Core Meta-Modelling Semantics of UML: The pUML Approach

Andy Evans[1] and Stuart Kent[2]

[1] Department of Computer Science,
University of York, York, UK.
andye@cs.york.ac.uk
[2] Computing Laboratory, University of Kent,
Canterbury, UK.
s.j.h.kent@ukc.ac.uk

Abstract. The current UML semantics documentation has made a significant step towards providing a precise description of the UML. However, at present the semantic model it proposes only provides a description of the language's syntax and well-formedness rules. The meaning of the language, which is mainly described in English, is too informal and unstructured to provide a foundation for developing formal analysis and development techniques. Another problem is the scope of the model, which is both complex and large. This paper describes work currently being undertaken by the precise UML group (pUML), an international group of researchers and practitioners, to address these problems. A formalisation strategy is presented which concentrates on giving a precise denotational semantics to core elements of UML. This is illustrated through the development of precise definitions of two important concepts: generalization and packages. Finally, a viewpoint architecture is proposed as a means of providing improved separation of concerns in the semantics definition.

1 Introduction

The Unified Modeling Language (UML) [BRJ98,RJB99] is rapidly becoming a de-facto language for modelling object-oriented systems. An important aspect of the language is the recognition by its authors of the need to provide a precise description of its semantics. Their intention is that this should act as an unambiguous description of the language, while also permitting extensibility so that it may adapt to future changes in object-oriented analysis and design. This has resulted in a Semantics Document [Gro99], which is presently being managed by the Object Management Group, and forms an important part of the language's standard definition. The approach taken in the Semantics Document is to give a meta-model description of the language. This is presented in terms of three views: the abstract syntax, well-formedness rules, and modelling element semantics. The abstract syntax is expressed using a subset of UML static modelling notations. The abstract syntax model is supported by natural language descriptions of the syntactic structure of UML constructs. The well-formedness rules

are expressed in the *Object Constraint Language* (OCL). Finally, the semantics of modelling elements are described in natural language.

A potential advantage of providing a semantics for UML is that many of the benefits of using a formal language such as Z [Spi92] might be transferable to UML. Some of the major benefits of having a precise semantics for UML are given below:

Clarity: The formally stated semantics can act as a point of reference to resolve disagreements over intended interpretation and to clear up confusion over the precise meaning of a construct.
Equivalence and Consistency: A precise semantics provides an unambiguous basis from which to compare and contrast the UML with other techniques and notations, and for ensuring consistency between its different components.
Extendibility: The soundness of extensions to the UML can be verified (as encouraged by the UML authors).
Refinement: The correctness of design steps in the UML can be verified and precisely documented. In particular, a properly developed semantics supports the development of design transformations, in which a more abstract model is diagrammatically transformed into an implementation model.
Proof: Justified proofs and rigorous analysis of important properties of a system described in the UML require precise semantics. Proof and rigorous analysis are not currently supported by UML.
Tools: The tools that make use of semantics, for example a code generator or consistency checker, require that semantics to be precise, whether it be expressed as part of the standard or invented in the code by the tool developer.

Unfortunately, the current UML semantics are not sufficiently formal to realise many of these benefits. Although much of the syntax of the language has been defined, and some static semantics given, dynamic semantics are mostly described using lengthy paragraphs of often ambiguous informal English, or are missing entirely. Furthermore, little consideration has been paid to important issues such as proof, compositionality and rigorous tool support. A further problem is the extensive scope of the language, all of which must be dealt with before the language is completely defined.

This paper describes work being carried out by the precise UML (pUML) group and documented in [pG99,FELR98,EFLR98,ARKB99]. pUML is an international group of researchers and practitioners who share the goal of developing UML as a precise (formal) modelling language. The paper reports on work that aims to strengthen the existing meta-model semantics of UML. In section 2 a formalisation strategy is described (developed through the experiences of the group) that is being used to precisely describe the semantics of UML. This aims to give a precise denotational semantics to the core elements of UML. Section 3 identifies a core semantics model for UML, and in sections 4 and 5 an illustration is given of the formalisation of two core concepts: generalization and package.

Finally, section 6 proposes a 'model-instance viewpoint architecture' (MVA), as a route towards integrating the core semantics within the UML semantics.

