Chapter 9

A META-MODEL SEMANTICS FOR STRUCTURAL CONSTRAINTS IN UML

Stuart Kent
University of Kent, UK
s.j.h.kent@ukc.ac.uk

Stephen Gaito
Nortel Networks, UK
stga@nortelnetworks.com

Niall Ross
Nortel Networks, UK
nfr@nortelnetworks.com

Abstract The UML standard has adopted a meta-modelling approach to defining the abstract syntax of UML. A meta-modelling approach is taken essentially to aid the construction of automated tools, but the semantics is defined by statements in English. A meta-model that incorporates precise semantics would support the construction of tools that could perform semantically-oriented tasks, such as consistency checking. One approach to defining the formal semantics of a language is denotational: essentially elaborating (in mathematics) the value or instance denoted by an expression of the language in a particular context. However, instances must also be expressed in some language: in UML, instances of the static model are expressed using object diagrams. Thus a meta-model can be constructed which incorporates (a) the modelling language itself, (b) the modelling language of instances, and (c) the mapping of one into the other. The current UML meta-model provides some support for (a) and (b), but not (c). (c) is the part that carries the semantics. This paper presents one such meta-model, for a fragment of UML suitable for describing and constraining object structures. The fragment includes parts of class diagrams and invariants in the style of OCL. An indication is given as to how the approach could be extended to models characterising dynamic behaviour.

1. PRELIMINARIES

The team responsible for standardising the Unified Modelling Language (UML, [UML99]) has chosen a meta-modelling approach to make precise its abstract syntax. A meta-model for a language is a definition of that language written in terms of itself. There are good reasons for adopting a meta-modelling approach:

- UML has been designed in part to support OO software development. The UML meta-model provides a blueprint for the core of any CASE tool built using an OO programming language.
- If a tool can be built which generates code from UML, or, more specifically from a particular UML model, namely the meta-model, then the tool can be used to *bootstrap* itself, provided every time the meta-model is changed or extended the code generation is suitably updated.

  (We envisage the following scenario. The meta-model is extended in a predictable way, for example a new kind of model specific to a particular kind of domain, i.e. a framework, with restrictions on the nature and structure of classes it can contain. The Java code for the extension is generated largely automatically, perhaps with some manual fine-tuning. The .jar file produced is dropped into the appropriate directory, and immediately the tool gains a new menu item allowing you to create instances of the domain-specific framework.)
- If the meta-model is written in UML, there is a greater likelihood that those using the language will be able to grasp its subtleties more easily.

The UML meta-model itself has little or no semantic content, although it is accompanied by an English commentary that states informally the meanings of the various constructs. Whilst this may be adequate for a developer to use UML effectively (and many doubt even that), it is certainly not adequate if you wish to be sure that the language is unambiguous and well-formed, or if you wish to build automatic tools which are able to make use of semantic content. Tools we have in mind include, for example, a consistency checker that makes sure invariants defined on a model do not conflict, an instance generator which allows examples satisfying a model to be easily generated, etc.

In order to give a semantics to a modelling language (which may not be directly executable) there are, essentially, two approaches: an axiomatic approach, which states what sentences in the languages can be derived from other sentences; and a denotational approach, where expressions are mapped to the "instances" they denote. For example, a program may be thought of as denoting the set of execution traces that would be produced, if the program was executed from every conceivable starting state with every conceivable set of external stimuli during its execution. For a good starting point on the different approaches to semantics see [S86], pages 3-5.

In the context of UML, an axiomatic semantics would be a set of rules that would dictate, for example, which diagrams were derivable from other diagrams. A denotational approach would be realised by a) a definition of the form of an instance of every UML language element (e.g. the objects that could be denoted by a class, the links that could be denoted by associations, etc.), and b) a set of rules which determine which instances are and are not denoted by a particular language element.

This paper adopts a denotational, meta-modelling approach to the semantics of a part of the UML. There are three main steps to the approach:

- Define the meta-model for the language of models: classes, roles, models.
- Define the meta-model for the language of instances: objects, links, snapshots.
- Define the mapping (also within the meta-model) between these two languages.

The part of the UML considered here is a subset for describing object structure. For characterising *models*, it includes the essentials of class diagrams, and a significant fragment of the *object constraint language* (OCL, [WK98]), a precise language based on first-order predicate logic, used for expressing constraints on object structure which can not be expressed by class diagrams alone. For characterising *instances of models*, it includes the language of object diagrams.

