Appendix A

OCL 2.0 Grammar

In this appendix we summarise the concrete syntax of OCL [113] using an extended Backus-Naur format [8]. The grammar in [113] is different from the grammar presented here. It is incomplete and context-sensitive parser. The grammar described here is a context-free grammar suitable for predictive parsers with 3 tokens of look-ahead. The start symbols are ⟨file⟩ and ⟨expression⟩. The reserved keywords of OCL are given in Table A.1:

<table>
<thead>
<tr>
<th>and</th>
<th>context</th>
<th>def</th>
<th>derive</th>
<th>else</th>
</tr>
</thead>
<tbody>
<tr>
<td>endif</td>
<td>endpackage</td>
<td>false</td>
<td>if</td>
<td>implies</td>
</tr>
<tr>
<td>in</td>
<td>init</td>
<td>inv</td>
<td>let</td>
<td>null</td>
</tr>
<tr>
<td>not</td>
<td>or</td>
<td>package</td>
<td>post</td>
<td>pre</td>
</tr>
<tr>
<td>then</td>
<td>true</td>
<td>xor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table A.1: Reserved keywords

A.1 Literals

We define the literals of OCL. All white-space characters are token delimiters.

⟨literal⟩ ::= ⟨primitive-literal⟩ | ⟨collection-literal⟩ | ⟨tuple-literal⟩
⟨primitive-literal⟩ ::= ⟨boolean-literal⟩ | ⟨integer-literal⟩ | ⟨real-literal⟩
| ⟨string-literal⟩ | null
⟨boolean-literal⟩ ::= true | false
⟨integer-literal⟩ ::= 0..9(0..9)
⟨real-literal⟩ ::= ⟨integer-literal⟩.(⟨integer-literal⟩[e E][−](⟨integer-literal⟩))
⟨string-literal⟩ ::= ’⟨characters⟩’
⟨identifier⟩ ::= ⟨letter⟩ | ⟨identifier⟩ | ⟨identifier⟩
⟨path-name⟩ ::= ⟨identifier⟩ | ⟨path-name⟩ : ⟨identifier⟩
Appendix A  OCL 2.0 Grammar

\[
\begin{align*}
\text{(collection-literal)} & ::= \text{(identifier)} \{\text{(collection-literal-part)}\} \\
\text{(collection-literal-part)} & ::= \text{(expression)} \mid \text{(expression)} . \text{(expression)} \\
\text{(tuple-literal)} & ::= \text{Tuple}[\text{variable} \rightarrow \text{decl}]
\end{align*}
\]

A.2 Files

The first start symbol of the grammar is \text{<file>}. It is used to parse a separate OCL file.

\[
\begin{align*}
\text{<file>} & ::= \text{<package-declaration>}\text{<file>} \\
& \quad \mid \text{<context-declaration>}\text{<file>} \mid \epsilon \\
\text{<package-declaration>} & ::= \text{package}\text{(path-name)}\{\text{(constraint-declaration)}\} \\
& \quad \text{endpackage} \\
\text{<constraint-declaration>} & ::= \{\text{(context-declaration)}\text{(constraint)}\} \\
\text{<context-declaration>} & ::= \text{context}\text{(classifier-context)} \\
& \quad \mid \text{context}\text{(operation-context)} \\
\text{<classifier-context>} & ::= \text{path-name} \mid \text{<identifier>}\text{:}\text{(path-name)} \\
\text{<operation-context>} & ::= \text{path-name}\{\{\text{(formal-parameter)}\}\} \\
& \quad \mid \text{path-name}\{\{\text{(formal-parameter)}\}\} :\text{<type>} \\
\text{<constraint>} & ::= \text{<stereo-type>}\{\text{(identifier)}\} :\{\text{(expression)}\} \\
& \quad \mid \text{def}\{\text{(identifier)}\} :\{\text{(let-expression)}\} \\
\text{<stereo-type>} & ::= \text{inv} \mid \text{post} \mid \text{pre}
\end{align*}
\]

