A Translational Semantics for UML

Tony Clark
Desmond D'Souza

1.0 Overview

We propose a very small semantic domain for UML and show how all syntactic constructs can translate to this domain. The translations are layered thereby supporting the incremental definition of new modelling constructs whose meanings can be understood in terms of lower level modelling constructs.

The aim is to show how an approach that extends both syntax and semantics by adding new entities by specialization can co-exist with a translational approach. This is important for several reasons.

- Consistency and understandability: The larger language can be understood in terms of a small core. UML 2.0 might turn out to be a very large language, even without the scores of profiles that are waiting in the wings.
- Model interchange: a tool which natively supports more sophisticated language constructs can still exchange models with a tool which understands only the simpler translated constructs.
- Extensibility: New language constructs, whether "lightweight" or "heavyweight", can be defined as (a) brand new elements; (b) elements that translate into the base language, or (c) both, so a tool can choose to use the most appropriate one.

1.1 Table of contents

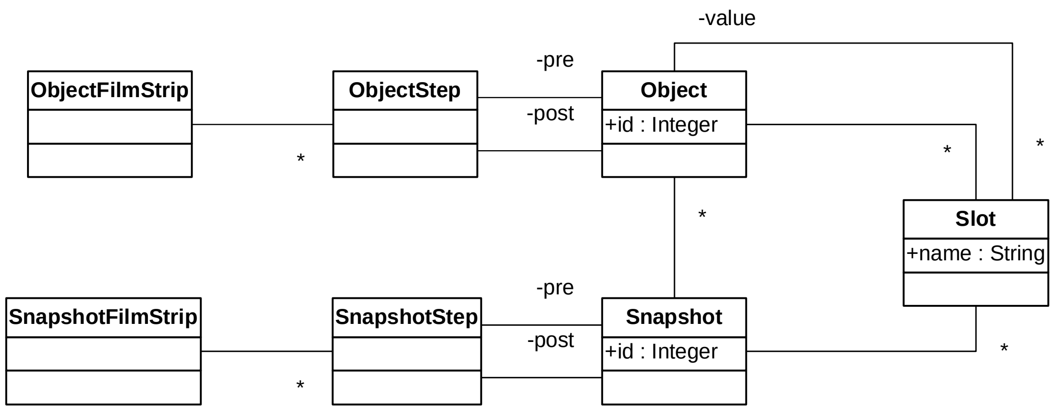
2.0 Semantic Domain

A UML model for:

- Object a container of slots with an id and a classifier (class).
- Slot a named object.
- ObjectStep between two objects with the same id.
- ObjectFilmStrip a container of object steps.
- Snapshot a container of objects with an id and a classifier (package).
- SnapshotStep between two snapshots with the same id.
- SnapshotFilmStrip a container of snapshot steps.

For each semantic domain element there is a set of the freely constructed model elements, for example: Objects and Snapshots.

The following is a model of the semantic domain. Some features have been simplified, for example if we have a model of *name spaces* then names (and possibly object identities) may be defined differently. Slots could have corresponding *SlotStep* and *SlotFilmStrip* classes. These issues will get tidied up when this model is generated from templates.



The above model does not show how objects are classified. All objects are linked to their classifier via an association. The end linked to the classifier is named ‘of’. In the proposed model, since everything is an object this association goes from Object to itself (or to a suitably specialized version of itself). Similarly, steps are linked to operations by a link named ‘of’.

The above model is incomplete. In particular it does not show the properties of filmstrips. This is because the list of properties has yet to be finalised however we are likely to have an initial starting state (possibly more than one?) for a filmstrip; given a state in a filmstrip we can ask for all possible subsequent states; and given a step we can ask for all subsequent steps.

Every object has a method ‘isKindOf’. The result o.IsKindOf(C) where o is an object and C is a classifier (also an object) is true when o.of = C or when C inherits from o.of. The method is not defined here (but should not cause any problem).

3.0 A Simple Language

Objects are represented by an object expression (like an instance diagram element):

object :C name = exp; name = exp; ...; name = exp **end**

Expressions may also be OCL. All expressions denote semantic domain elements. C is the classifier of the object (the value of the slot named ‘of’). C is an expression that denotes an object.