2 The pUML approach

In this section, we briefly present some of the key objectives of the pUML group's approach to formalising the UML. The formalisation strategy that is currently adopted by the group is also described. A detailed discussion of the approach and formalisation strategy can be found in [ARKB99].

2.1 Working with the standard

An important aim of the pUML approach is to work firmly in the context of the existing UML semantics. The reasons for taking this approach (as opposed to developing our own semantic model) are as follows:

1. We recognise that UML is a standard and that considerable time and effort has been invested in the development of its semantics. It cannot be expected that radically different semantic proposals will be incorporated in new versions.
2. We believe that the existing UML semantics documentation and the meta-modelling approach already provide a good foundation for a precise semantics. As described below, the use of denotational semantics is the key to describing the semantics of UML precisely.

Thus, the pUML approach aims to identify and make precise areas of ambiguity and/or missing semantic details within the current UML meta-model.

2.2 Core semantics

To cope with the large scope of the UML it is natural to concentrate on essential concepts of the language to build a clear and precise foundation as a basis for formalisation. Therefore, the approach taken in the group's work is to concentrate on identifying and formalising a core semantic model for UML before tackling other features of the language. This has a number of advantages: firstly, it makes the formalisation task more manageable; secondly, a more precise core will act as a foundation for understanding the semantics of the remainder of the language. This is useful in the case of the many diagrammatical notations supported by UML, as each diagram's semantics can be defined as a particular 'view' of the core model semantics. For example, the meaning of an interaction diagram should be understandable in terms of a subset of the behavioural semantics of the core.

2.3 Adopting a denotational approach

One of the best known (and most popular) approaches to describing the semantics of languages is the denotational approach (for an in-depth discussion see [Sch86]). The denotational approach assigns semantics to a language by giving a mapping from its syntactical elements to a meaningful representation. For example, an association may denoted by a set of links between objects, while a class may be denoted by a set of objects.

UML already partially adopts the denotational approach to describe aspects of the language. The meta-modelling approach used in the UML semantics naturally supports the description of denotational relationships between model elements: model elements and their denotations can both be abstracted as conceptual classes, and the relationships between them can be formalised by associations and OCL constraints. It consequently makes good sense to continue using the denotational approach in the formalisation strategy. The distinguishing feature of the pUML approach is its emphasis on obtaining *precise* denotational descriptions of UML modelling elements.

2.4 Review and feedback

Constructing a semantics for a language as large and complex as UML is clearly not a simple task. Thus, obtaining feedback and reviews of semantic proposals is a key goal of the pUML approach. This is currently being achieved through publications, open collaborations and the group's web-site. Future aims of the group are to develop semantic tests, which can be used to validate the correctness of new semantic proposals. The use of formal notations to gain a alternative view of a semantic proposal is also used.

2.5 Tool support

Tool support is essential if the benefits of a precise semantics are to be realised. Sophisticated analysis and design tools (that support verification and refinement) require a meta-model semantics that can be implemented efficiently, and which supports sophisticated automation by tool vendors.

2.6 Formalisation strategy

In order to implement the pUML approach it is necessary to develop a strategy for formalising the UML. This is intended to act as a step by step guide to the formalisation process, thus permitting a more rigorous and traceable work program. The formalisation strategy consists of the following steps (a more detailed account can be found in [ARKB99]):

1. Identify specific modelling element/s that contribute to a core semantic model.

2. Iteratively examine the element/s, seeking to verify their completeness. Here, completeness is achieved when: (1) the modelling element has a precise syntax, (2) is well-formed, and (3) has a precise denotation in terms of some fundamental aspect of the core semantic model.
3. Use formal techniques to gain better insight into the existing definitions as shown in [FELR98,EFLR98].
4. Where in-completeness is identified, we attempt to address it by strengthening the existing model, or extending it in the most conservative way possible.
5. Feed the results into the UML meta-model, and disseminate to interested parties for feedback.

In the next section, we identify some core concepts for UML before showing how the strategy can be used to formalise their semantics.

3 The UML core

What parts of the UML semantics should be included in the core semantics? This question is already partially answered in the UML semantics document. It identifies a 'Core - Relationships' package and a number of 'Common Behaviour' packages. The Core Relationship package defines a set of model elements that are common to all UML diagrams, such as relationship, classifier, association and generalization. However, it only describes their syntax.

