THE OBJECT CONSTRAINT LANGUAGE: AN OVERVIEW

This Appendix describes all the constructs of the Object Constraint Language (OCL) with which we can write constraints in MOF metamodels. A detailed description may be found at (OCL, 2006).

OCL was developed as a business modeling language within the IBM Insurance division and has its roots in the SYNTROPY method. It is a semi-formal language that remains easy to read and write.

OCL enables one to describe constraints on object oriented models and other object modeling artifacts. A constraint is a restriction on one or more values of (part of) an object oriented model or system. The OCL expressions are written in the context of UML diagrams and in general, specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. For instance, in UML static diagrams, OCL expressions are linked to classifiers with their properties and relationships. In this context, classifiers can be classes, interfaces, primitive types and packages.

All classifiers of UML diagrams as well as attributes, association-ends, method and operations are considered valid types in OCL expressions. OCL expressions do not have side effects; i.e. their evaluation cannot alter the state of the corresponding executing system.

Next, OCL constructs linked to Essential OCL are described.

DOI: 10.4018/978-1-61520-649-0.ch017
Context Specification

OCL expressions are written in the context of instances of specific types. The context of an expression within an UML model can be specified through a context declaration at the beginning of the respective OCL expression.

In an OCL expression the reserved word `self` is used to refer to the contextual instance.

Invariants

An invariant is a constraint associated with a Classifier that is referred to as a type, and it must be true for all instances of that type at any time. Invariants are written in a context declaration followed as the name of the type. The `self` keyword can be dropped if the context is clear.

Package Context

If it is necessary to specify explicitly in which package, due to the package in which the Classifier belongs is not clear from the environment, we can use the package context. Invariants, preconditions and post-condition constraints can be enclosed between “package” and “endpackage” statements as follows:

```
Package Package::SubPackage
context X
inv: ...some invariant...
context X::operationName (...)
pre:...some precondition...
post:...some postcondition...
endpackage
```

Objects and Properties

OCL expressions can refer to Classifiers, e.g. types, classes, interfaces, associations (acting as types) and data types. All attributes, association-ends, methods and operations without side effects that are defined on these types can be used in OCL expressions. Operations and methods are defined to be side effect free if the `isQuery` attribute of an operation is True. A property can be one of: an attribute, an association-end, an operation with `isQuery` being true and a method with `isQuery` being true.

The value of a property of an object is written in an OCL expression by a dot followed by name of the property.

An operation can be specified in OCL by means of preconditions and postconditions as follows:

```
Typename:: OperationName (parameter1:Type1,...): ReturnType
pre:_ some expression of self and parameter1
post: Result = _ some function of self and parameter1
```

`self` can be used in the expression to refer to the object on which an operation was called and the name `Result` is the name of the returned object, if there is any. The names of the `parameter (parameter1,...)` can also be used in the expression.
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, the property name has a postfix with “@” followed by the keyword “pre”. Other predefined constraints can appear: changeable, addOnly and frozen in attributes and, ordered, changeable, addOnly, frozen, xor and subset in associations.

Navigations that are Derived from Associations

An association can be navigated from a specific object to other objects connected by the association and their properties. The syntax for expressing navigations uses the role of the association-end as follows:

object.rolename

If the role name does not appear in the UML diagram, the convention is to use the name of the association-end class.

The expression object.rolename refers to a collection of objects. OCL provides the predefined collection types Collection, Set, Bag, and Sequence. They are used to specify the exact results of navigation through associations in class models. They have a large number of predefined operations on them. Properties of collection are accessed by an arrow “- >” followed by the name of the property:

collection - > operationName

Properties can be combined to make more complicated expressions. Due to an OCL expression always evaluates to a specific object of a specific type, it is possible always apply another property to the result to get a new result value. Thus, each OCL expression can be evaluated left-to-right.

Pathnames for Packages

Within MOF metamodels, types are organized in packages. A package pathname prefix enables referring to types in other packages. 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

Accessing Overridden Properties of Supertypes

Whenever properties are redefined within a type, the property of the supertypes can be accessed using the oclAsType () operation. T.oclAsType() evaluates the object asoclType

In OCL, a number of basic types are predefined. The most basic types are Boolean, Integer, Real, String, Enumerate and Tuple. OCL defines a number of operations on these predefined types.

Also, OCL provides a hierarchy of collection types including Collection, Set, Bag, OrderedSet and
Sequence.

If the name does not appear in the diagram, is associated by convention the name of the class
association-end.