A.3 Expressions

The second start symbol of the grammar is \text{<expression>}. It is used to parse constraint associated to model elements in XMI files.

\[
\begin{align*}
\text{<expression>} & ::= \text{<logical-implies-expression>} \mid \text{<let-in-expression>} \\
\text{<let-in-expression>} & ::= \text{let}\{\text{(identifier)}\} :\{\{\text{(formal-parameter)}\}\} :\{\text{(type)}\} \\
& \quad = \{\text{(expression)}\} \\
\text{<logical-implies-expression>} & ::= \text{<logical-xor-expression>} \\
& \quad \mid \text{<logical-implies-expression>}\text{implies} \text{<logical-xor-expression>} \\
\text{<logical-xor-expression>} & ::= \text{<logical-or-expression>} \\
& \quad \mid \text{<logical-xor-expression>}\text{xor}\text{<logical-or-expression>}
\end{align*}
\]

168
A.3 Expressions

\[
\begin{align*}
\langle \text{logical-or-expression} \rangle & ::= \langle \text{logical-and-expression} \rangle \\
& \quad \mid \langle \text{logical-or-expression} \rangle \text{or} \langle \text{logical-and-expression} \rangle \\
\langle \text{logical-and-expression} \rangle & ::= \langle \text{relational-expression} \rangle \text{and} \langle \text{relational-expression} \rangle \\
\langle \text{relational-expression} \rangle & ::= \langle \text{add-expression} \rangle \\
& \quad \mid \langle \text{add-expression} \rangle \langle \text{relational-operator} \rangle \langle \text{add-expression} \rangle \\
\langle \text{relational-operator} \rangle & ::= < \mid \leq \mid > \mid \geq \mid <> \mid = \\
\langle \text{add-expression} \rangle & ::= \langle \text{mul-expression} \rangle \\
& \quad \mid \langle \text{add-expression} \rangle + \langle \text{mul-expression} \rangle \\
& \quad \mid \langle \text{add-expression} \rangle - \langle \text{mul-expression} \rangle \\
\langle \text{mul-expression} \rangle & ::= \langle \text{unary-expression} \rangle \\
& \quad \mid \langle \text{mul-expression} \rangle \ast \langle \text{unary-expression} \rangle \\
& \quad \mid \langle \text{mul-expression} \rangle / \langle \text{unary-expression} \rangle \\
\langle \text{unary-expression} \rangle & ::= \langle \text{primary-expression} \rangle \mid - \langle \text{unary-expression} \rangle \\
& \quad \mid \text{not} \langle \text{unary-expression} \rangle \\
\langle \text{primary-expression} \rangle & ::= \langle \text{literal} \rangle \mid (\langle \text{expression} \rangle) \mid \langle \text{simple-property-call} \rangle \\
& \quad \mid \langle \text{postfix-expression} \rangle \mid \langle \text{operation-call} \rangle \\
\langle \text{simple-property-call} \rangle & ::= \langle \text{operation-name} \rangle[@\text{pre}]\langle \text{qualifiers} \rangle \\
& \quad \langle \text{property-call-params} \rangle \\
\langle \text{postfix-expression} \rangle & ::= \langle \text{primary-expression} \rangle . \langle \text{property-call} \rangle \\
& \quad \langle \text{primary-expression} \rangle \rightarrow \langle \text{property-call} \rangle \\
& \quad \langle \text{primary-expression} \rangle \langle \text{message-call} \rangle \\
& \quad \langle \text{primary-expression} \rangle \langle \text{message-call} \rangle \\
\langle \text{property-call} \rangle & ::= \langle \text{operation-name} \rangle[@\text{pre}]\langle \text{qualifiers} \rangle \\
& \quad \langle \text{property-call-params} \rangle \\
\langle \text{property-call-params} \rangle & ::= (\langle \text{declarator} \rangle(\langle \text{expression} \rangle)) \\
\langle \text{declarator} \rangle & ::= (\langle \text{identifier} \rangle : \langle \text{type} \rangle) [ ; \langle \text{identifier} \rangle : \langle \text{type} \rangle] \\
& \quad = \langle \text{expression} \rangle \\
\langle \text{message-call} \rangle & ::= \langle \text{path-name} \rangle(\langle \text{message-call-argument} \rangle) \\
\langle \text{message-call-argument} \rangle & ::= ?[ : \langle \text{type} \rangle] \mid \langle \text{expression} \rangle \\
\langle \text{qualifiers} \rangle & ::= [\langle \text{expression} \rangle]
\end{align*}
\]
\(\text{operation-name} ::= \text{identifier} | < | \leq | > | \geq | <> | = | \text{and} | \text{or} | \text{xor} | \text{not} | \text{implies} | + | - | * | /\)