The paper does not consider constructs for describing dynamic behaviour, such as sequence diagrams and state diagrams (though state diagrams without transitions are considered).

We have chosen to start with a clean slate, rather than the existing meta-model of UML. The reasons are twofold:

- To ensure that only concepts essential to the semantics are introduced, thereby avoiding unnecessary distractions. A future task will be to incorporate the ideas set out here into the UML standard with as little disruption as possible.
- Because the existing meta-model is silent about the OCL. A major component of the meta-model presented here is a representation of the concepts underpinning OCL.

Thus the meta-model developed here elaborates the conceptual core of UML/OCL and is not tied to any particular concrete syntax. For example, the meta-model for the OCL fragment supports the visual syntax for constraints first described in [K97] and further developed in [KG98] and [GHK99].

Section 2 describes the meta-model for simple models, involving only classes, roles and inheritance, where an association is represented by 2 roles. Section 3 describes the meta-model for instances of models, which are essentially object diagrams. Section 4 makes the connection between models and their instances, focussing, for the time being, on just the roles and classes. This introduces the basic form of the semantic approach described here. Section 5 then mixes in the meta-model for invariants, providing, essentially, the abstract syntax for OCL. Section 6 proceeds to extend the connection between model and instance with rules that take account of invariants. Section 7 concludes with a discussion of how to extend this meta-modelling approach into other parts of UML: work is already underway on handling model extension, composition and refinement as introduced by e.g. [SW99]; and dynamic behaviour (state diagrams, sequence diagrams, pre/post conditions).

2. MODELS, CLASSES, STATES AND ROLES

Figure 1 is a UML class diagram defining a model to be a set of roles and classes. A role is one end of an association, and has a source and target class. A role has a name, and upper and lower bounds on its cardinality (the number of links that any

object of the source class may have to objects of the target class). A role may have an inverse. If it has, then the role and its inverse constitute an association. Thus in our model, associations are mapped to two roles, where each is the inverse of the other.

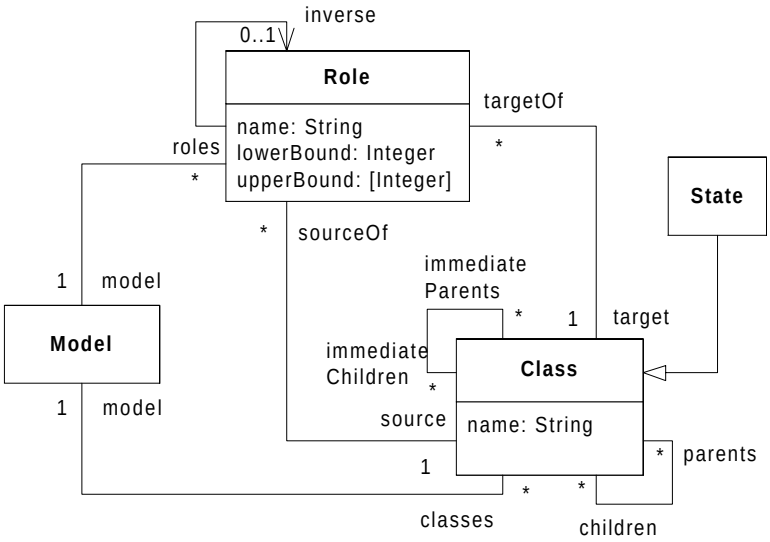

Figure 1. Models

It would be possible to introduce a class `Association` of objects constituted of two roles, each the inverse of the other, and have two kinds of role, one in association, and one without inverse. Roles are also restricted, here, to have a single source and target. Roles with multiple sources/targets represent relations between tuples – tuples of the sources are related to tuples of the targets, and could be used ot encode n-ary associations. However, the purpose of this paper is to show all the pieces of a denotational semantics embedded in a meta-model, focussing, in particular, on constraints expressed in OCL; roles with inverses are quite sufficient for this purpose. We envisage no fundamental difficulties in extending the semantics to n-ary associations; the constraints on the meta-model will just be a little more sophisticated.

Classes may have children and parents, where the parents of a class A are those classes from which A inherits. A simplified view of inheritance is taken in this paper, which admits multiple and single inheritance, but does not admit renaming of, or strengthening of cardinalities on, roles. (This can be handled, but requires a more sophisticated model of inheritance which there is not room to explore here.) Constraints on a class are carried down to children of that class.By the inverse nature of an association, if a class A is the parent of another class B, then A will have B as one of its children, and vice-versa. We distinguish the immediate children/parents of a class; the children/parents of a class are the immediate children/parents union their children/parents.