All types must conform in a valid expression. A type \( t_1 \) conforms to a type \( t_2 \) when an instance of \( t_1 \)
can be substituted at each place where an instance of \( t_2 \) is expected. The basic type conformance rules
are the following ones:

- Each type conforms to each of its supertypes
- Type conformance is transitive: if \( t_1 \) conforms to \( t_2 \), and \( t_2 \) conforms to \( t_3 \), then \( t_1 \) conforms to \( t_3 \).

Predefined Types Collections

Collection is the abstract supertype whose subtypes are \( Set, OrderedSet, Bag \) and \( Sequence \). All opera-
tions on collections are denoted by the following syntax:

\(<\text{collection}>\rightarrow \langle\text{operation}\rangle\)

OCL Collections are automatically flattened; that is, a collection never contains collections but
contains only simple objects.

OCL defines standard operations on collections such as \( size, count, includes, includesAll, isEmpty \)
and \( notEmpty \). Also, OCL provides many operations on the collection types that allow us to iterate over
its elements. They are \( select, reject, collect, exists, forall \) and \( iterate \). These operations take each element
and evaluate an expression on them. Following, we describe these operations.

The \( select \) and \( reject \) operations specify a selection of a special subset from a specific collection. The
\( select \) operation gets the subset of all elements of the collection for which the expression evaluates to
True. The syntax of the select operation looks in three different forms:

\(\text{collection} \rightarrow \text{select} \ (v: \text{Type} \ | \text{boolean-expression-with-v})\)
\(\text{collection} \rightarrow \text{select} \ (v \ | \text{boolean-expression-with-v})\)
\(\text{collection} \rightarrow \text{select} \ (\text{boolean-expression})\)

The first form declares an iterator variable called \( v \). The type of this iterator variable is declared as
Type. The second form is a shorthand notation, in which the type of the iterator variable is omitted. The
third form is the shortest one. It can be used only if an explicit reference to the iterator is not needed in
the expression.

The \( reject \) operation is identical to the select operation, but with \( reject \) we get the subset of all ele-
ments of the collection for which the expression evaluates to False.

The syntax of the select operation looks also in three different forms:

\(\text{collection} \rightarrow \text{reject} \ (v: \text{Type} \ | \text{boolean-expression-with-v})\)
\(\text{collection} \rightarrow \text{reject} \ (v \ | \text{boolean-expression-with-v})\)
\(\text{collection} \rightarrow \text{reject} \ (\text{boolean-expression})\)
The collect operation specify a collection which is derived from some other collection, but which contains different objects from the original collection (i.e., it does not return sub-collections as a select or reject operation). The syntax of the select operation comes in three different forms:

\[
\text{collection -> collect } (v: \text{Type} \mid \text{expression-with-v}) \\
\text{collection -> collect } (v \mid \text{expression-with-v}) \\
\text{collection -> collect } (\text{expression})
\]

Because navigation through many objects is very common, OCL provides a shorthand notation for the collect operation. For any property name that is defined as a property on the objects in the collection, the following two expressions are equivalents:

\[
\text{collection.propertyname (par1, par2,...)} \\
\text{collection -> collect } (\text{propertyname (par1, par2,...})
\]

The exists operation in OCL allows specifying a Boolean expression, which must hold for at least one element in the collection. The syntax for the exists operation is as follows:

\[
\text{collection -> exists } (v: \text{Type} \mid \text{Boolean-expression-with-v}) \\
\text{collection -> exists } (v \mid \text{Boolean-expression-with-v}) \\
\text{collection -> exists } (\text{Boolean-expression})
\]

The iterate operation is the more generic so that operations reject, select, forAll, exists, collect can be described in terms of the iterate operation. The syntax of the iterate operation is as follows:

\[
\text{collection -> iterate } (\text{elem: T; acc: T = <expression> } \mid \text{expression-with-elem-and-acc})
\]

The variable elem is an iterator; the variable acc is an accumulator. acc gets an initial value <expression>. The iterates operation iterates over the elements of the collection and the expression-with-elem-and-acc is evaluated and its value is assigned to acc. The value of acc is built up during the iteration of the collection.

The result of the iterate operation can be calculated as is shown in the following pseudocode:

```
iterate (elem: T; acc: T2 = value) 
{ acc = value;
  for (Enumeration e = collection.elements (); e.hasMoreElements ();) 
  { elem= e.nextElement ();
    acc = <expression-with-elem-acc>
  }
  return acc;
}
```
### FROM OCL TO NEREUS: A SYSTEM OF TRANSFORMATION RULES