\(\text{formal-parameter} ::= \text{identifier} : \text{type} \)

\(\text{type} ::= \text{path-name} | \text{collection-type} | \text{tuple-type} \)

\(\text{collection-type} ::= \text{identifier}((\text{type}))\)

\(\text{tuple-type} ::= \text{identifier}\{\{\text{variable-declaration}\}\}\)

\(\text{variable-declaration} ::= \text{variable-declaration-no-init}[ = \text{expression}]\)

\(\text{variable-declaration-no-init} ::= \text{identifier}[:\text{type}]\)
Appendix B

Semantics of OCL in PVS

We summarise the formal semantics of OCL constraints in PVS. We assume extensive knowledge of PVS. If a type of a function is not defined in this appendix, it is either defined in the prelude file of the PVS 3.3 release candidate or in the PVS library of NASA Langley. All type-consistency constraints generated by the theory described here have been proved.

Recall, that in this thesis we decided to work with a shallow embedding. Neither the syntax nor the semantics is formalised in PVS. OCL constraints are translated into PVS constraints which have the same semantics (see Chapter 4).

The mapping from types in OCL to types in PVS is shown in Table B.1. Considering this mapping, the OCL standard library is formalised by the following PVS theory. Observe, that the type OclInvalid is not represented in the PVS theory. An expression is of this type if and only if the associated type consistency constraints do not hold. Expressions, which cannot be typed in PVS, result in inconsistent theories.

OCL[Classes: TYPE+, <=: (partial_order?[Classes]), Values: TYPE+ Attributes: TYPE, References: TYPE, Locations: TYPE]: THEORY
BEGIN
As displayed in the preceding fragment of PVS, the theory is parameterised of the

<table>
<thead>
<tr>
<th>OCL Type</th>
<th>PVS Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>bool</td>
</tr>
<tr>
<td>Integer</td>
<td>int</td>
</tr>
<tr>
<td>Real</td>
<td>real</td>
</tr>
<tr>
<td>Collection(T)</td>
<td>N/A</td>
</tr>
<tr>
<td>Set(T)</td>
<td>finite_set[T]</td>
</tr>
<tr>
<td>Sequence(T)</td>
<td>finite_sequence[T]</td>
</tr>
<tr>
<td>Bag(T)</td>
<td>finite_bag(T)</td>
</tr>
<tr>
<td>OclAny</td>
<td>OclAny</td>
</tr>
<tr>
<td>OclVoid</td>
<td>OclVoid</td>
</tr>
</tbody>
</table>

Table B.1: Mapping OCL types to PVS types
Appendix B  Semantics of OCL in PVS

names of the classes occurring in the model, a partial order on these classes representing the inheritance relation, a type used to interpret attributes, the names of the attributes, the names of the association-end names (here called references), and the locations of the different state machines.

The operations not, and, or, implies, iff, and xor of the OCL type Boolean are identified with the respective functions in the PVS.