A state is a dynamic class: an object of a state may move into a different state, so it is not the case that an object created in a state (for a dynamic class) remains in that state (remains a member of that class) for its whole lifetime. The equation of state with dynamic class dates back to [S92], and is now embodied in the Catalysis method [SW99].

Roles and classes are here restricted to belong to a single model. This would not be the case in a meta-model that supported the notion of model extension, i.e. a model A being constructed by including everything in model B then adding some. However, the notion of model extension brings with it additional complications that there is not the space to consider in this paper.

As with the standard UML-meta model, we can make precise the well-formedness rules that apply to valid instances of the UML-meta model. That is, these rules further constrain the abstract syntax of UML to ensure that only valid expressions of the UML are permitted. The rules for this fragment of the meta-model are given below. They are written in OCL, the precise constraint language of UML.

```
context m:Model inv:
     the source and target classes of m's roles are classes in m
     m.classes->includesAll(m.roles.source
          ->union(m.roles.target))
context c:Class inv:
     the children/parents of c are the immediate children/parents union
     the children/parents of those
     c.children=c.immediateChildren
          ->union(c.immediateChildren.children)
     and c.parents=c.immediateParents
          ->union(c.immediateParents.parents)
context c:Class inv:
     the parents of c are in the same model as c
     c.parents.model=c.model
context c:Class inv:
     c is not a parent or child of itself
     not(c.parents->includes(c) or c.children->includes(c))
context r:Role inv:
     r.inverse->isEmpty or
     the inverse of the inverse of r is r
     (r.inverse.inverse=r and
     the inverse of r reverses the source and target of r
     r.inverse.source=r.target and r.inverse.target=r.source
     the inverse of r is in the same model as r
     and r.model=r.inverse.model)
```

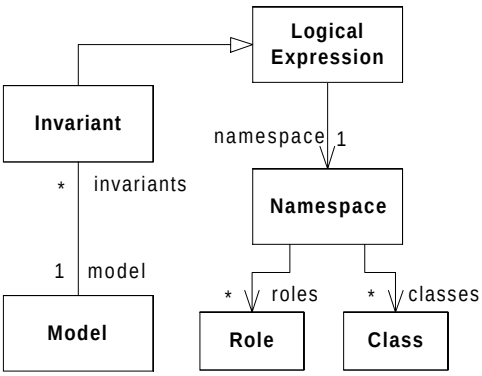

Figure 2. Invariants

Figure 2 shows the relationships between models and invariants. An invariant is a logical expression, which always comes with its own namespace: the classes and roles referred to in the logical expression.

A constraint is required to ensure that the language defined in the namespace of invariants of a model are provided by that model:

```
context m:Model inv:
    m.classes->includesAll(m.invariants.namespace.classes)
    and m.roles->includesAll(m.invariants.namespace.roles)
```

The full structure of logical expressions will be revealed in Section 5. However, there is already enough in this part of the meta-model to consider the semantics of a UML model.

3. INSTANCES

The semantics of a UML model is given by constraining the relationship between a model and possible instances of that model. That is, constraining the relationship between expressions of the UML abstract syntax for models and expressions of the UML abstract syntax for instances. The latter requires a meta-model representing the abstract syntax of instances.

Figure 3 defines a *snapshot* (Catalysis parlance; represented in UML as an *object diagram*) to be a set of objects and a set of links. Snapshots, links and objects are instances of models, roles and classes, respectively. The UML standard meta-model does not make the connection between models and snapshots, but the symmetry between the two halves of the diagram is too hard to resist.