<table>
<thead>
<tr>
<th>Rule</th>
<th>OCL</th>
<th>NEREUS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R1</strong></td>
<td>v (variable)</td>
<td>v (variable)</td>
</tr>
<tr>
<td><strong>R2</strong></td>
<td>Type -&gt; operationName (parameter1: Type1, parameter2: Type2, ...): ReturnType</td>
<td>operationName: Type x Type1 x Type2 x ... -&gt; ReturnType</td>
</tr>
<tr>
<td><strong>R3</strong></td>
<td>Type:operationName (parameter1: Type1, parameter2: Type2, ...): ReturnType</td>
<td>operationName: Type x Type1 x Type2 X ... -&gt; ReturnType</td>
</tr>
<tr>
<td><strong>R4</strong></td>
<td>collection -&gt; operationName (expr: OCLBooleanExp, parameter1: Type1, ...): ReturnType</td>
<td>operationName: Collection x (Elem -&gt; Boolean) x Type1 x ... -&gt; ReturnType</td>
</tr>
<tr>
<td><strong>R5</strong></td>
<td>collection -&gt; operationName (expr: OCLExprType, parameter1: Type1, ...): ReturnType</td>
<td>operationName: Collection x (Elem -&gt; Type) x Type1 x ... -&gt; ReturnType</td>
</tr>
<tr>
<td><strong>R6</strong></td>
<td>v. operationName (parameters)</td>
<td>operationName (TranslateNEREUS(v), TranslateNEREUS(parameters))</td>
</tr>
<tr>
<td><strong>R7</strong></td>
<td>self.operationName (parameters)</td>
<td>operationName (c, TranslateNEREUS(parameters)) with [c] -&gt; self</td>
</tr>
<tr>
<td><strong>R8</strong></td>
<td>Type:operationName (parameters): ReturnType</td>
<td>operationName = expression operationName (c, TranslateNEREUS(parameters)) = TranslateNEREUS(expression)) with [c] -&gt; self</td>
</tr>
<tr>
<td><strong>R9</strong></td>
<td>V -&gt; operationName (parameters)</td>
<td>operationName (TranslateNEREUS(v), TranslateNEREUS(parameters))</td>
</tr>
<tr>
<td><strong>R10</strong></td>
<td>v.attributeName</td>
<td>attributeName (v)</td>
</tr>
<tr>
<td><strong>R11</strong></td>
<td>context AssociationName</td>
<td>object.roleName</td>
</tr>
<tr>
<td><strong>AXIOMS</strong></td>
<td>a: AssociationName</td>
<td>get_roleName (a, object) with [a] -&gt; Assoc</td>
</tr>
<tr>
<td><strong>R12</strong></td>
<td>expression.operationName</td>
<td>operationName (TranslateNEREUS(expression))</td>
</tr>
<tr>
<td><strong>R13</strong></td>
<td>Expression1 binaryOperator expression2 TranslateNEREUS(expression1)</td>
<td>TranslateNEREUS(binaryOperator) TranslateNEREUS(expression2)</td>
</tr>
<tr>
<td><strong>R14</strong></td>
<td>unaryOperator expression</td>
<td>TranslateNEREUS(unaryOperator) TranslateNEREUS(expression)</td>
</tr>
<tr>
<td><strong>R15</strong></td>
<td>if booleanExpression then expression1 else expression2 endif</td>
<td>IFTranslateNEREUS(booleanExpression) THENTranslateNEREUS(expression1) ELSE TranslateNEREUS(expression2)</td>
</tr>
</tbody>
</table>
Appendix C

R16  let v: Type = expression1 in expression2-with-v
     LET
     v = Translate(expression1)
     IN
     Translate(expression2-with-v)
     END-LET
     Translate(expression2-with-v)
     WHERE
     v = Translate(expression1)
     END-WHERE

R17  Collection -> operationName (v:Element| boolean-expr-with-v)

operationName::= forAll │ exists

OPERATIONS
operationName: Collection x (Element -> Boolean) -> Boolean
AXIOMS...

LET
OPERATIONS
f: Element -> Boolean
AXIOMS v: Element
f(v) = Translate(boolean-expr-with-v)
IN
operationName (collection, f)
END-LET

operationName (collection, f)
WHERE
OPERATIONS
f: Element -> Boolean
AXIOMS v: Element
f(v) = Translate(boolean-expr-with-v)
END-WHERE
Shorthand notation
operationName (v) (collection, [f (v)])

R18  Collection -> operationName (v:Element| boolean-expr-with-v)

operationName::= forAll │ exists

Collection [Element]
OPERATIONS
operationName: Collection x (Element-> Boolean) -> Boolean
AXIOMS...