The operations +, −, *, /, <, >, <=, >=, abs, floor, ceil, min, and max of the OCL type Real are identified with the respective operations defined in the PVS prelude. The operation round is defined by expanding the definition in OCL (see Section 2.3.4), i.e.:

\[
\text{round}(x: \text{real}): \text{int} = \text{floor}(x + 1/2)
\]

The type Integer of OCL is identified with the type int in PVS. The operations +, −, *, /, <, >, <=, >=, abs, floor, ceil, min, and max are identified with the respective operations defined in the PVS prelude. The operation mod is identified with the function rem and the operation div is identified with ndiv in PVS.

The OCL types String and Unlimited Integer are not considered in our work. However, strings can be formalised using the PVS prelude. Formalising Unlimited Integer, which is a type with only one value, is in principle possible with the ordinal ω. We have doubts whether such a formalisation results in an adequate theory.

After having formalised the primitive types we define the type OclAny, i.e., the type of all objects. Observe, that we do not consider the elementary types to be subtypes of OclAny, because this would entail a rewrite of the prelude library. PVS does not allow to add new super-types to existing types. For convenience, we equate OclAny with nat.

\[
\text{OclAny: TYPE+ = nat CONTAINING } \emptyset
\]

null: OclAny = \emptyset

\[
\text{OclAnyNotNull} = \{\text{obj: OclAny | obj } \neq \text{null}\}
\]

Next, we define the state of an object and the state of the system.

\[
\text{ObjectState: TYPE = [#}
\text{ class: Classes,}
\text{ location: Locations,}
\text{ aval: [Attributes -> Values],}
\text{ rval: [References -> OclAny] #]}
\]

\[
\text{State: TYPE = [OclAnyNotNull -> ObjectState]}
\]

The operations = and <> are identified with = and /= in PVS. The operations oclIsInvalid() and oclIsUndefined are not represented in PVS. In PVS they would hold if a type consistency constraint is unprovable. The operation oclAsType is not represented.
in PVS, because the more powerful type system of PVS makes retyping unnecessary. Next, the operations `oclIsTypeOf`, `oclIsKindOf`, and `oclInState` are defined by:

\[
is_{\text{type of}}(self: \text{OclAny})(T: \text{Type})(state: \text{State}) = 
\text{state}(self)'.\text{type} = T
\]

\[
is_{\text{kind of}}(self: \text{OclAny})(T: \text{Type})(state: \text{State}) = 
\text{state}(self)'.\text{type} \leq T
\]

\[
is_{\text{type of kind of}}: \text{LEMMA}
\begin{align*}
&\text{OclAny}_{\text{isTypeOf}}(self)(T)(state) \implies \\
&\text{OclAny}_{\text{isKindOf}}(self)(T)(state)
\end{align*}
\]

\[
in_{\text{state}}(self: \text{OclAny})(l: \text{Location})(state: \text{State}) = 
\text{state}(self)'.\text{location} = l
\]

Next we define the type `OclVoid`.

\[
\text{OclVoid}: \text{TYPE FROM OclAny} = \{\text{obj} : \text{OclAny} \mid \text{false}\}
\]

Because `OclVoid` is defined as the empty type there are no operations defined for this type.

The type `Collection` in OCL is abstract and is not represented in PVS. If the type checker cannot determine what the concrete class of the collection is, each call to a property of the collection is translated into a case distinction in PVS. Generating the case distinction is necessary, because the type checker raises an error if it cannot resolve the type of the collection.