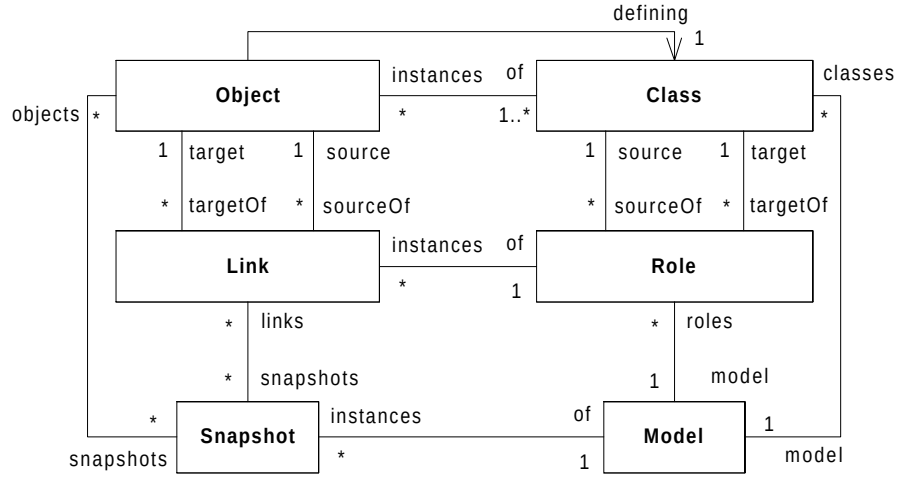

Figure 3. Objects, links, snapshots

A snapshot can only be an instance of a single model, because, here, models are self-contained and disjoint from other models (excepting the special case where a model is behaving as a namespace for an invariant). Snapshots could be instances of more than one model in a context where models could be extensions of other models, and thus have components in common.

On the other hand, objects are instances of one or more classes, hence they may be of one or more types. With no further constraints, it is possible for an object to change the classes of which it is an instance; thus this meta-model supports dynamic types. Indeed, as mentioned earlier, it is our view that states (as in state diagrams) are just dynamic classes.

There is one well-formedness rule for instances (i.e. the left-hand side of the diagram), which is given below:

```
context s:Snapshot inv:
    the source and target objects of s's links are objects in s
    s.objects->includesAll(s.links.source
        ->union(s.links.target))
    links between two objects are unique per role
    & s.links->forAll(l | s.links->select(l'|
    l'.source=l.source & l'.target=l.target & l'.of=l.of)=l)
```

Other invariants are required constraining the relationships between models and instances. These constitute the semantics and are the subject of the next section.

4. SEMANTICS: THE BASICS

Our semantics for UML focuses on the relationship between a model and its possible instances. The constraints on this relationship are relatively simple, ignoring invariants for the time being, but they do demonstrate the general principle.

Firstly, there are two constraints relating to objects and links, respectively. The first shows how inheritance relationships can force an object to be of many classes. The second ensures that a link connects objects of classes as dictated by its role.

context o:Object inv:

the classes of o must be a single class and all the parents of that class

```
o.of->exists(c | o.of=c->union(c.parents))
```

context l:Link inv:

objects which are the source/target of links are of classes which are at the source/target of the corresponding roles

```
(l.of.source)->intersection(l.source.of)->notEmpty
and (l.of.target->intersection(l.target.of)->notEmpty
```

Secondly, there are four constraints which ensure that a snapshot is a valid snapshot of the model it is claimed to be a snapshot of. Of these four constraints, the first and second ensure that objects and links are associated with classes and roles known in the model. The third constraint ensures that within the snapshot cardinality constraints on roles are observed. The fourth that reverse links are in place for roles with inverses.

context s:Snapshot inv:

the model, that s is a snapshot of, includes all the classes that s.objects are instances of

```
s.of.classes->includesAll(s.objects.of)
```

context s:Snapshot inv:

the model. that s is a snapshot of, includes all the roles that s.links are instances of

```
s.of.roles->includesAll(s.roles.of)
```

context s:Snapshot inv:

the links of s respect cardinality constraints for their corresponding role

```
s.links.of->forAll(r | let links_in_s be
     r.instances->intersect(s.links) in
          (r.upperBound->notEmpty implies
               links_in_s->size <= r.upperBound)
          and links_in_s->size >= r.lowerBound)
```

context s:Snapshot inv:

if a link is of a role with an inverse, then there is a corresponding reverse link

```
s.links->forAll(l | l.of.role.inverse->notEmpty implies
     s.links->select(l'| l'.source=l.target &
     l'.target=l.source & l'.of=l.of.inverse)->size=1
```

5. INVARIANTS

This section provides the abstract syntax for a fragment of the OCL used to write invariants, many examples of which have appeared already as constraints on the meta-model. The OCL fragment under scrutiny includes navigation expressions, logical connectives and set comparisons, but does not include number denoting expressions, general quantification or filtered sets. These latter three concepts bring in other layers of complexity into the semantics, which there is not space to consider here.