LET
OPERATIONS
f: Element -> Boolean
AXIOMS v: Element
f(v) = Translate(boolean-expr-with-v)
IN
operationName (collection, f)
END-LET

operationName (collection, f)
WHERE
OPERATIONS
f: Element -> Boolean
AXIOMS v: Element
f(v) = Translate(boolean-expr-with-v)
END-WHERE
Shorthand notation
operationName (v) (collection, [f (v)])
| R19  | Collection -> operationName (v | boolean-expr)  
|      | operationName::= forAll | exists  
|      | Collection [Element]  
|      | OPERATIONS  
|      | operationName: Collection x (Element-> Boolean)-> Boolean  
|      | AXIOMS...  
|      | LET  
|      | OPERATIONS  
|      | f: Element -> Boolean  
|      | AXIOMS v: Elem  
|      | f(v) = Translate_{NEREUS} (boolean-expr)  
|      | IN  
|      | operationName (collection, f)  
|      | END-LET  
|      |-------------------------------------------------------------------------------------  
|      | operationName (collection, f)  
|      | WHERE  
|      | OPERATIONS  
|      | f: Element -> Boolean  
|      | AXIOMS v: Elem  
|      | f (v) = Translate_{NEREUS} (boolean-expr)  
|      | END-WHERE  
|      |-------------------------------------------------------------------------------------  
|      | Shorthand notation  
|      | operationName (v) (collection, [f(v)])  
| R20  | Collection -> forAll (v1, v2 | boolean-expr-with-v1-and-v2)  
|      | Collection [Element]  
|      | Collection -> forAll (v1|forAll (v2, boolean-expr-with-v1-and-v2))  
| R21  | Collection -> operationName (v: Element | boolean-expr-with-v)  
|      | operationName::= select | reject  
|      | OPERATIONS  
|      | operationName: Collection x (Element -> Boolean) -> Collection  
|      | AXIOMS  
|      | LET OPERATIONS  
|      | f: Element -> Boolean  
|      | AXIOMS v: Element  
|      | f (v) = Translate_{NEREUS} (boolean-expr-with-v)  
|      | IN  
|      | operationName (collection, f)  
|      | END-LET  
|      | operationName (collection, f)  
|      | WHERE  
|      | OPERATIONS  
|      | f: Element -> Boolean  
|      | AXIOMS v: Element  
|      | f (v) = Translate_{NEREUS} (boolean-expr-with-v)  
|      | END-WHERE  
|      |-------------------------------------------------------------------------------------  
|      | Shorthand notation  
|      | operationName (v) (collection, [f(v)])  
|
Appendix C

R22  Collection-> operationName (v | boolean-expr-with-v)
    operationName ::= select | reject
    Collection[Element]

    OPERATIONS
    operationName: Collection x (Element-> Boolean) -> Collection

    AXIOMS
    LET OPERATIONS
    f: Element -> Boolean
    AXIOMS v: Element
    f (v)= Translate_nereus(boolean-expr-with-v)
    IN
    operationName (collection, f)
    END-LET
    operationName (collection, f)
    WHERE
    OPERATIONS
    f: Element -> Boolean
    AXIOMS v: Element
    f (v)= Translate_nereus(boolean-expr-with-v)
    END-WHERE

    Shorthand notation
    operationName (v) (collection, [f (v)])

R23  Collection-> operationName (boolean-expr)
    operationName ::= select | reject
    Collection [Element]

    OPERATIONS
    operationName: Collection x (Element-> Boolean) -> Collection

    AXIOMS
    LET OPERATIONS
    f: Element -> Boolean
    AXIOMS v: Element
    f (v)= Translate_nereus(boolean-expr)
    IN
    operationName (collection, f)
    END-LET
    operationName (collection, f)
    WHERE
    OPERATIONS
    f: Element -> Boolean
    AXIOMS v: Element
    f (v)= Translate_nereus(boolean-expr)
    END-WHERE

    Shorthand notation
    operationName (v) (collection, [f (v)])
| R24 | Collection -> collect (v: Element | expression-with-v)

Let $\text{Type(expression-with-v)}$ be $S$

**OPERATIONS**
collect: Collection x (Element -> Boolean) -> Collection

**AXIOMS**

**LET**

**OPERATIONS**
f: Element -> S

**AXIOMS v: Element**
f (v)=\( \text{Translate NEREUS} \text{(expr-with-v)} \)

**IN**
collect (collection, f)

**END-LET**
collect (collection, f)

**WHERE**

**OPERATIONS**
f: Element -> S

**AXIOMS v: Element**
f (v)=\( \text{Translate NEREUS} \text{(expr-with-v)} \)

**END-WHERE**

---

**Shorthand notation**
Collect (v) (collection, [f (v)])

| R25 | Collection -> collect (v: Element | expression-with-v)