The Common Behavior package gives a partial denotational meaning to the model elements in the core package. For instance, it describes an association between classifiers and instances. This establishes the connection between the representation of a classifier and its meaning, which is a collection of instances. The meaning of an association (a collection of object links) is also given, along with a connection between association roles and attribute values. Finally, the Common Behaviour package also introduces the notion of an action and stimulus. These specifically relate to the modelling of behaviour in UML.

To illustrate the scope, and to show the potential for realising a compact core semantics, the relevant class diagrams of the two models are shown in the Figures 1 and 2. Well-formedness rules and some classes are omitted for brevity.

An appropriate starting point for a formalisation is to consider these two models in isolation, with the aim of improving the rigor with which the syntax of UML core elements are associated with (or mapped to) their denotations (core instances).

4 Generalization/Specialization

This section presents a precise definition of the meaning of generalization and specifically how it relates to instance conformance. The presentation is more structured and detailed than above due to the greater number of omissions in this part of the UML semantics document.

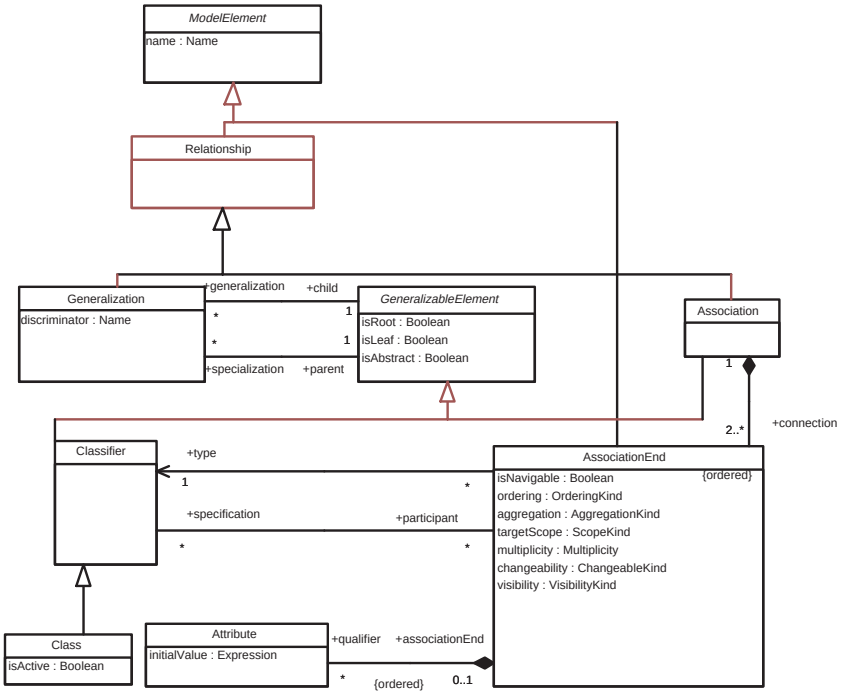

Fig. 1. Fragment of the core relationships package

4.1 Informal description

In UML, a generalization is defined as "a taxonomic relationship between a more general element and a more specific element", where "the more specific element is fully consistent with the more general element (it has all of its properties, members, and relationships) and may contain additional information" [Gro99] (page 2-35) .

Closely related to the UML meaning of generalization is the notion of direct and indirect instances: This is alluded to in the meta-model as the requirement that "no object may be a direct instance of an abstract class, although an object may be an indirect instance of one through a subclass that is non-abstract" [Gro99] (page 2-59).

UML also places standard constraints on subclasses. The default constraint is that a set of generalizations are disjoint, i.e. " (an) instance of the parent (class) may be an instance of no more than one of the given children .." [Gro99] (page 2-36). Abstract classes enforce a further constraint, which implies that no instance can be a direct instance of an abstract class.