4.0 UML Features

This section shows how many static and dynamic UML features can be translated to expressions in the simple language. In all cases we assume a simple textual language for expressing models.

4.1 Classes

A class is an object that expresses structural and behavioural information. By default, a class object is an instance of the meta-class named Class. The *dynamic* instances of a default type of class are filmstrips that express steps between objects that are *static* instances of the class.

class C
end
	C =
	  object :Class
	    name = ‘C’;
	    staticInstances =
	      Objects->select(o |
	        o.isKindOf(C))
	  end

The class definition on the left denotes an object of type Class on the right with the given name. All static instances of C are objects whose classifier is C. The method ‘isKindOf’ permits the most specific classifier of o to be a sub-class of C.

4.1.1 Class Attributes

Attributes (aka *variables*) define structural features of classes. An attribute definition is

class C
  x:Integer
end
	C =
	  object :Class
	    name = ‘C’;
	    staticInstances =
	      Objects->select(o |
	        o.isKindOf(C) and
	        o.slots.names->includes(‘x’) and
	        o.x.isKindOf(Integer))
	  end

a constraint that the instances of the class should all have the slot whose value is of the appropriate type. In addition to the static instances, a class will have a slot that contains all its attributes. Since this is quite verbose and mechanical it is shown once below and then elided from the following examples.

```
class C
  x:Integer
end
```

```
C =
  object :Class
    name = ‘C’;
    attributes = Set{
      object :Attribute
        name = ‘x’;
        type = Integer;
        class = C
      end;
    staticInstances =
      Objects->select(o |
        o.isKindOf(C) and
        o.slots.names->includes(‘x’) and
        o.x.isKindOf(Integer))
  end
```

4.1.2 Static Invariants

A static class invariant defines a property that must hold for all *static instances* of the class. A static instance of a class is an object.

```
class C
  x : Integer
  static
    self.x > 10
end
```

```
C =
  object :Class
    name = ‘C’;
    staticInstances =
      Objects->select(o |
        o.isKindOf(C) and
        o.slots.names->includes(‘x’) and
        o.a.isKindOf(Integer) and
        o.x > 10)
  end
```

4.1.3 Operations

Class operations are named. Other information about the operation is optional. The following example shows a simple class operation named ‘m’. An operation places constraints on the dynamic instances of a class. Each dynamic instance is an object filmstrip whose steps represent state changes to instances of the class. Note that the static instances can then be derived from the dynamic instances. Like attributes, operations are represented as objects contained by the class. We do not restrict the temporal relationships between object steps.

```
class C
  m()
end
```

```
O =
  object :Operation
    name = ‘m’;
    args = Seq{};
    class = C
  end

 C =
  object :Class
     name = ‘C’;
     operations = Set{O};
     dynamicInstances =
       ObjectFilmStrips->select(f |
         f.steps->forAll(s |
           s.pre.isKindOf(C) and
            s.post.isKindOf(C) and
           s.pre.id = s.post.id and
           s.of = O));
     staticInstances =
       C.dynamicInstances.steps.pre->union(
         C.dynamicInstances.steps.post)
  end
```

Operations may be specified in terms of pre and post conditions. Such an operation has instances that are steps for which the pre state must satisfy the precondition and the

```
class C
  x:Integer;
  m(y:Integer)
    pre
      true
    post
      x = x@pre + y
end
```

```
O = ...
C =
  object :Class
     name = ‘C’;
    operations = Set{O};
     dynamicInstances =
       ObjectFilmStrips->select(f |
         f.steps->forAll(s |
           C.staticInstances->includes(s.pre)and
            C.staticInstances->includes(s.post) and
           s.pre.id = s.post.id and
             s.of = O implies
               s.argNamed(‘y’).isKindOf(Integer) implies
                 s.post.x = s.pre.x + s.argNamed(‘y’)));
     staticInstances =
       Objects->select(o |
         o.isKindOf(C) and
          o.slots.names->includes(‘x’) and
          o.x.isKindOf(Integer))
  end
```

post state must satisfy the postcondition. In general, given a precondition p and post-condition q, the truth of the precondition implies the truth of the postcondition:

```
class C
  m()
    pre p
    post q
end
```

```
O = ...
C =
  object :Class
    name = ‘C’;
    operations = Set{O};
    dynamicInstances =
      ObjectFilmStrips->select(f |
        f.steps->forAll(s |
          C.staticInstances->includes(s.pre)and
          C.staticInstances->includes(s.post) and
          s.of = O implies
            p[s.pre/self] implies
              q[s.post/self,s.pre/self@pre])
    staticInstances =
      Objects->select(o |
        o.isKindOf(C))
  end
```