Let $\text{Type(expression-with-v)}$ be $S$

**Collection [Element]**

**OPERATIONS**
collect: Collection x (Element -> Boolean) -> Collection

**AXIOMS**

**LET**

**OPERATIONS**
f: Element -> S

**AXIOMS v: Element**
f (v)=\( \text{Translate NEREUS} \text{(expr-with-v)} \)

**IN**
collect (collection, f)

**END-LET**
collect (collection, f)

**WHERE**

**OPERATIONS**
f: Element -> S

**AXIOMS v: Element**
f (v)=\( \text{Translate NEREUS} \text{(expr-with-v)} \)

**END-WHERE**

---

**Shorthand notation**
Collect (v) (collection, [f (v)])
### Appendix C

**R26**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection -&gt; collect</td>
<td>(v: Element</td>
</tr>
</tbody>
</table>

Let Type(expression-with-v) be S

**Collection [Element]**

**OPERATIONS**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collect: Collection x (Element -&gt; Boolean) -&gt; Collection</td>
<td></td>
</tr>
</tbody>
</table>

**AXIOMS**

**LET**

**OPERATIONS**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f: Element -&gt; S</td>
<td></td>
</tr>
</tbody>
</table>

**AXIOMS v: Element**

f (v)= \( \text{Translate}_{\text{NEREUS}} \) (expr-with-v)

**IN**

collect (collection, f)

**END-LET**

collect (collection, f)

**WHERE**

**OPERATIONS**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f: Element -&gt; S</td>
<td></td>
</tr>
</tbody>
</table>

**AXIOMS v: Element**

f (v)= \( \text{Translate}_{\text{NEREUS}} \) (expr-with-v)

**END-WHERE**

---

**Shorthand notation**

Collect (v)(collection, [f (v)])

---

**R27**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection -&gt; iterate</td>
<td>(v: Element; acc: Type = exp</td>
</tr>
</tbody>
</table>

**OPERATIONS**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iterate: Collection x (Element x Acc: ANY) x -&gt; Acc</td>
<td></td>
</tr>
</tbody>
</table>

**AXIOMS**

**LET**

**OPERATIONS**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f: Element x Type -&gt; Type</td>
<td></td>
</tr>
</tbody>
</table>

base: -> Type

**AXIOMS v: Element; acc: Type**

f (v, acc)= \( \text{Translate}_{\text{NEREUS}} \) (expr-with-v-and-acc)

base = \( \text{Translate}_{\text{NEREUS}} \) (exp)

**IN**

iterate (collection, f, base)

**END-LET**

iterate (collection, f, base)

**WHERE**

**OPERATIONS**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f: Element x Type -&gt; Type</td>
<td></td>
</tr>
</tbody>
</table>

base: -> Type

**AXIOMS v: Element; acc: Type**

f (v, acc)= \( \text{Translate}_{\text{NEREUS}} \) (expr-with-v-and-acc)

base = \( \text{Translate}_{\text{NEREUS}} \) (exp)

**END-WHERE**

---

**R28**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type::operationName (par1: Type1,…): ReturnType</td>
<td></td>
</tr>
</tbody>
</table>

| pre | expression-with-self - or-attribute-or--par1..pari |

| pre | Translate_{NEREUS} (expression-with-self -or-attribute-or--par1..pari) |

| with | [self] -> t; attribute@pre -> attribute(t), par1 | -> t1;...pari | -> ti |

---

**R29**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type::operationName (par1: Type1,…): ReturnType</td>
<td></td>
</tr>
</tbody>
</table>

| post | expression-with-self -or-attribute-or self |

**OPERATIONS**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translate_{NEREUS} Type::operationName (par1: Type1,…): ReturnType</td>
<td></td>
</tr>
</tbody>
</table>

**AXIOMS**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translate_{NEREUS} (expression-with-self -or-attribute@pre-or result)</td>
<td></td>
</tr>
</tbody>
</table>