Invariants are logical expressions, which appear in the context of models (see also *Figure 2*). This paper only considers invariants on a single class, as embodied in OCL. We realise that this can be too restrictive and that, often, model-wide invariants are required (indeed there is one such invariant in Section 6). It should not be difficult to extend our approach to cover this situation.

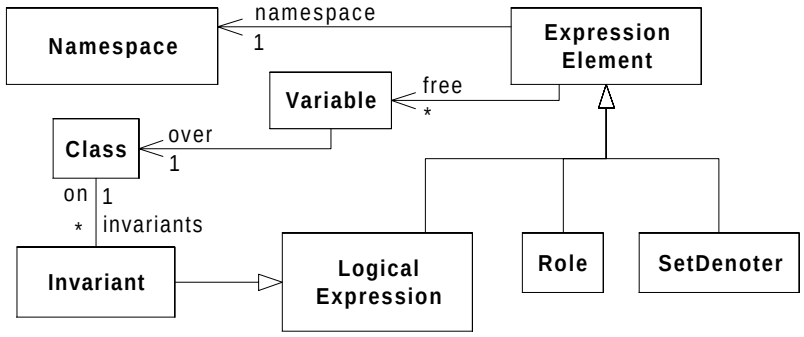

Figure 4. Expressions

An invariant has a single free variable which ranges over its class:

```
context i:Invariant inv:
    i.free.over=i.on and i.free->size=1
```

As will be shown in Section 6, this variable is universally quantified. Thus our treatment of this restricted form of invariant does show the way for a more general treatment of quantification.

Logical expressions are one kind of expression, the others being roles and set denoters. All expressions are associated with a namespace and one or more free variables (though sub-expressions of invariants considered here will have at most one free variable). The remainder of this section explores the different kinds of set denoter and logical expression.

5.1. Logical connectives

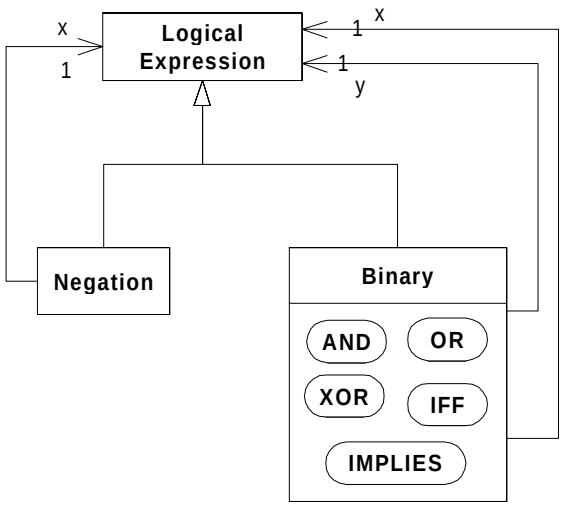

Figure 5. Logical connectives

The logical connectives are introduced via two kinds of logical expression (see *Figure 5*): a negation and a binary expression. The binary expression has state indicating the kind of connective used. (States are shown here using the state diagram shape – a box with rounded corners. They can be read as dynamic subclasses of the class in which tney are contained.)

There are constraints to propagate namespaces and free variables up through the structure of a complex expression.

```
context n:Negation inv:
    n.namespace=n.x.namespace & n.free=n.x.free

context b:Binary inv:
    b.namespace.classes=b.x.classes->union(b.y.classes)
    & b.namespace.roles=b.x.roles->union(b.y.roles)
    & b.free=b.x.free->union(b.y.free)
```

5.2. Set comparison

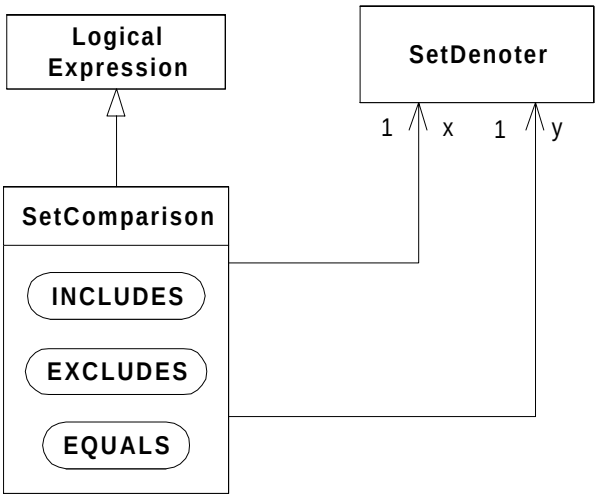

Figure 6. Set comparison

Logical expressions are also formed by comparing sets, here by containment, exclusion or equality. A set S excludes T if S has no elements in common with T; clearly if S excludes T, then T excludes S. A more usual term for *includes* is *contains*; a more usual term for excludes is *disjoint*. We have used includes and excludes as they are the terms used in OCL.

