl.’ Although theoretically there may still be name conflicts, they can be avoided. One can also use the pathname construct to explicitly refer to the OclAny properties.

Features of OclAny, 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.oclType : OclType

The type of the *object.*

---

object.oclIsKindOf(type : OclType) : Boolean

True if *type* is a supertype (transitive) of the type of *object*.
post: result = type.allSuperTypes->includes(object.oclType) or
type = object->oclType

---

object.oclIsTypeOf(type : OclType) : Boolean

True if *type* is equal to the type of *object.*
post: result = (object.oclType = type)

---

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)

OclExpression

Each OCL expression itself is an object in the context of OCL. The type of the expression is OclExpression. This type and its features are used to define the semantics of features 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.

Features of OclExpression, the instance of OclExpression is called *expression*.

expression.evaluationType : OclType

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

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.

Features of Real, the instance of Real is called *r*.

r = (r2 : Real) : Boolean

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

---

r + (r1 : Real) : Real

The value of the addition of *r* and *r1*.

---

r - (r1 : Real) : Real

The value of the subtraction of *r1* from *r*.

---

r * (r1 : Real) : Real

The value of the multiplication of *r* and *r1*.

---

r * (r1 : Real) : Real

The value of *r* divided by *r1*.

---

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.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)

Integer

The OCL type Integer represents the mathematical concept of integer.

Features of Integer, the instance of Integer is called *i*.

i = (i2 : Integer) : Boolean

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

---

i + (i2 : Integer) : Integer

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

---

i + (r1 : Real) : Real

The value of the addition of *i* and *r1*.

---

i - (i2 : Integer) : Integer

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

---

i - (r1 : Real) : Real

The value of the subtraction of *r1* from *i*.

---

i * (i2 : Integer) : Integer

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

---

i * (r1 : Real) : Real

The value of the multiplication of *i* and *r1*.

---

i / (i2 : Integer) : Real

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

---

i / (r1 : Real) : Real

The value of *i* divided by *r1*.

---

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*.
post: result * i2 <= i
post: result * (i2 + 1) > i

---

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

String

The OCL type String represents ASCII strings.

Features of String, 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, string2.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*.

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*.

Enumeration

The OCL type Enumeration represents the enumerations defined in an 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)

B.7.2 Collection-Related Typed

The following sections define the features on collections (i.e., these features 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.

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 operations on Collections that have identical semantics for all collection subtypes. Some operations 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 operations is different for each subtype. These operations are not mentioned in this section.

Features of Collection, 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->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->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 addition (Integer and Real)

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

Iterates over the collection. See “Iterate Operation” on page B-22 for a complete description. This is the basic collection operation with which the other collection operations can be described.

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: T.allInstances->forAll(elem |
result->includes(elem) =
set->includes(elem) or set2->includes(elem)

---

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

The union of *set* and *bag*.

post: T.allInstances->forAll(elem |
result->count(elem) =
set->count(elem) + bag->count(elem))

---

set = (set2 : Set) : Boolean

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

post: result = T.allInstances->forAll(elem |
set->includes(elem) = set2->includes(elem))

---

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

The intersection of *set* and *set2* (i.e, the set of all elements that are in both *set* and *set2*).

post: T.allInstances->forAll(elem |
result->includes(elem) =
set->includes(elem) and set2->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: T.allInstances->forAll(elem |
result->includes(elem) =
set->includes(elem) and not set2-
>includes(elem))

---

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

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

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

---

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

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

post: T.allInstances->forAll(elem |
result->includes(elem) =
set->includes(elem) and not(elem = 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: T.allInstances->forAll(elem |
result->includes(elem) =
set->includes(elem) xor set2->includes(elem))

---

set->select(expr : OclExpression) : Set(expr.type)

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(expr.type)

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

---

set->collect(expression : OclExpression) : Bag(expression.oclType)

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

post: result = set->iterate(elem; acc : Bag(T) = 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 random order.

post: T.allInstances->forAll(elem |
result->count(elem) = set->count(elem))

---

set->asBag : Bag(T)

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

post: T.allInstances->forAll(elem |
result->count(elem) = set->count(elem))

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. Features of Bag, the instance of Bag is called *bag*.

bag = (bag2 : Bag) : Boolean

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

post: result = T.allInstances->forAll(elem |
bag->count(elem) = bag2->count(elem))

---

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

The union of *bag* and *bag2*.

post: T.allInstances->forAll(elem |
result->count(elem) =
bag->count(elem) + bag2->count(elem))

---

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

The union of *bag* and *set*.

post: T.allInstances->forAll(elem |
result->count(elem) =
bag->count(elem) + set->count(elem))

---

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

The intersection of *bag* and *bag2*.