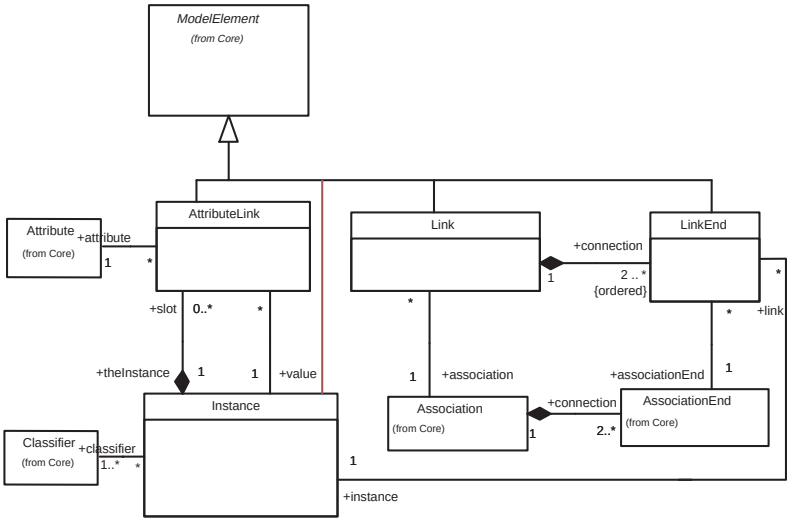

Fig. 2. Fragment of the common behaviour package

We now examine whether these properties are adequately specified in the UML semantics document.

4.2 Existing formal definitions

Bruel and France [BR98] have defined a formal model of generalization. Classes are denoted by a set of object references, where each reference maps to a set of attribute values and operations. Generalization implies inheritance of attributes and operations from parent classes (as expected). In addition, class denotations are used to formalise the meaning of direct and indirect instances, disjoint and abstract classes. This is achieved by constraining the sets of objects assigned to classes in different ways depending on the roles the classes play in a particular generalisation hierarchy. For example, assume that `Ai` is the set of object references belonging to the class `A` and that `B` and `C` are subclasses of `A`. Because instances of `B` and `C` are also instances of `A`, it is required that $\texttt{Bi} \subseteq \texttt{Ai}$ and $\texttt{Ci} \subseteq \texttt{Ai}$, where `Bi` and `Ci` are the set of object references of `B` and `C`.

This model also enables constraints on generalisations to be elegantly formalised in terms of simple constraints on sets of object references. In the case of the standard 'disjoint' constraint on subclasses, the following must hold: $\texttt{Bi} \cap \texttt{Ci} = \emptyset$, i.e. there can be no instances belonging to both subclasses. For an abstract class, this constraint is further strengthened by requiring that `Bi` and `Ci` partition `Ai`. In other words, there can be no instances of `A`, which are not instances of `B` or `C`. This is formally stated as $\texttt{Ai} = \texttt{Bi} \cup \texttt{Ci}$.

We will adopt this model in order to strengthen the existing meta-model definition of generalization as it applied to classifiers and classes.

4.3 Syntax and well-formedness

The abstract syntax of the Generalization model element is described by the meta-model fragment shown in Figure 3. This is taken from the core relationships package of the UML Semantics Document.

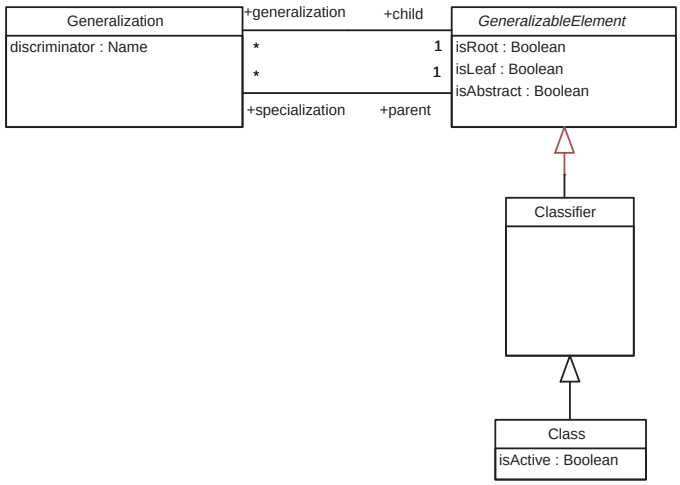

Fig. 3. Syntax of Generalization/Specialization

The most important well-formedness rule which applies to this model element, that is not already ensured by the class diagram, is that circular inheritance is not allowed. This constraint is decribed using the Object Constraint Language (OCL). Assuming 'allParents' returns all the parents of GeneralizableElement, then it must hold that:

```
context GeneralizableElement
invariant
not self.allParents -> includes(self)
```

Here, the *context* of the OCL expression is any instance of a GeneralizableElement.