Namespaces and free variables are propagated in the same way as for binary logical expressions.

5.3. Set denoters

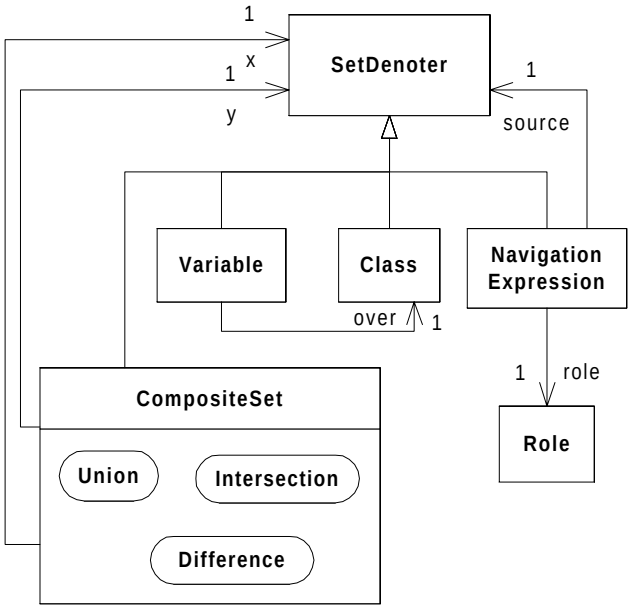

Figure 7. Set denoters

Finally, there are different kinds of set denoters, as indicated by *Figure 7*. Set denoters denote sets of objects. All are obvious except, perhaps, navigation expressions. In OCL, a navigation expression takes the concrete form `source.role`, where source denotes a set and role is a role. Of course source may itself be a navigation expression, allowing arbitrary length navigation expressions to be recursively defined.

Namespaces and free variables are propagated as follows:

```
context c:Class inv:
    c.namespace.classes=c & c.namespace.roles->isEmpty
    & c.free->isEmpty

context r:Role inv:
    r.namespace.roles=r & r.namespace.classes->isEmpty
    & r.free->isEmpty
```

Composite sets are treated like binary logical expressions. Finally:

```
context v:Variable inv:
    v.namespace.classes->isEmpty
    & v.namespace.roles->isEmpty
    & v.free=v
```

```
context ne:NavigationExpression inv:
     ne.namespace.roles=ne.source.namespace.roles->union(role)
     & ne.namespace.classes=ne.source.namespace.classes
     & ne.free=ne.source.free
```

6. SEMANTICS OF INVARIANTS

The semantics of invariants requires constraints that ensure that a snapshot of a model satisfies the invariants of the model, which in turn requires each invariant to hold for all objects in the snapshot of the class which the invariant is on. This requires the relationship between instances and models to be enriched. The two main problems when checking against an invariant are:

- To derive all the sets required by the invariant for the particular snapshot concerned, for example the sets represented by particular navigation expressions, classes etc.
- To work out a system for calculating the denotation of an expression for a particular substitution of the variable ranging over objects of the class the invariant is on.

These problems are solved in *Figure 8* by a qualified role from `Expression` to `Denotation`. Provide an expression with a snapshot and an object, and it will return the denotation of that expression in that snapshot for that object. Rules for each kind of expression determine the exact nature of the denotation returned.