4.1.4 Dynamic Invariants

A class dynamic invariant expresses a constraint on the dynamic behaviour of the class instances. An ObjectFilmStrip is an historical record of the behaviour of a particular object. In order for an object filmstrip to be a well formed dynamic instance of a class the filmstrip must satisfy the dynamic invariant of the class.

A class dynamic invariant is an OCL expression in which ‘self’ is an object filmstrip.

```
class C
  x:Integer;
  m(y:Integer)
  dynamic
    steps->forAll(s |
      s.name = ‘m’ implies
        s.post.x = s.pre.x +
          s.argNamed(‘y’))
end
```

```
O = ...
C =
  object :Class
    name = ‘C’;
    dynamicInstances =
      ObjectFilmStrips->select(f |
        f.steps->forAll(s |
          C.staticInstances->includes(s.pre)and
           C.staticInstances->includes(s.post) and
          s.pre.id = s.post.id and
          s.of = O
           s.post.x = s.pre.x + s.argNamed(‘y’)));
    staticInstances =
      Objects->select(o |
        o.isKindOf(C) and
         o.slots.names->includes(‘x’) and
          o.x.isKindOf(Integer))
  end
```

The invariant can express a constraint on any part of the behaviour represented by the filmstrip.

In general we may require a rich language of constraints on operations that allow us to express epochs of time such as ‘during the execution of this operation’ and ‘before this operation’ and ‘between these two operations’. Such language features will all translate onto the basic notion of dynamic invariant.

4.1.5 Class Constants

There are at least two variations on the theme of constant-hood: a slot value may be constant throughout all instances of a classifier; a slot value may not be changed throughout the life-time of a given instance (but the value may differ between instances). The former is similar to the **static** declaration in Java and the latter is similar to **final** in Java (or **const** in Pascal-like languages).

class C
  static x:Integer
end
	C =
	  let x = Integer.selectElement()
	  in object :Class
	        name = ‘C’;
	        staticInstances =
	          Objects->select(o |
	           o.isKindOf(C) and
	            o.slots.names->includes(‘x’) and
	            o.x.isKindOf(Integer) and
	            o.x = x)
	      end
	  end

The table above shows a static declaration in a class. The value of the slot named ‘x’ in all instances must be the same but may change over time. If any of the instances changes the value then it changes in all instances. Note that since we don’t know the particular value of the slot named ‘x’ then it is selected at random. We say nothing about the dynamic instances of the class C; therefore the value of ‘x’ may change but the static invariant must be satisfied.

A constant definition in a class must be expressed in terms of the dynamic instances:

class C
  const x:Integer
end
	C =
	  let x = Integer.selectElement()
	  in object :Class
	        name = ‘C’;
	        dynamicInstances =
	          ObjectFilmStrips->select(f |
	           f.steps->forAll(s |
	             s.pre.x = x and
	            s.post.x = x))
	     end
	  end

The slot named ‘x’ can be any value but the value must not change throughout the life-time of the object. Different objects may have different values for ‘x’.

4.2 Packages

A package is a container of classes and sub-packages. A package may be a specialization of another package (this issue is not addressed here). The static instances of a package are snapshots that must contain instances (1+?) of (only?) the classifiers contained in the package:

package P	P =
end	**object** :Package
	name = ‘P’;
	staticInstances =
	Snapshots->select(s |
	s.isKindOf(P))
	end

4.2.1 Contents

The contents of a package are represented directly as a slot in the package object. Each contained element must have a slot leading back to its container:

package P	C =
class C **end**	**object** :Class
end	name = ‘C’;
	package = P;
	staticInstances =
	Objects->select(o |
	o.isKindOf(C))
	end
	P =
	object :Package
	name = ‘P’;
	classes = Set{C};
	staticInstances =
	Snapshots->select(s |
	s.isKindOf(P) and
	s.slots.names->includes(‘C’) and
	s.contents->includesAll(s.C) and
	s.C->forAll(c |
	C.staticInstances->includes(c)))
	end