As displayed in Table B.1, the concrete collection types (we do not consider OrderedSet here) are identified with the corresponding finite collections in PVS. We show the definition of the conversion functions between the different collections, which are defined in separate theory (this is done to take advantage of parametric polymorphism in PVS).

```
Injections[T: TYPE]: THEORY
BEGIN

IMPORTING bags@top_bags

as_set(s: finite_set[T]): finite_set[T] = s

as_set(s: finite_sequence[T]): finite_set[T] = 
IF s'.length = 0 THEN emptyset 
ELSE {e: T \mid EXISTS (i: below[s'.length]): e = s'.seq(i)}
ENDIF
```
Appendix B  Semantics of OCL in PVS

as_set(s: finite_bag[T]): finite_set[T] = bag_to_set(s)

set_to_sequence(s: finite_set[T]):
  RECURSIVE finite_sequence[T] =
  IF empty?(s) THEN empty_seq
  ELSE (# length := 1,
       seq := LAMBDA (i: below[1]): choose(s) #)
     o set_to_sequence(rest(s))
  ENDIF
  MEASURE card(s)

as_sequence(s: finite_set[T]):
  {f: finite_sequence[T] |
   IF empty?(s) THEN f = empty_seq
   ELSE EXISTS (g: [below[card(s)] -> (s)]):
     f = (# length := card(s), seq := g #) ENDIF}

The nondeterminism involved in converting sets to sequences is expressed by defining as_sequence as an uninterpreted constant and axiomatising the properties of the function. A similar declaration is used for bags below. The constructive definition of as_sequence is given by set_to_sequence. This definition is mainly used to prove the existence of a constant as_sequence(s) for any s.

as_sequence(s: finite_sequence[T]): finite_sequence[T] = s

as_sequence(s: finite_bag[T]):
  {f: finite_sequence[T] |
   IF empty?(s) THEN f = empty_seq
   ELSE EXISTS (g: [below[card(s)] -> {e: T | member(e, s)}]):
     (FORALL (e: T):
      card({i: below[card(s)] | g(i) = e}) =
      count(e, s))
     AND f = (# length := card(s), seq := g #)
   ENDIF}

as_bag(s: finite_set[T]): finite_bag[T] =
  LAMBDA (e: T): IF member(e, s) THEN 1 ELSE 0 ENDIF

as_bag(s: finite_sequence[T]): finite_bag[T] =
Using these functions we define the meaning of flatten in PVS. There are nine variants, depending on the collection to be flattened and the type of the collection contained in it. We show only three of these functions as an example, because they all follow the same pattern.

**Flatten[T: TYPE]: THEORY**

BEGIN

IMPORTING Injections[T]

IMPORTING bags@top_bags

flatten(s: finite_set[finite_set[T]]):
RECURSIVE finite_set[T] =
   IF empty?(s) THEN emptyset
   ELSE union(choose(s), flatten(rest(s))) ENDIF
   MEASURE card(s)

flatten(s: finite_set[finite_sequence[T]]):
RECURSIVE finite_set[T] =
   IF empty?(s) THEN emptyset
   ELSE union(as_set(choose(s)), flatten(rest(s))) ENDIF
   MEASURE card(s)

flatten(s: finite_bag[finite_bag[T]]):
RECURSIVE finite_bag[T] =
   IF nonempty_bag?(s) THEN plus(choose(s), flatten(rest(s)))
   ELSE emptybag ENDIF
   MEASURE card(s)

END Flatten

Finally, we recall the definition of iterate-expressions. finite_sequence can be abbreviated as finseq.

**Iterate[T: TYPE, S: TYPE]: THEORY**

BEGIN

IMPORTING bags@top_bags

END
iterate(s: finite_set[T], a: S, f: [T, S -> S]): RECURSIVE S =
    IF empty?(s) THEN a
    ELSE iterate(rest(s), f(choose(s), a), f) ENDIF
    MEASURE card(s)

iter(s: finseq[T], a: S, f: [T, S -> S])(i: upto[s'length]):
    RECURSIVE S =
        IF i = s'length THEN a
        ELSE iter(s, f(s'seq(i), a), f)(i + 1) ENDIF
        MEASURE s'length - i

iterate(s: finseq[T], a: S, f: [T, S -> S]): S =
    iter(s, a, f)(0)

iterate(s: finite_bag[T], a: S, f: [T, S -> S]): RECURSIVE S =
    IF nonempty_bag?(s) THEN iterate(rest(s), f(choose(s), a), f)
    ELSE a
    ENDIF
    MEASURE card(s)
END Iterate

For each collection type we define its own function iterate. The function has the
collection s to iterate over as its argument, an initial value for the accumulator a, and a
function to apply for each iteration step.

