Verifying OCL specifications of UML models : tool support and compositionality

Kyas, M.

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:

`and`	`context`	`def`	`derive`	`else`
`endif`	`endpackage`	`false`	`if`	`implies`
`in`	`init`	`inv`	`let`	`null`
`not`	`or`	`package`	`post`	`pre`
`then`	`true`	`xor`		

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⟩ | _){⟨letter⟩ | _ | ⟨digit⟩}
⟨path-name⟩ ::= ⟨identifier⟩ | ⟨path-name⟩`::`⟨identifier⟩

⟨collection-literal⟩ ::= ⟨identifier⟩{[⟨collection-literal-part⟩]}
⟨collection-literal-part⟩ ::= ⟨expression⟩ | ⟨expression⟩..⟨expression⟩
⟨tuple-literal⟩ ::= Tuple{[*variable* − *decl*]}

A.2 Files

The first start symbol of the grammar is ⟨file⟩. It is used to parse a separate OCL file.

⟨file⟩ ::= ⟨package-declaration⟩⟨file⟩
| ⟨context-declaration⟩⟨file⟩ | ϵ
⟨package-declaration⟩ ::= package⟨path-name⟩{⟨constraint-declaration⟩}
endpackage
⟨constraint-declaration⟩ ::= {⟨context-declaration⟩⟨constraint⟩}
⟨context-declaration⟩ ::= context⟨classifier-context⟩
| context⟨operation-context⟩
⟨classifier-context⟩ ::= ⟨path-name⟩ | ⟨identifier⟩:⟨path-name⟩
⟨operation-context⟩ ::= ⟨path-name⟩([⟨formal-parameter⟩])]
| ⟨path-name⟩([⟨formal-parameter⟩]]):⟨type⟩
⟨constraint⟩ ::= ⟨stereo-type⟩[⟨identifier⟩]:[⟨expression⟩]
| def[⟨identifier⟩]:{⟨let-expression⟩}
⟨stereo-type⟩ ::= inv | post | pre

A.3 Expressions

The second start symbol of the grammar is ⟨expression⟩. It is used to parse constraint associated to model elements in XMI files.

⟨expression⟩ ::= ⟨logical-implies-expression⟩ | ⟨let-in-expression⟩
⟨let-expression⟩ ::= let⟨identifier⟩[([⟨formal-parameter⟩]])][:⟨type⟩]
= ⟨expression⟩
⟨logical-implies-expression⟩ ::= ⟨logical-xor-expression⟩
| ⟨logical-implies-expression⟩implies
⟨logical-xor-expression⟩
⟨logical-xor-expression⟩ ::= ⟨logical-or-expression⟩
| ⟨logical-xor-expression⟩xor⟨logical-or-expression⟩

⟨logical-or-expression⟩ ::= ⟨logical-and-expression⟩
| ⟨logical-or-expression⟩`or`⟨logical-and-expression⟩

⟨logical-and-expression⟩ ::= ⟨relational-expression⟩
| ⟨logical-and-expression⟩`and`⟨relational-expression⟩

⟨relational-expression⟩ ::= ⟨add-expression⟩
| ⟨add-expression⟩⟨relational-operator⟩⟨add-expression⟩

⟨relational-operator⟩ ::= `<` | `<=` | `>` | `>=` | `<>` | `=`

⟨add-expression⟩ ::= ⟨mul-expression⟩
| ⟨add-expression⟩ + ⟨mul-expression⟩
| ⟨add-expression⟩ − ⟨mul-expression⟩

⟨mul-expression⟩ ::= ⟨unary-expression⟩
| ⟨mul-expression⟩ ∗ ⟨unary-expression⟩
| ⟨mul-expression⟩/⟨unary-expression⟩

⟨unary-expression⟩ ::= ⟨primary-expression⟩ | −⟨unary-expression⟩
| `not` ⟨unary-expression⟩

⟨primary-expression⟩ ::= ⟨literal⟩ | `(`⟨expression⟩`)` | ⟨simple-property-call⟩
| ⟨postfix-expression⟩ | ⟨operation-call⟩

⟨simple-property-call⟩ ::= ⟨operation-name⟩[`@pre`][⟨qualifiers⟩]
[⟨poperty-call-params⟩]

⟨postfix-expression⟩ ::= ⟨primary-expression⟩`.`⟨property-call⟩
| ⟨primary-expression⟩`->`⟨property-call⟩
| ⟨primary-expression⟩`^`⟨message-call⟩
| ⟨primary-expression⟩`^^`⟨message-call⟩

⟨property-call⟩ ::= ⟨operation-name⟩[`@pre`][⟨qualifiers⟩]
[⟨poperty-call-params⟩]

⟨property-call-params⟩ ::= `(`[⟨declarator⟩]{⟨expression⟩}`)`

⟨declarator⟩ ::= {⟨identifier⟩[`:`⟨type⟩]}[`;`⟨identifier⟩[`:`⟨type⟩]
= ⟨expression⟩]|

⟨message-call⟩ ::= ⟨path-name⟩`(`{⟨message-call-argument⟩}`)`

⟨message-call-argument⟩ ::= `?`[`:`⟨type⟩] | ⟨expression⟩

⟨qualifiers⟩ ::= `[`{⟨expression⟩}`]`

⟨operation-name⟩ ::= ⟨identifier⟩ | `<` | `<=` | `>` | `>=` | `<>` | `=` | `and`
| `or` | `xor` | `not` | `implies` | `+` | `-` | `*` | `/`