4.2.2 Package Attributes

Like classes, packages can have attribute definitions. This leads to appropriate slots in the static instances of the package as shown on the right:

package P x:Integer **end**		P = **object** :Package name = ‘P’; classes = Set{}; staticInstances = Snapshots->select(s | s.isKindOf(P) and s.slots.names->includes(‘x’) and s.x.isKindOf(Integer)) **end**

The attributes of a package may be defined **static** and **const.** These are treated in the same manner as the equivalent declarations in classes. The only difference between class attribute and package attributes is one of *scope*. In a class the attributes are limited to the objects which are instances of the class. In a package attributes are available to all the definitions in the package; therefore, instances of two otherwise non-related classes in a package can communicate through a package-level attribute.

4.2.3 Static Invariants

Like classes, packages can have static invariants. The invariant may apply to package attributes as shown on the left:

package P x:Integer **static** x > 10 **end**	P = **object** :Package name = ‘P’; classes = Set{}; staticInstances = Snapshots->select(s | s.isKindOf(P) and s.slots.names->includes(‘x’) and s.x.isKindOf(Integer) and s.x > 10) **end**

package P
  class C
    x:Integer
  end;
  class D
    y:Integer
  end
  static
    C->forAll(c |
      D->exists(d |
        c.x > d.y))
end
	The package definition on the left shows how package level static invariants provide more expressive power than class level static invariants by themselves. The invariant requires that in any legal snapshot of P there must exist an instance of D whose ‘y’-value is lower than all the ‘x’values of instances of C in the same snapshot.

4.2.4 Operations

A package operation is essentially the same as a class operation except where a class operation gives rise to steps betwen objects, a package operation gives rise to steps between snapshots.

package P
  m()
end
	O = ...
	P =
	  object :Package
	      name = ‘P’;
	      operations = Set{O};
	      dynamicInstances =
	        SnapshotFilmStrips->select(f |
	          f.steps->forAll(s |
	            s.pre.isKindOf(P) and
	            s.post.isKindOf(P) and
	            s.of = O));
	      staticInstances =
	        P.dynamicInstances.steps.pre->union(
	          P.dynamicInstances.steps.post)
	  end

Package operations may be specified using pre and post conditions. They are more expressive than class level operation specifications because they can refer to the entire snapshot rather than a single instance. The class on the left shows how creation and deletion methods can be specified.

```
package P
  class C end;
  createC():C
    pre true
    post
      (C - C@pre)->size = 1
      result = C - C@pre;
  killC(c:C)
    pre C->includes(c)
    post
     (C@pre - C) = Set{c}
end
```

```
CreateC = ...
KillC = ...
P =
  object :Package
     name = ‘P’;
     classes = Set{C};
     operations = Set{CreateC,KillC};
     dynamicInstances =
        SnapshotFilmStrips->select(f |
           f.steps->forAll(s |
             s.pre.isKindOf(P) and
             s.post.isKindOf(P) and
              s.name = CreateC implies
                (s.post.C - s.pre.C)->size = 1 and
                s.result = s.post.C - s.pre.C and
               s.of = KillC implies
            s.pre.C->includes(s.argNamed(‘c’))
               and  (s.pre.C - s.post.C) =
                          Set{s.argNamed(‘c’)});
     staticInstances =
       P.dynamicInstances.steps.pre->union(
          P.dynamicInstances.steps.post)
  end
```

The following is another example of a package level operation that involves concurrently swapping slot values:

```
package P
  class C
    x:Integer
  end;
  class D
    y:Integer
  end;
  m(c:C,d:D)
    pre
      C->includes(c) and
      D->includes(d)
    post
      C->exists(c’ | c’.id = c.id and
        D->exists(d’ | d’.id = d.id and
           c’.x = d.y and
           d’.y = c.x)))
end
```

