CIS 771: Software Specifications

Lecture 14:
Advanced OCL Expressions

Outline

- Coding transitive closure with recursion
- Useful expressions
- Undefined values
- Meta-modeling

...with the Academia model as the running example.

Transitive Closure in OCL

- OCL does not have a primitive operation for transitive closure
- OCL does allow recursion
- We must implement transitive closure directly in terms of recursion

Transitive Closure in OCL

Consider the following definitions (transitive-closure-1.use)

```
class A
end
```

```
association R between
  A role pred
  A role succ
end
```

We can attempt to code the transitive closure of R as follows

```
class A
operations
  closure() : Set(A) =
    succ.closure()->asSet()->including(self)
end
```

Transitive Closure in OCL

Consider the following instantiation (transitive-closure-instantiation-1.cmd)

```
!create a1:A
!create a2:A
!create a3:A
!insert (a1,a2) into R
!insert (a2,a3) into R
```

An example evaluation

```
use> ? a1.closure()
-> Set{@a1,@a2,@a3} : Set(A)
```


Transitive Closure in OCL

What is happening on a1.closure?

```
class A
operations
  closure() : Set(A) =
    succ.closure()->asSet()->including(self)
end
```

Tracing the evaluation through the recursion...

Level 1 call: self = a1, a1.succ = a2
Level 2 call: self = a2, a2.succ = a3
Level 3 call: self = a3, a3.succ = {}
Level 3 return: Set{@a3}
Level 2 return: Set{@a2,@a3}
Level 1 return: Set{@a1,@a2,@a3}

For You To Do…

- Pause the lecture…
- Load the model in *transitive-closure-1.use* into USE
- Run the script *transitive-closure-instantiation-1.cmd*
- Now give the following command at the USE command line

```
use> ? a1.closure()
```

  - what happens?
- Now give the following commands at the USE command line

```
use> !insert (a3,a1) into R
use> ? a1.closure()
```

  - what happens? why? can you fix the problem?

Transitive Closure in OCL

Consider the following instantiation (transitive-closure-instantiation-2.cmd)

```
!create a1:A
!create a2:A
!create a3:A
!insert (a1,a2) into R
!insert (a2,a3) into R
!insert (a3,a1) into R
```

An example evaluation

```
use> ? a1.closure()
…java.lang.RuntimeException: StackOverflow…
```

Assessment

- The problem is that we have an infinite path through *R* and the *closure* operation doesn't know how to stop.
- Intuitively, we should stop when we have collected all the elements that we encounter when walking across *R* starting from the initial value (e.g., *a1*).
- In other words, we should stop when we don't find anything "new" when walking across R.

If-then-else

- if *bool-expr* then *expr1* else *expr2* endif
  - Returns *expr1* if *bool-expr* is true
  - Returns expr2 if *bool-expr* if false
  - Undefined if *bool-expr* is undefined

...we can use the if-then-else construct to help us code an appropriate transitive closure operation

Transitive Closure in OCL

The correct coding of (reflexive) transitive closure

```
closure(s : Set(A)) : Set(A) =
    if s->includesAll(s.succ->asSet) then s
    else closure(s->union(s.succ->asSet))
    endif
```

Note: stop when we don't find anything new via R (succ) to add to s.

Note: the closure is reflexive because argument s must be included in the result

An initial call to compute reflexive transitive closure of {self}

```
reachableFromSelf() : Set(A) = closure(Set{self})
```


Transitive Closure in OCL

What is happening on *a1.reachableFromSelf()* ?

```
class A
operations
  closure(s : Set(A)) : Set(A) =
    if s->includesAll(s.succ->asSet) then s
    else closure(s->union(s.succ->asSet))
    endif
  reachableFromSelf() : Set(A) = closure(Set{self})
end
```

Tracing the evaluation through the recursion...

Level 1 call: s = {@a1}, s.succ = {@a2}
Level 2 call: s = {@a1,@a2}, s.succ = {@a2,@a3}
Level 3 call: s = {@a1,@a2,@a3}, s.succ = {@a1,@a2,@a3}
Level 3 return: Set{@a1,@a2,@a3}
Level 2 return: Set{@a1,@a2,@a3}
Level 1 return: Set{@a1,@a2,@a3}

For You To Do...

- Pause the lecture...
- Load the model in *transitive-closure-2.use* into USE
- Run the script *transitive-closure-instantiation-2.cmd* Note that this script adds (a3,a1) to R to create a cycle in R
- Now give the following command at the USE command line

```
use> ? a1.reachableFromSelf()
```

  - what happens? why?