4.4 Semantics

The completeness of the semantic formalisation versus the desired properties of generalizations is now examined. We restrict ourselves to examining whether

the properties of instance conformance, identified in section 4.2, are preserved by the meaning of Generalisation/Specialisation in the UML semantics. Therefore the most important denotational relationship to be examined is that between a classifier and instance. This relationship is formalised in the existing UML semantics by the meta-model fragment shown in Fig 4. This denotes the fact that a Classifier is described by the set of objects that may be instantiated from it. Note that the more generic term for a class in UML is the classifier.

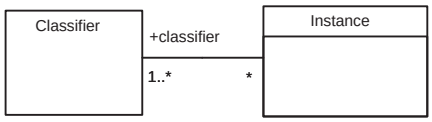

Fig. 4. Meta-model fragment for Classifier and Instance relationship

However, unlike the formal model described in section 4.2, the UML meta-model does not describe the meaning of generalization in terms of the Classifier/Instance relationship. Thus, in order to give a precise denotational meaning to a Generalization, the meta-model must be strengthened with additional constraints on the relationship between the Generalization model elements and the Classifier-Instance relationship.

Indirect instances The first constraint relates to the meaning of indirect instances.

An Instance of a Classifier is also an *indirect* Instance of its parent Classifiers. This is specified as follows:

```
context c : Classifier
invariant
    c.generalization.parent -> forall(s : Classifier |
        s.instance -> includesAll(c.instance))
```

Instance identity Unfortunately, this constraint does not guarantee that every instance is a direct or indirect instance of a related classifier (it only states that generalization implies the existence of indirect instances).

Thus, an additional constraint must be added in order to rule out the possibility of an instance being instantiated from two or more un-related classifiers. This is the *unique identity* constraint:

```
context i : Instance
invariant
    i.classifier -> exists(direct : Classifier |
        direct.allParents -> union(Set{direct}) = i.classifier
```

This states that the *only* Classifiers that an object can be instantiated from are either the Classifier that it is directly instantiated from or its parents [1].

Direct instances The meaning of a direct instance can now be precisely defined:

```
context i : Instance
isDirectInstanceOf(c : Classifier) : Boolean
isDirectInstanceOf(c) = c.allParents -> union(Set{c}) = i.classifier
```

A direct Instance directly instantiates a Classifier and indirectly instantiates its parents.

Disjoint constraints Once direct and indirect Instances are formalised, it is possible to give a precise description to the meaning of constraints on generalizations (for example the disjoint constraint).

The disjoint constraint can be formalised as follows:

```
context c : Class
invariant
    c.specialization.child -> forall(i,j : Classifier |
        i <> j implies i.instance ->
            intersection(j.instance) -> isEmpty)
```

For any two children of a Classifier, `i` and `j`, the set of instances of `i` will be disjoint from the set of instances of `j`. Note that the disjoint constraint is only applied to Classes in UML, not Classifiers.

Abstract classifiers Finally, the following OCL constraint formalises the required property of an abstract classifier that it cannot be directly instantiated:

```
context c : Classifier
invariant
    c.isAbstract implies
        c.specialization.child.instance -> asSet = c.instance
```

Note, the result of the specialization.child path is a bag of instances belonging to each subtype of `c`. Applying the asSet operation results in a set of instances. Equating this to to the instances of `c` implies that all the instances of `c` are covered by the instances of its children. This, in conjunction with the disjoint property above, implies the required partition of instances, and completes the formalisation of this concept.

[1] The UML standard does in fact state that static/dynamic multiple classification of instances is permitted without generalization being present. However, the conditions under which this is permitted are not defined, and we therefore defer consideration of this aspect for now.

5 Package Instances

Links, link ends and objects do not generally appear in isolation. A UML object diagram represents a *snapshot* [DA98] of the state of the system. Yet there is no corresponding concept in the meta-model. A reason for this is given in the accompanying English semantics, [Gro99] (page 2-179):

> The purpose of the package construct is to provide a general grouping mechanism. A package cannot be instantiated, thus it has no runtime semantics.