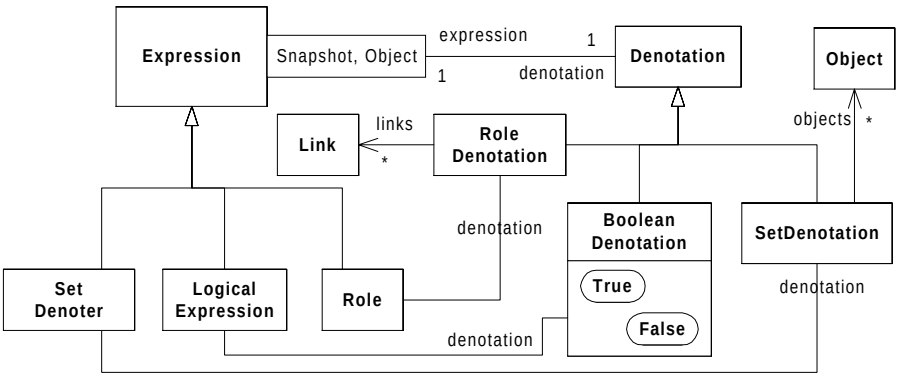

Figure 8. Denotations

With this in place, it is now possible to state the main constraint that ensures a snapshot satisfies the invariants of the model of which it is claimed to be an instance.

```
context s:Snapshot, o':Object inv:
     s.model.invariants->forAll(i |
     i.on.denotation(s,o').objects->forAll(o |
     i.denotation(s,o).oclIsInState(BooleanDenotation::True)))
```

(Strictly, OCL only allows an invariant to be applied to a single class. However, using quantifiers it is possible to write an invariant with variables that range over many classes. The syntax used here is far less cumbersome that the latter approach, and, we feel, fits naturally within OCL.)

That is, for every invariant of the model that `s` is an instance of, and every object `o`, which in `s` and of the class that the invariant is on, the denotation of the invariant in `s` for object `o` must be true. Note that `o'` is just a placeholder: when obtaining the denotation of a class in a snapshot, the object argument is ignored.

All that now remains is to give the rules stipulating the denotation returned for each kind of expression, for a particular snapshot and object. These rules work compositionally: they state how the denotation of a complex expression is built up from the denotation of its parts.

We consider each kind of expression in the same order as they were introduced in Section 5.

SECTION 5.1, LOGICAL CONNECTIVES, *FIGURE 5*

```
context b:Binary::AND, s:Snapshot, o:Object inv:
b.denotation(s,o).oclIsInState(BooleanDenotation::True) =
     b.x.denotation(s,o).oclIsInState(BooleanDenotation::True)
 &   b.y.denotation(s,o).oclIsInState(BooleanDenotation::True)
```

Similarly for OR, XOR, IFF, IMPLIES & NOT.

SECTION 5.2, SET COMPARISON, *FIGURE 6*

```
context sc:SetComparison::INCLUDES, s:Snapshot, o:Object inv:
     sc.denotation(s,o).oclIsInState(BooleanDenotation::True)=
          sc.x.denotation(s,o).objects
               ->includesAll(sc.y.denotation(s,o).objects)
```

Similarly for EXCLUDES and EQUALS.

SECTION 5.3, SET DENOTERS, *FIGURE 7*

```
context cs:CompositeSet::UNION, s:Snapshot, o:Object inv:
     cs.denotation(s,o).objects=
          cs.x.denotation(s,o).objects
               ->union(cs.y.denotation(s,o).objects)
```

Similarly for INTERSECTION and DIFFERENCE.

The denotation of a variable is the object passed to it as argument, as that object represents its substitution for this denotation:

```
context v:Variable, s:Snapshot, o:Object inv:
    v.denotation(s,o).objects=o
```

Although not very general (any variable will be substituted for the object that is passed as an argument), this approach is sufficient for generating the denotation of the restricted form of invariant considered here, which only has a single variable. A more general treatment of variables and quantification will be given in a future paper.

The denotation of a class in a snapshot is all the objects of that class in the snapshot:

```
context c:Class, s:Snapshot, o:Object inv:
    c.denotation(s,o).objects =
        s.objects->select(o'| o'.of->includes(c))
```

Similarly for roles.

Finally, the denotation of a navigation expression in a snapshot collects together the target objects of all the links of its role in the snapshot that are sourced on an object of the denotation of its source:

```
context ne:NavigationExpression, s:Snapshot, o:Object inv:
    ne.denotation(s,o).objects = ( ne.role.denotation.links
        ->select(l | ne.source.denotation(s,o).objects
            ->includes(l.source)) ).target
```

7. FURTHER WORK

The meta-model for a significant part of the UML/OCL for models has been provided, and its semantics expressed within the meta-model by constraining the relationship to model-instances, also captured within the meta-model. Some shortcuts were taken, and some generalisations need to be made:

- Quantification in OCL has not been modelled, although the treatment of invariants should easily be extended to accomodate quantifiers.
- Set filtering (select & reject in OCL) should follow on from a treatment of quantifiers, as it, too, requires a general treatment of variables.
- Numerical expressions remain to be dealt with. This will require a tie up between pre-defined expressions (e.g. set size) and user-defined expressions, e.g. roles targeted on the class Number.
- A treatment has only be given for simple roles, i.e. ones with a single source and target. Qualified roles, as used, for example, in *Figure 8*, require roles with

multiple sources. Roles with multiple sources/targets, will be required to capture n-ary associations.

