ASSL – A Snapshot Sequence Language

24. April 2001

1 Concrete Syntax

```
procedure ::= "procedure" name "(" variableDeclarationList ")"
              ("var" variableDeclarationList ";")?
              "begin"
                 instructionList
              "end" ";"

variableDeclarationList ::=
              (variableDeclaration (","variableDeclaration)*)?

instructionList ::= ( instruction ";" )*

instruction ::= variableAssignment
                | attributeAssignment
                | loop
                | atomicInstruction

variableAssignment ::= name ":=" extendedExpression

attributeAssignment ::= oclexpression DOT name
                        ":=" extendedExpression

loop ::= "for" name "in" oclexpression "begin"
            instructionList
         "end"

atomicInstruction ::=
   name "(" (instructionParameter(","instructionParameter)*)? ")"

instructionParameter ::= oclexpression | type

extendedExpression ::= oclexpression | atomicInstruction

oclexpression ::= "[" expression "]"
```

The start symbol is `procedure`. The non-terminal symbols `name`, `expression` and `variableDeclaration` are defined in [OMG99]. `type` is a (nested) type expression defined in [RG98].

2 Informal Semantics of Atomic Instructions

An atomic instruction is an instruction produced by the rule:

```
atomicInstruction ::=
name "(" (instructionParameter(","instructionParameter)*)? ")"
```

2.1 Changing the system state

Some atomic instructions are changing the system state:

`- Create( C: Class ): C`

Adds a new object of class C to the system state and returns the object. C is a class of the current model. This instruction is like the USE command `!create name: C`, but the name of the new object will be set automatically by the generator. Note that the instruction returns an object, which can be assigned to a variable or attribute.

`- CreateN( C: Class, n: Integer): Sequence(C)`

Adds n objects of class C to the system state and returns a sequence of the new objects in order of their creation. C is a class of the current model. n is a positive integer.

`- Insert( A: Association, e1:T1, e2:T2, ..., en:Tn)`

Inserts into the system state a new link connecting the objects returned by evaluating e1, e2, ... en. A is an association of the current model. Tm is the object type of the m-th association end of the association A. The changes to the system state are like `!insert(o1,o2,...,on) into A`. `o1,o2,...on` are object identifiers. In contrast, the parameters `e1,e2,...,en` are ocl expressions.

`- Delete( A: Association, e1:T1, e2:T2, ..., en:Tn)`

The same as `Delete( A: Association, e1:T1, e2:T2, ..., en:Tn)`, but the specified link will be removed from the system state. The USE command related to this instruction is `!delete(o1,o2,...,on) from A`. `o1,o2,...on` are object identifiers. In contrast, the parameters `e1,e2,...,en` are ocl expressions.

2.2 Random values

The generator can be started with a given integer to define a stream of pseudorandom numbers (the -r option). These pseudorandom numbers are used to evaluate instructions:

`- Any( seq: Sequence(T) ): T`

Returns a pseudorandom element from seq.
```
Post: seq->includes(result)
```

```
- Sub( seq: Sequence(T) ): Sequence(T)
```
Returns a pseudorandom subset of seq.
```
Post: seq->includesAll(result)
```

```
- Sub( seq: Sequence(T), n:Integer ): Sequence(T)
```
Returns a pseudorandom n-sized subset of seq.
```
Pre:  0<=n and n<=seq->size.
Post: result->size = n
Post: seq->includesAll(result)
```

2.3 Multiple transitions

The purpose of an instruction described as yet is to change a given configuration. A *configuration* is a tripel: The first element is the snapshot, which can be modified using the instructions listed in section 2.1. The second element is the current assignment of the variables. The third element is the current state of the pseudo random number stream (Section 2.2).

The instructions listed as yet are limited to their expressiveness: A configuration has only *one* transition to *one* following configuration. The purpose of ASSL is to define a snapshot space, that means a sequence of snapshots, which can be checked for validity by the generator.

The instructions listed below (try-instructions) will transfer a given configuration to one or more following configurations. That means a configuration can have more than one transition. If a procedure has more than one try-instruction, the procedure describes a configuration tree. The leaves of the tree are those configurations, whose snapshots will be checked by the generator.

```
- Try( seq: Sequence(T) ): T
```
Returns the elements of seq subsequently. If the value is assigned to a variable or an attribute of an object, this instruction produces `seq->size` configurations.
```
Post: seq->includes(result)
```

```
- Try(A:Association,s1:Sequence(T1),...,sn:Sequence(Tn)
```

```
Let links be A.link->select( l |
               s1->includes(l.linkEnd->at(1)) and
               s2->includes(l.linkEnd->at(2)) and
               ...
               sn->includes(l.linkEnd->at(n))
             )
```

At first every link in `links` is removed from the snapshot (cleaning up). Starting from this cleaned snapshot, $2^{links->size}$ configurations are produced: Each configuration has another combination of the links contained in `links`. That means every link combination will be created. The instruction does not effect links not included in `links`.

Literatur

[OMG99] OMG - OBJECT MANAGEMENT GROUP: *OMG - Unified Modeling Language Specification*, 1999. Version 1.3, http://www.rational.com/uml/resources/documentation.

[RG98] RICHTERS, MARK und MARTIN GOGOLLA: *On Formalizing the UML Object Constraint Language OCL.* In: LING, TOK WANG, SUDHA RAM und MONG LI LEE (Herausgeber): *Proc. 17th Int. Conf. Conceptual Modeling (ER'98)*, Band 1507 der Reihe *Lecture Notes in Computer Science*, Seiten 449--464. Springer, Berlin, 1998.