Using the above definitions, we can describe the mapping of the operations defined
for collections in OCL to the ones in PVS. The operations are mapped according to
Table B.2.

The mapping of Sequence operations is given in Table B.3.

insert_at(s: finite_sequence[T], e: T, i: posnat):
    finite_sequence[T] =
        s^(0, i-1) o
            (# length := 1, seq := LAMBDA (i: below[1]): e #) o
            s^(i, s'length)

excluding(s: finite_sequence[T], e: T):
    RECURSIVE finite_sequence[T] =
        IF s'length = 0 THEN empty_seq
        ELSE IF s'seq(s'length - 1) = e
            THEN excluding(s ^ (0, s'length - 1), e)
            ELSE excluding(s ^ (0, s'length - 1), e) o
            ENDIF
        ENDIF
Table B.2: Representing Set operations in PVS

<table>
<thead>
<tr>
<th>OCL Expression</th>
<th>Translation to PVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s \rightarrow \text{size}()$</td>
<td>$\text{card}(s)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{count}(e)$</td>
<td>$\text{card}({x: (s) \mid x = s})$</td>
</tr>
<tr>
<td>$s \rightarrow \text{includes}(e)$</td>
<td>$\text{member}(e, s)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{excludes}(e)$</td>
<td>$\text{NOT member}(e, s)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{includesAll}(t)$</td>
<td>$\text{subset}(t, s)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{excludesAll}(t)$</td>
<td>$\text{disjoint?}(s, t)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{isEmpty}()$</td>
<td>$\text{empty?}(s)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{notEmpty}()$</td>
<td>$\text{NOT empty?}(s)$</td>
</tr>
<tr>
<td>$s = t$</td>
<td>$s = t$</td>
</tr>
<tr>
<td>$s &lt;&gt; t$</td>
<td>$s /= t$</td>
</tr>
<tr>
<td>$s \rightarrow \text{including}(e)$</td>
<td>$\text{union}(s, {e})$</td>
</tr>
<tr>
<td>$s \rightarrow \text{union}(t)$</td>
<td>$\text{union}(s, t)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{intersection}(t)$</td>
<td>$\text{intersection}(s, t)$</td>
</tr>
<tr>
<td>$s - t$</td>
<td>$\text{difference}(s, t)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{flatten}()$</td>
<td>$\text{flatten}(s)$ if $s$ is a set of collections, $s$ otherwise.</td>
</tr>
<tr>
<td>$s \rightarrow \text{asSet}()$</td>
<td>$s$</td>
</tr>
<tr>
<td>$s \rightarrow \text{asSequence}()$</td>
<td>$\text{as_sequence}(s)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{asBag}()$</td>
<td>$\text{as_bag}(s)$</td>
</tr>
</tbody>
</table>

The mapping of Bag operations is displayed in Table B.4. We use the definition of finite bags provided in the PVS library of NASA. Differences on bags in PVS is defined as:

\[
\text{difference}(b, c : \text{bag}[T]) : \text{bag}[T] = \\
(LAMBDA (t: T): \text{IF } b(t) > c(t) \text{ THEN } b(t) - c(t) \text{ ELSE } 0 \text{ ENDIF})
\]

This concludes the description of the semantics of OCL in PVS. Refer to Chapter 4 of how OCL expressions are embedded into PVS. Especially, Definition 4.2 defines the translation of OCL expressions into PVS expressions.