The above issues should provide a rich basis for producing a single static model. We have also been working on the meta-model (including semantics) for:

- Model extension, including class/role renaming, strengthening of role cardinalities and strengthening of invariants. A model extension mechanism would allow a model to be constructed from other models, which may include models representing patterns. This work may be viewed as a formalisation of the template mechanism described in Catalysis [SW99].
- Model refinement, focusing on the mapping of object structures at one level of abstraction to object structures at a more concrete level. This is in fact the most important aspect for us. We are interested in realising services (e.g. a guaranteed throughput) over concrete networks (e.g. IP networks). Formalisation of the modelling language and refinement and their semantics is essential to support the development of tools that will, for example, walk the refinement relationships to configure a network to support certain services.

It is also intended to extend the meta-model to incorporate *dynamic* modelling. This will require formalisation of operations (actions), state diagrams, sequence diagrams, and, of course, the corresponding instances of these constructs. We have a good idea how to handle some of this. Actions have arguments and pre/post conditions, for which the logical expression infrastructure already exists. States have already been introduced, as a kind of class, and transitions in state diagrams are effectively constraints on how an object configuration can change when certain actions are invoked with particular participants, that is a particular kind of post condition. Sequence diagrams impose constraints on how actions may be sequenced, and these constraints may depend on which objects participate in the actions. On the instance side, a notion of *trace* will be required, where a trace is a sequence of snapshots (in Catalysis [SW99], a *filmstrip*), with action instances between each pair. Traces must satisfy the constraints derived from the pre/post conditions, the sequence and state diagrams.

We have started work under the auspices of the pUML (precise UML) group (http://www.cs.york.ac.uk/puml) to incorporate many of the semantic ideas into the UML standard meta-model.

Finally, tools incorporating parts of the meta-model are being developed at Nortel to support network modelling and management of network services. We hope, and expect, that these tools will have a wider application in the general area of conceptual modelling and will serve to improve the lot of the practising modeller.

REFERENCES

[S92] D. D'Souza, Education and Training: Teacher! Teacher!, *Journal of Object-Oriented Programming*, vol. 5, pp. 12-17, 1992.

[SW99] D. D'Souza and A.C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison Wesley, 1999.

[GHK99] Y. Gil, J. Howse, and S. Kent, Formalizing Spider Diagrams, submitted for publication, 1999.

[K97] S. Kent. Constraint Diagrams: Visualising Invariants in Object Oriented Models. In: *Proceedings of OOPSLA97*, ACM Press, 1997.

[KG98] S. Kent and Y. Gil, Visualising Action Contracts in OO Modelling, *IEE Proceedings: Software*, vol. 145, 1998.

[S86] D.A. Schmidt, *Denotational Semantics: A Methodology for Language Development*, Allyn and Bacon, Massachusetts, 1986.

[UML99] UML task force. *UML 1.1. Specification*, Object Management Group, 1999.

[WK98] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1998.

About the Authors

Stuart Kent is a Senior Lecturer in Computing at the University of Kent, UK. He researches, teaches and mentors in design/modelling notations and techniques. He has publihsed some 30 refereed papers and regularly presents at major international conferences. This work arose out of a collaboration with the co-authors whilst on a consultancy engagement at Nortel Networks.

Stephen Gaito is a senior software engineer working for Nortel Networks. He is currently researching designs for, and helping to build, a model repository capable of managing complex multi-layer multi-protocol telecomms networks and services. Stephen's previous work with Nortel Networks was on a KBS based configuration engine capable of configuring complex Nortel Networks PBXs given a set of customer requirements.

Niall Ross is a senior employee of Nortel Networks. He leads a team researching how to manage complex multi-layer multi-protocol telecomms networks and services. Niall's previous work includes leading edge research on object-oriented methods, type systems and databases, and on software metrics. His current development unites KBS and OO approaches.