<p>| With | [self] -&gt; t : attribute@pre | -&gt; attribute (t) ; result | -&gt; operationName (t, par1...) |</p>
<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| **R30** | Collection → operationName (parameterList): Boolean  
**post**: result = collection → forAll (elem: Element | bool-exp-with-elem)  
**OPERATIONS**  
Translate_{\text{NEREUS}} (collection → operationName (parameterList): Boolean)  
**AXIOMS** c: Collection; elem: Element; ...  
operationName (create, parameterList) = TRUE  
operationName (add(c, elem), parameterList) =  
operationName (c, parameterList) AND Translate_{\text{NEREUS}} (bool-exp-with-elem) |
| **R31** | Collection → operationName (parameterList): Boolean  
**post**: result = collection → exists (elem: Element | boolean-expression-with-elem)  
**OPERATIONS**  
Translate_{\text{NEREUS}} (collection → operationName (parameterList): Boolean)  
**AXIOMS** c: Collection; elem: Element; ...  
operationName (create, parameterList) = FALSE  
operationName (add(c, elem), parameterList) =  
operationName (c, parameterList) OR Translate_{\text{NEREUS}} (boolean-expression-with-elem) |
| **R32** | Sequence → operationName (parameterList): Boolean  
**post**: result = Sequence { 1 ..sequence → size } →  
forAll (index: Integer | boolean-exp-with-index)  
**OPERATIONS**  
Translate_{\text{NEREUS}} (sequence → operationName (parameterList): Boolean)  
**AXIOMS** s: Sequence; index: Nat; ...  
operationName (s, parameterList) = (1 ≤ index ≤ size (s)) implies  
Translate_{\text{NEREUS}} (boolean-exp-with-index) |
| **R33** | Sequence { 1 ..sequence → size } →  
forAll (index: Integer | boolean-exp-with-index)  
**AXIOMS** s: Sequence; index: Nat; ...  
(1 ≤ index ≤ size (s)) implies Translate_{\text{NEREUS}} (boolean-exp-with-index) |
| **R34** | Collection -> operationName (t1:T1; t2: T2;...): Boolean  
**post**: result = collection -> iterate (elem: Element; acc: Boolean = exp | bool-exp-with-elem-and-acc)  
**OPERATIONS**  
Translate_{\text{NEREUS}} (collection -> operationName (t1:T1; t2: T2;...): Boolean)  
**AXIOMS** c: Collection; elem: Element; t1:T1; t2:T2  
operationName (create, t1, t2) = Translate_{\text{NEREUS}} (exp)  
operationName (add (c, elem), t1, t2,.....) =  
Translate_{\text{NEREUS}} (bool-exp-with-elem-and-acc)  
With [acc |-> operationName (c, t1, t2,....)] |
### Appendix C