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

---

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

The intersection of *bag* and *set*.

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

---

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

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

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

---

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

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

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

---

bag->collect(expression: OclExpression) : Bag(expression.oclType)

The Bag of elements which results from applying *expression* to every member of *bag*.

post: result = bag->iterate(elem; acc : Bag(T) = 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 random order.

post: T.allInstances->forAll(elem |
bag->count(elem) = result->count(elem))

---

bag->asSet : Set(T)

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

post: T.allInstances(elem |
bag->includes(elem) = result->includes(elem))

Sequence

A sequence is a collection where the elements are ordered. An element may be part of a sequence more than once. Features of Sequence(T), 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 all elements in *sequence*, followed by *object.*

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*.

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

---

sequence->at(i : Integer) : T

The *i-th* element of sequence.
post: i <= 0 or sequence->size < i implies result = Undefined

---

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.oclType)

The Sequence of elements which 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: T.allInstances->forAll(elem |
result->count(elem) = sequence->count(elem))

---

sequence->asSet() : Set(T)

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

post: T.allInstances->forAll(elem |
result->includes(elem) = sequence->includes(elem))

B.11 Grammar for OCL

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. In the description of the *name*, *typeName*, and *string*, the syntax for lexical tokens from the JavaCC parser generator is used. (See http://www.suntest.com/JavaCC.)

```
expression          := logicalExpression
ifExpression        := "if" expression
                           "then" expression
                           "else" expression
                           "endif"
logicalExpression   := relationalExpression
```

```
                                    ( logicalOperator
                                      relationalExpression )*
relationalExpression                := additiveExpression
                                    ( relationalOperator
                                      additiveExpression )?
additiveExpression                  := multiplicative Expression
                                    ( addOperator
                                      multiplicativeExpression )*
multiplicativeExpressin             := unaryExpression
                                    ( multiplyOperator unaryExpression )*
unaryExpression                     :=  ( unaryOperator postfixExpression )
                                    | postfixExpression
postfixExpression                   := primaryExpression ( "." | "->")
                                      featureCall )*
primaryExpression                   := literalCollection
                                    | literal
                                    | pathName timeExpression? qualifier?
                                featureCallParameters?
                                    | "(" expression ")"
                                    | ifExpression
featureCallParameters               := "(" ( declarator )?
                                      ( actualParameterList )? ")"
literal                             := <STRING> | <number> | "#" <name>
enumerationType                     := "enum" "{" "#" <name> ( "," "#" <name>
                                                )* "}"
simpleTypeSpecifier                 := pathTypeName
                                    | enumerationType
literalCollection                   := collectionKind "{" expressionListOrRange? "}"
expressionListOrRange               := expression
                                     ( ( "," expression )+
                                    | ( ".." expression )
                                     )?
featureCall                         := pathName timeExpression? qualifiers?
                                      featureCallParameters?
qualifiers                          := "[" actualParameterList "]"
declarator                          := <name> ( "," <name> )*
                                    ( ":" simpleTypeSpecifier )? "|"
pathTypeName                        := <typeName> ( "::" <typeName> )*
pathName                            := ( <typeName> | <name> )
                                     ( "::" ( <typeName> | <name> ) )*
timeExpression                      := "@" <name>
```

```
actualParameterList     := expression ( "," expression )*
logicalOperator         := "and" | "or" | "xor" | "implies"
collectionKind          := "Set" | "Bag" | "Sequence" |
                           "Collection"
relationalOperator      := "=" | ">" | "<" | ">=" | "<=" | "<>"
addOperator             := "+" | "-"
multiplyOperator        := "*" | "/"
unaryOperator           := "-" | "not"
typeName                := "A"-"Z" ( "a"-"z" | "0"-"9" | "A"-"Z"
                                   | "_")*
name                    := "a"-"z" ( "a"-"z" | "0"-"9" | "A"-"Z"
                                   | "_")*
number                  := "0"-"9" ("0"-"9")*
string                  := """ ( (~["""","\\","\n","\r"])
                                   | ("\\"
                            ( ["n","t","b","r","f","\\","""","\'"]
                                   | ["0"-"7"] ( ["0"-"7"] )?
                                   | ["0"-"3"] ["0"-"7"] ["0"-"7"]
                                    )
                                   )
                                  )*
                                 """
```