END OCL
### OCL Expression | Translation to PVS
---|---
$s \rightarrow \text{size}()$ | $s'\text{length}$
$s \rightarrow \text{count}(e)$ | $\text{card(\{i: below[s'length] | e = s(i)\})}$
$s \rightarrow \text{includes}(e)$ | $\text{EXISTS (i: below[s'length]): e = s(i)}$
$s \rightarrow \text{excludes}(e)$ | $\text{NOT EXISTS (i: below[s'length]): e = s(i)}$
$s \rightarrow \text{includesAll}(t)$ | $\text{subset?(as_set(t), as_set(s))}$
$s \rightarrow \text{excludesAll}(t)$ | $\text{disjoint?(as_set(s), as_set(t))}$
$s \rightarrow \text{isEmpty}()$ | $s'\text{length} = 0$
$s \rightarrow \text{notEmpty}()$ | $s'\text{length} /\neq 0$
$s \rightarrow \text{at}(i)$ | $s(i)$
$s = t$ | $s = t$
$s <> t$ | $s /= t$
$s \rightarrow \text{append}(e)$ | $s \circ (# \text{length} := 1, \text{seq} := \lambda x: e \ #)$
$s \rightarrow \text{prepend}(e)$ | $(# \text{length} := 1, \text{seq} := \lambda x: e \ #) \circ s$
$s \rightarrow \text{union}(t)$ | $s \circ t$
$s \rightarrow \text{excluding}(e)$ | $\text{excluding}(s, e)$
$s \rightarrow \text{flatten}()$ | $\text{flatten}(s)$ if $s$ is a sequence of collections, $s$ otherwise.
$s \rightarrow \text{subSequence}(l, u)$ | $s^{\land(l, u)}$
$s \rightarrow \text{asSet}()$ | $\text{as_set}(s)$
$s \rightarrow \text{asSequence}()$ | $s$
$s \rightarrow \text{asBag}()$ | $\text{as_bag}(s)$

**Table B.3:** Representing Sequence operations in PVS
<table>
<thead>
<tr>
<th>OCL Expression</th>
<th>Translation to PVS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s \rightarrow \text{size()}$</td>
<td>$\text{card}(s)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{count}(e)$</td>
<td>$\text{count}(s, e)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{includes}(e)$</td>
<td>$\text{member}(s, e)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{excludes}(e)$</td>
<td>$\text{NOT member}(s, e)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{includesAll}(t)$</td>
<td>$\text{FORALL } e: \ t(e) \geq s(e)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{excludesAll}(t)$</td>
<td>$\text{FORALL } e: \ t(e) &gt; 0 \text{ IMPLIES } s(e) = \emptyset$</td>
</tr>
<tr>
<td>$s \rightarrow \text{isEmpty}()$</td>
<td>$\text{empty?}(s)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{notEmpty}()$</td>
<td>$\text{NOT empty?}(s)$</td>
</tr>
<tr>
<td>$s = t$</td>
<td>$s = t$</td>
</tr>
<tr>
<td>$s &lt;!!!!!!!!!!!!= t$</td>
<td>$s /= t$</td>
</tr>
<tr>
<td>$s \rightarrow \text{union}(t)$</td>
<td>$\text{plus}(s, t)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{intersection}(t)$</td>
<td>$\text{intersection}(s, t)$</td>
</tr>
<tr>
<td>$s - t$</td>
<td>$\text{difference}(s,t)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{excluding}(e)$</td>
<td>$\text{purge}(s,e)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{flatten}()$</td>
<td>$\text{flatten}(s)$ if $s$ is a bag of collections, $s$ otherwise.</td>
</tr>
<tr>
<td>$s \rightarrow \text{asSet}()$</td>
<td>$\text{as_set}(s)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{asSequence}()$</td>
<td>$\text{as_sequence}(s)$</td>
</tr>
<tr>
<td>$s \rightarrow \text{asBag}()$</td>
<td>$s$</td>
</tr>
</tbody>
</table>

**Table B.4:** Representing Bags operations in PVS
Appendix B  Semantics of OCL in PVS