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| **R35** | $\text{collection} \rightarrow \text{operationName} (t_1: T_1; t_2: T_2; \ldots): \text{returnType}$  
post: $\text{result} = \text{collection} \rightarrow \text{iterate}(\text{elem}: \text{Element}; \text{acc}: \text{Type} = \text{exp} | \text{expr-with-elem-and-acc})$  
OPERATIONS  
$\text{Translate}_{\text{NEREUS}} (\text{collection} \rightarrow \text{operationName} (t_1: T_1; t_2: T_2; \ldots): \text{returnType})$  
AXIOMS $\text{e}: \text{Collection}; t_1: T_1; t_2: T_2,$  
LET  
OPERATIONS  
g: $\text{Element} \times \text{Type} \rightarrow \text{Type}$  
base: $\rightarrow \text{Type}$  
AXIOMS $\text{e}: \text{Element}; \text{acc}: \text{Type}$  
g ($\text{e}, \text{acc}) = \text{Translate}_{\text{NEREUS}} (\text{expr-with-elem-and-acc})$  
base = $\text{Translate}_{\text{NEREUS}} (\text{expr})$  
IN  
operationName ($\text{c}, t_1, t_2, \ldots$) = $\text{iterate} (\text{c}, g, \text{base})$  
END-LET  
AXIOMS $\text{c}: \text{Collection}; t_1: T_1; t_2: T_2,$  
operationName ($\text{c}, t_1, t_2, \ldots$) = $\text{iterate} (\text{c}, g, \text{base})$  
WHERE  
OPERATIONS  
g: $\text{Element} \times \text{Type} \rightarrow \text{Type}$  
base: $\rightarrow \text{Type}$  
AXIOMS $\text{e}: \text{Element}; \text{acc}: \text{Type}$  
g ($\text{e}, \text{acc}) = \text{Translate}_{\text{NEREUS}} (\text{expr-with-elem-and-acc})$  
base = $\text{Translate}_{\text{NEREUS}} (\text{expr})$  
END-WHERE |
| **R36** | $\text{sequence} \rightarrow \text{operationName} (t_1: T_1; t_2: T_2; \ldots): \text{returnType}$  
post: $\text{result} = \text{sequence} \rightarrow \text{iterate}(\text{elem}: \text{Element}; \text{acc}: \text{Type} = \text{expr} | \text{expr-with-elem-and-acc})$  
OPERATIONS  
$\text{Translate}_{\text{NEREUS}} (\text{sequence} \rightarrow \text{operationName} (t_1: T_1; t_2: T_2; \ldots): \text{returnType})$  
AXIOMS $\text{s}: \text{Sequence}; \text{elem}: \text{Element}; t_1: T_1; t_2: T_2;\ldots,$  
operationName ($\text{create}, t_1, t_2, \ldots$) = $\text{Translate}_{\text{NEREUS}} (\text{expr})$  
operationName ($\text{add} (\text{s}, \text{elem}), t_1, t_2, \ldots$) = $\text{Translate}_{\text{NEREUS}} (\text{expr-with-elem-and-acc})$  
with $[\text{acc} \mapsto \text{operationName} (\text{s}, t_1, t_2, \ldots)]$ |
| **R37** | $\text{bag} \rightarrow \text{operationName} (t_1: T_1; t_2: T_2; \ldots): \text{returnType}$  
post: $\text{result} = \text{bag} \rightarrow \text{iterate}(\text{elem}: \text{Element}; \text{acc}: \text{Type} = \text{expr} | \text{expr-with-elem-and-acc})$  
OPERATIONS  
$\text{Translate}_{\text{NEREUS}} (\text{bag} \rightarrow \text{operationName} (t_1: T_1; \ldots): \text{returnType})$  
AXIOMS $\text{b}: \text{Bag}; \text{elem}: \text{Element}; t_1: T_1; t_2: T_2;\ldots,$  
operationName ($\text{create}, t_1, t_2, \ldots$) = $\text{Translate}_{\text{NEREUS}} (\text{expr})$  
operationName ($\text{add} (\text{b}, \text{elem}), t_1, t_2,\ldots$) = $\text{Translate}_{\text{NEREUS}} (\text{expr-with-elem-and-acc})$  
with $[\text{acc} \mapsto \text{operationName} (\text{b}, t_1, t_2,\ldots)]$ |
**Appendix C**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| **R38** | Collection -> operationName (t1:T1; t2: T2;...): returnType  
post: result = collection -> iterate (elem:Element; acc: Type = exp | expr-with-elem-and-acc)  
OPERATIONS  
TranslateNEREUS (collection -> op (t1:T1; t2: T2;...): returnType)  
AXIOMS c: Collection ; t1: T1; t2: T2;...  
LET  
OPERATIONS  
g: Element x Type -> Type  
base: -> Type  
AXIOMS elem: Element; acc: Type  
g (elem, acc) = TranslateNEREUS ( expr-with-elem-and-acc)  
base = TranslateNEREUS ( expr)  
IN  
operationName (c, t1, t2,...) = iterate (c, g, base)  
END-LET  
AXIOMS c: Collection ; t1: T1; t2: T2;...  
operationName (c, t1, t2,...) = iterate (c, g, base)  
WHERE  
OPERATIONS  
g: Element x Type -> Type  
base: -> Type  
AXIOMS elem: Element; acc:Type  
g (elem,acc) = TranslateNEREUS ( expr-with-elem-and-acc)  
base = TranslateNEREUS ( expr)  
END-WHERE  
**R39** | sequence → operationName (t1: T1,t2: T2,..): returnType  
post: result = sequence→iterate (elem: Element; acc: Type = expr | expr-with-elem-and-acc)  
OPERATIONS:  
TranslateNEREUS (sequence → operationName (t1: T1, t2: T2,..): returnType)  
AXIOMS s: Sequence; elem: Element; t1:T1; t2: T2, ...  
operationName (create, t1, t2,..) = TranslateNEREUS (expr)  
opperationName (add (s, elem), t1, t2,.....) = TranslateNEREUS (expr-with-elem-and-acc)  
with [acc |-> operationName (s, t1, t2,...)]  
**R40** | bag → operationName (t1: T1,t2: T2,..): returnType  
post: result = bag → iterate (elem: Element; acc: Type = expr | expr-with-elem-and-acc)  
OPERATIONS:  
TranslateNEREUS (bag→operationName (t1:T1, ..):returnType)  
AXIOMS b: Bag; elem: Element; t1: T1; t2: T2, ...  
opperationName (create, t1, t2,..) = TranslateNEREUS (expr)  
opperationName (add (b, elem), t1, t2,.....) = TranslateNEREUS (expr-with-elem-and-acc)  
With [acc|->operationName (b, t1, t2,...)]  
**R41** | set → operationName (t1: T1, t2: T2, ..): returnType  
post: result = set → iterate (elem: Element; acc: Type = expr | expr-with-elem-and-acc)  
OPERATIONS:  
TranslateNEREUS (set→operationName (t1:T1, t2: T2, ..): returnType)  
AXIOMS s: Set; elem: Elem; t1:T1; t2:T2,...  
opperationName (create, t1,t2,...) = TranslateNEREUS (expr)  
opperationName (add (s, elem), tl, t2,..) = TranslateNEREUS (expr-with-elem-and-acc)  
With [acc |-> operationName (excluding (s, e), t1, t2,...)]
### Appendix C