⟨formal-parameter⟩ ::= ⟨identifier⟩`:`⟨type⟩

⟨type⟩ ::= ⟨path-name⟩ | ⟨collection-type⟩ | ⟨tuple-type⟩

⟨collection-type⟩ ::= ⟨identifier⟩`(`⟨type⟩`)`

⟨tuple-type⟩ ::= ⟨identifier⟩`{{`⟨variable-declaration⟩`}}`

⟨variable-declaration⟩ ::= ⟨variable-declaration-no-init⟩[= ⟨expression⟩]

⟨variable-declaration-no-init⟩ ::= ⟨identifier⟩[`:`⟨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

OCL Type	PVS Type
Boolean	bool
Integer	int
Real	real
Collection(T)	N/A
Set(T)	finite_set[T]
Sequence(T)	finite_sequence[T]
Bag(T)	finite_bag(T)
OclAny	OclAny
OclVoid	OclVoid

Table B.1: Mapping OCL types to PVS types

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.:

```
round(x: real): int = 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 he 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.

```
OclAny: TYPE+ = nat CONTAINING 0

null: OclAny = 0

OclAnyNotNull = {obj: OclAny | obj /= null}
```

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

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

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_type_of(self: OclAny)(T: Type)(state: State) =
    state(self)`type = T

  is_kind_of(self: OclAny)(T: Type)(state: State) =
    state(self)`type <= T

  is_type_of_kind_of: LEMMA
    OclAny_isTypeOf(self)(T)(state) IMPLIES
      OclAny_isKindOf(self)(T)(state)

  in_state(self: OclAny)(l: Location)(state: State) =
    state(self)`location = l
```

Next we define the type OclVoid.

```
  OclVoid: TYPE FROM OclAny = {obj : OclAny | 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 | EXISTS (i: below[s`length]): e = s`seq(i)}
      ENDIF
```

```
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] =
```

```
      LAMBDA (e: T): card({i: below[s`length] | s`seq(i) = e})

  as_bag(s: finite_bag[T]): finite_bag[T] = s
 END Injections
```

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
```

```
  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
```

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

Table B.2: Representing Set operations in PVS

```
            (# length := 1, seq := LAMBDA (x: below[1]): e #)
      ENDIF
  ENDIF
   MEASURE s`length
```

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:

```
difference(b,c : bag[T]): bag[T] =
  (LAMBDA (t: T): IF b(t) > c(t) THEN b(t) - c(t) ELSE 0 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
$s \rightarrow size()$	`s‘length`
$s \rightarrow count(e)$	`card({i:below[s‘length] | e = s(i)})`
$s \rightarrow includes(e)$	`EXISTS (i: below[s‘length]): e = s(i)`
$s \rightarrow excludes(e)$	`NOT EXISTS (i: below[s‘length]): e = s(i)`
$s \rightarrow includesAll(t)$	`subset?(as_set(t), as_set(s))`
$s \rightarrow excludesAll(t)$	`disjoint?(as_set(s), as_set(t))`
$s \rightarrow isEmpty()$	`s‘length = 0`
$s \rightarrow notEmpty()$	`s‘length /= 0`
$s \rightarrow at(i)$	`s(i)`
$s = t$	`s = t`
$s <> t$	`s /= t`
$s \rightarrow append(e)$	`s o (# length := 1, seq := lambda x: e #)`
$s \rightarrow prepend(e)$	`(# length := 1, seq := lambda x: e #) o s`
$s \rightarrow union(t)$	`s o t`
$s \rightarrow excluding(e)$	`excluding(s,e)`
$s \rightarrow flatten()$	`flatten(s)` if s is a sequence of collections, s otherwise.
$s \rightarrow subSequence(l, u)$	`s^(l, u)`
$s \rightarrow asSet()$	`as_set(s)`
$s \rightarrow asSequence()$	`s`
$s \rightarrow asBag()$	`as_bag(s)`

Table B.3: Representing Sequence operations in PVS

OCL Expression	Translation to PVS
$s \rightarrow size()$	`card(s)`
$s \rightarrow count(e)$	`count(s, e)`
$s \rightarrow includes(e)$	`member(s, e)`
$s \rightarrow excludes(e)$	`NOT member(s, e)`
$s \rightarrow includesAll(t)$	`FORALL e: t(e) >= s(e)`
$s \rightarrow excludesAll(t)$	`FORALL e: t(e) > 0 IMPLIES s(e) = 0`
$s \rightarrow isEmpty()$	`empty?(s)`
$s \rightarrow notEmpty()$	`NOT empty?(s)`
$s = t$	`s = t`
$s <> t$	`s /= t`
$s \rightarrow union(t)$	`plus(s, t)`
$s \rightarrow intersection(t)$	`intersection(s, t)`
$s - t$	`difference(s,t)`
$s \rightarrow excluding(e)$	`purge(s,e)`
$s \rightarrow flatten()$	`flatten(s)` if s is a bag of collections, s otherwise.
$s \rightarrow asSet()$	`as_set(s)`
$s \rightarrow asSequence()$	`as_sequence(s)`
$s \rightarrow asBag()$	`s`

Table B.4: Representing Bags operations in PVS