Enumeration Types (per OCL spec)

General Form

enum {*value*$_1$, *value*$_2$, ..., *value*$_n$}

Example: Academia Grades

enum {A, B, C, D, F, X, W}

Enumeration Values

#A, #B, #C, #D, #F, #X, #W

Enumeration Types (per USE)

General Form – declare an *enum* type (e.g., at top of model)

enum *TypeName* {$value_1$, $value_2$, ..., $value_n$}

Example: Academia Grades

```
enum Grade {A, B, C, D, F, X, W}
…
class TranscriptEntry
attributes
   course : Course
   grade   : Grade
end
```

```
use> create e:TranscriptEntry
use> !set e.grade = #A
```


Ordered Associations

- Sometimes we want the result of navigating an association to be a sequence.
- Example:

```
association offspring between
  Person[0..2] role parents
  Person[*] role children ordered
end
```

- Then p.children is a sequence.

Operations on Sequences

- s->at(i) — *the ith element of s*
- s->first() — *the first element of s*
- s->last() — *the last element of s*
- s->append(a) — *adds a to end*
- s->prepend(a) — *adds a to front*
- s->asSet() — *converts to a set*

`let` Expressions

- let *x* : *Type* = *expr1* in *expr2*
  - evaluates *expr2* with each occurrence of *x* replaced by the value of *expr1*
  - avoids evaluating the same expression multiple times

Example

```
context Person inv:
  let income : Integer = self.job.salary->sum in
  if isUnemployed then
     income < 100
  else
     income >= 100
  endif
```


Helper Operations

```
...
let x : Type1 = expr1 in
  ...
  ...x...
  ...
  ...x...
...
```

```
...
f(expr1)
...

f(x : Type1) : Type2 =
  ...
  ...x...
  ...
  ...x...
```

For You To Do...

- Pause the lecture...
- Extend the model in *academia-7.use* as follows...
  - This model already contains an extension to *academia-5.use* that adds grades as an enumeration type to a *TranscriptEntry* class as done earlier in the lecture.
  - In the *Transcript* association, declare *transcriptEntries* to be ordered.
  - Using an enumeration type, add a *status* attribute to *Student* that can take on the values *#Normal* or *#Probation.*
  - Write an invariant that says that a student's status is normal iff they only have grades of A's and B's on their transcript. For this invariant, you may want to use a *let* expression since USE has no *iff* construct as a primitive. Specifically, you have to use implies twice and reverse the order of the arguments. Use a *let* to avoid duplicating large expressions.
  - Using transitive closure, add an invariant that states that there are no cycles in the prerequisite structure for courses.
  - Write a script to test your extensions.

Undefined Expressions

- Some expressions that can be undefined
  - object.oclAsType(T)
    - ...undefined when type of object has no subtype T
  - sequence->at(i)
    - ...undefined when i is greater than length of sequence
  - sequence->subSequence(i,j)
    - ...undefined when i,j lie outside the bounds of the sequence or when i > j
  - etc,

Undefined Expressions

- Undefined expressions tend to propagate
  - if bool-expr then expr-1 else expr-2
    - ...undefined if bool-expr is undefined
  - ...many other examples
- Exceptions:
  - true or anything = true
  - false and anything = false

For You To Do...

- Pause the lecture...
- Create some expressions whose values are undefined.
- Create some expressions where undefined values are propagated.
- Create some examples where *and* and *or* absorb the undefined values.

Collections are Flat (per OCL)

- In OCL,
  Set{Set{1, 2}, Set{2, 3}}
  and
  Set {1, 2, 3}
  have the same value.

- This happens implicitly and is beyond your control.

(see OMG-UML v1.3 Section 7.5.13 p.7.20)

Collections Are Usually Not Flat (per USE)

- In USE, Collection types can be nested to any level, e.g.,
  - Bag(Set(Sequence(Person))).

- Implicit flattening is only done when used with the shorthand notation for *collect*.

(see README.OCL in USE distribution)

Collections Are Usually Not Flat (per USE)

- You can always explicitly flatten a collection with the *flatten* operation that has been added in USE.
- For example,

```
company.branches->collect(c | c.employees)
```

results in Bag(Set(Employee)). This result value can be flattened into a Bag(Employee) by using the following expression:

```
company.branches->collect(c | c.employees)->flatten
```

(see README.OCL in USE distribution)

For You To Do…

- Pause the lecture…
- Try some examples of nested collections in USE (e.g., you can even use the *transitive-closure* models, and then define collections as literals)
- Flatten them with the *flatten* operation

(see OMG-UML v1.3 Section 7.8.1.1 po.7.28-7.29)

Meta Properties

- *type*.name : String
- *type*.attributes : Set(String)
- *type*.associationEnds : Set(String)
- *type*.operations : Set(String)
- *type*.supertypes : Set(OclType)
- *type*.allSupertypes : Set(OclType)
- *type*.allInstances : Set(type)

Note: it appears that only the last property is supported in USE.

Acknowledgements

- Material for this lecture is based on the following sources
  - Chapter 7 (the OCL chapter) of the OMG-UML specification (version 1.3 – March 2000)