4.2.5 Dynamic Invariants

package P **class** C x:Integer **end;** **class** D y:Integer **end;** **dynamic** start.C->forAll(c | start.D->forAll(d | c.x = d.y) and subsequently(start)-> forAll(s | s.C->forAll(c | not s.D->exists(d | c.x = d.y))) **end**	Package level dynamic constraints apply to snapshot filmstrips and can therefore range over all the steps from the beginning of a computation. The package on the left requires the values of all instances of C and D to agree on their ‘x’ and ‘y’ slots respectively. In all subsequent snapshots the values of these slots must be different. Therefore it requires something to have happened in the initial step to change their values.	C = ... D = ... P = **object :**Package name = ‘P’; dynamicInstances = SnapshotFilmStrips->select(f | f.steps->forAll(s | s.pre.isKindOf(P) and s.post.isKindOf(P)) and f.start.C->forAll(c | f.start.D->forAll(d | c.x = d.y)) and f.subsequently(f.start)->forAll(s | s.C->forAll(c | not s.D->exists(d | c.x = d.y))); staticInstances = P.dynamicInstances.steps.pre->union(P.dynamicInstances.steps.post) **end**

4.3 Objects

The semantic domain represents an abstract ideal. We can capture views of this ideal on object diagrams (and activity diagrams, collaboration diagrams and interaction diagrams etc) in UML. Snapshots are static instances of packages. Object diagrams (the concrete syntax for snapshots) are partial views of snapshots, and may be used as examples, counter examples, or, with the right quantifiers, as specifications.

Given an object diagram O which claims to be an instance of a package P we can construct the most specific package P’ such that O is a most specific instance of P’ then the claim that O:P is justified when P’ *statically conforms* to P. Package static conformance is essentially signature conformance and can be defined in terms of the static conformance of the package contents. This has yet to be defined.

4.4 Package and Class Equivalences

The translational approach defines a relationship between syntax constructs at different levels of abstraction. This induces relationships between syntax constructs at the same level of abstraction and allows us to show whether or not two different definitions are equivalent. The following are a few examples.

package P
  class C
    x:Integer
  end
end
	Defining a class attribute is equivalent to a pair of package operations. The snapshot filmstrip operator ‘between’ takes 2 args: a step s and a predicate and returns all the steps from s until the predicate is true. The operator ‘nextsetXFor’ takes an object and returns a predicate that returns true when supplied with a step labelled with a ‘setX’ operation whose first arg is the object.	**package** P
		  class C
		  end;
		  getX(c:C):Integer
		  setX(c:C,x:Integer)
		  dynamic
		    steps->forAll(s |
		        s.name = ‘setX’ implies
		          between(s,nextsetXFor(s.argNamed(‘c’)))->forAll(s’ |
		            s’.name = ‘getX’ implies s’.result = s.argNamed(‘x’)))
		  static
		    C->forall(c | c.x > 10)
		end

package P
  class C
    x:Integer
    static
      x > 10
  end
end
	Is equivalent to:	**package** P
		  class C
		    x:Integer
		  end
		  static
		    C->forall(c | c.x > 10)
		end

package P
  class C
    x:Integer;
    m()
      pre true
      post
        x = x@pre + 1
  end
end
	Is equivalent to:	**package** P
		  class C
		    x:Integer
		  end;
		  m(c:C)
		    pre C->includes(c)
		    post
		      C->exists(c’ |
		        c’.id = c.id and
		        c’.x = c.x + 1)
		end

4.5 State Machines

Translation or *desugaring* may be performed in stages. for example suppose we want a language of simple state machines. The syntactic extension on the left has a semantics in terms of a translation to the class definition on the right. The states become boolean variables and the transitions become dynamic invariants. The class definition on the right, in turn, has a semantics in terms of an object expression.

class C
  states s1, s2
  transitions
    t1: s1 -> s2
end
	class C
	  s1:Boolean;
	  s2:Boolean;
	  static
	    s1 xor s2
	  dynamic
	    steps->forAll(s |
	      s.pre.s1 and s.name = ‘t1’ implies
	        s.post.s2)
	end

Next add guards, next add actions.

4.6 Activity Diagrams

We could use activity diagrams as the basic way to compose behavior specs (expressions in the action language) and collab and sequence diagrams as special cases.

4.7 Collaboration Diagrams

4.8 Sequence Diagrams

4.9 Package Specialization

4.10 Templates