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| R42 | \( T \rightarrow \text{operationName} (\text{parameterList}): \text{returnType} \)  
**Post** | \( \text{result binary-operator expr} \)  
**Operations** | \( Translate_{\text{NEREUS}} (T \rightarrow \text{operationName} (\text{parameterList}): \text{returnType}) \)  
**Axioms** | \( \text{operationName} (t, \text{parameterList}) \)  
**TranslateNEREUS** | \( \text{binary-operator} \)  
**TranslateNEREUS** | \( \text{expr} \) |
| R43 | \( T \rightarrow \text{operationName} (\text{parameterList}): \text{returnType} \)  
**Post** | \( \text{result } \rightarrow \text{iteratorOperation (iteratorExpression)} \)  
**Operations** | \( Translate_{\text{NEREUS}} (T \rightarrow \text{operationName} (\text{parameterList}): \text{returnType}) \)  
**Axioms** | \( \text{iteratorOperation} (\text{operationName} (t, \text{parameterList}), \text{TranslateNEREUS} (\text{iteratorExpression})) \)
| R44 | \( T \rightarrow \text{operationName} (\text{parameterList}): \text{returnType} \)  
**Post** | \( \text{expression1 = expression2} \)  
**Operations** | \( Translate_{\text{NEREUS}} (T \rightarrow \text{operationName} (\text{parameterList}): \text{returnType}) \)  
**Axioms** | \( \text{TranslateNEREUS} (\text{expression1}) = \text{TranslateNEREUS} (\text{expression2}) \)
| R45 | \( T \rightarrow \text{operationName} (\text{parameterList}): \text{returnType} \)  
**Post** | \( \text{expression} \)  
**Operations** | \( Translate_{\text{NEREUS}} (T \rightarrow \text{operationName} (\text{parameterList}): \text{returnType}) \)  
**Axioms** | \( \text{TranslateNEREUS} (\text{expression}) \)
| R46 | \( T \rightarrow \text{operationName} (v: \text{Type} | \text{bool-expr-with-v}) \)  
**OperationName::= forAll | exists | select | reject \( T::= \text{Collection} | \text{Set} | \text{OrderedSet} | \text{Bag} \)  
**opérationName\((v) (\text{Translate}_{\text{NEREUS}} (T), \text{Translate}_{\text{NEREUS}} (\text{bool-expr-with-v}))\) |
| R47 | \( T \rightarrow \text{collect (v: type | v.property)} \)  
**Collect** | \( v \rightarrow \text{collect (property)} \)  
**Calculate** | \( \text{collect (v) (\text{Translate}_{\text{NEREUS}} (T), \text{Translate}_{\text{NEREUS}} (\text{v.property}))} \)
| R48 | \( c \rightarrow \text{property (Shorthand notation)} \)  
**Calculate** | \( c \rightarrow \text{collect (property)} \)  
**Calculate** | \( \text{collect (v) (\text{Translate}_{\text{NEREUS}} (c), \text{Translate}_{\text{NEREUS}} (\text{property}))} \)
| R49 | \( T \rightarrow \text{iterate (e: Element; acc: Type = expr | boolean-expr-with-e)} \)  
**Iterate** | \( \text{iterate (v) (\text{Translate}_{\text{NEREUS}} (T), \text{Translate}_{\text{NEREUS}} (\text{boolean-expr-with-e}), \text{Translate}_{\text{NEREUS}} (\text{expr}))} \)
| R50 | \( \text{Set} \{ \} \)  
**CreateSet** | \( \text{createSet} \)  
**CreateSet** | \( \text{createOrderedSet} \)  
**CreateSequence** | \( \text{createSequence} \)  
**CreateBag** | \( \text{createBag} \)  
**CreateSequence** | \( \text{Set \{ e1, e2, ..., ei\}} \)  
**CreateBag** | \( \text{Including (including (… including (createSet, e1), ..., e2), c1))} \)  
**CreateSequence** | \( \text{OrderedSet \{ e1, e2, ..., ei\}} \)  
**CreateSequence** | \( \text{Including (including (… including (createOrderedSet, e1), ..., e2), c1))} \)  
**CreateSequence** | \( \text{Sequence \{ e1, e2, ..., ei\}} \)  
**CreateBag** | \( \text{Including (including (… including (createSequence, e1), ..., e2), c1))} \)  
**CreateBag** | \( \text{Bag \{ e1, e2, ..., ei\}} \)  
**CreateBag** | \( \text{Including (including (… including (createBag, e1), ..., e2), c1))} \)
| R51 | \( \text{Packagename::rolename} \)  
**Packagename::rolename** |
| R52 | \( \text{Let } v \text{ be enum (e1, e2, ..., ei, ...)} \)  
**V** | \( v = \# e1 \)  
**V** | \( v = \# ei \)