This is a particularly implementation-oriented perspective. Object diagrams can be drawn to show instances of specification models (think of tools which simulate that model) as much as instances of implementation models. As has been identified in the discussion on abstract classes, whether something is instantiable or not at execution is captured not by whether it can or can not have instances, but whether those instances are only instances of that thing.

So, in this section, we develop the idea of instances of packages, corresponding to object diagrams. This also caters for instances of models and subsystems. We believe that this concept is essential for formalising the semantics of behavioural constraints specified in a package. Again, the model-instance view is emphasised in our formalisation.

Although a package is defined to be a collection of model elements (which include instance as well as design elements), it is constrained [Gro99] (page 2-175, [1]) only to include design elements. This suggests a new class is required to capture the concept of a package instance. The relevant class diagram is given in Fig 5.

Classes have also been added to make the distinction between design and instance element clearer, although only some of the different kinds of design and instance elements have been shown on the diagram. The new classes make the OCL constraint in [Gro99] (page 2-175, [1]), redundant.

There is another issue that needs to be considered – but not here as it is beyond the scope of this paper. The allContents associations from Package and PackageInstance, respectively, are examples of the recursive composite pattern. In that pattern there is usually another association to collect together all primitive (as opposed to composite) elements that are contained in the composite either directly, or indirectly through other composites that are elements of the composite. The meta-model has nothing to say about what it means for a package to be contained within another, or the relationship of that concept to package imports. Some hints are provided in the accompanying English semantics, though the description given for packages in general seems at odds with the description given for models. A similar issue arises for package instances.

The association 'accessed' from Package is an attempt at capturing the notion, mentioned in the informal semantics [Gro99] (page 2-173), that elements from other packages may be accessed from a package. The association 'accessible' then represents all those elements which are accessible from a package, i.e. those contained within it and those outside which it is able to access:

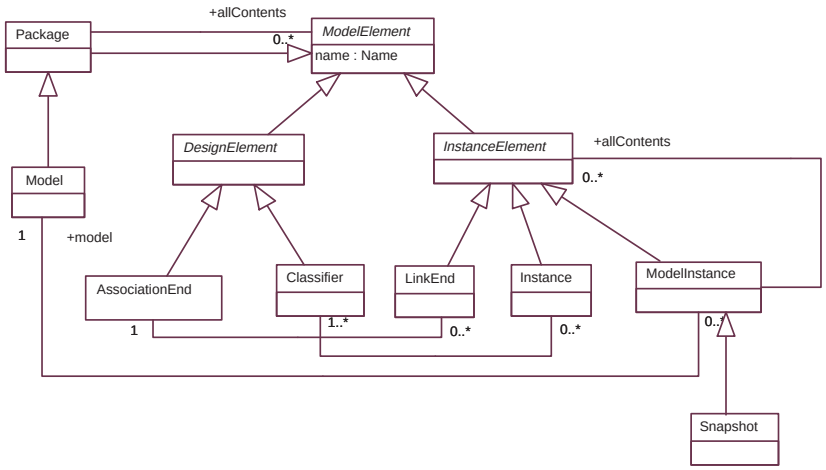

Fig. 5. Meta-model fragment for packages and their instances

```
context p : Package
invariant
    p.accessible = p.accessed->union(p.allContents)
```

At least one further constraint is required, that associations only refer to classifiers accessible to the package.

There can be many kinds of package instance. For example, to give the semantics for dynamic behavioural constraints, it is likely that an idea of a *trace* (in the formal sense of the word) or *filmstrip*, corresponding to an instance of the execution of some sequence of actions or operations will be required. In this paper we restrict ourselves to *snapshots*, instances that correspond to object diagrams. Snapshot is a subclass of PackageInstance. It is only associated with Instance's and Link's.

Of course it is not the case that any snapshot can be an instance of any package. Specifically, the links and instances must be of associations and classifiers, respectively, accessible to that package:

```
context s : Snapshot
invariant
    s.package.accessible->select(oclIsKindOf(Association))
        ->includesAll(s.allContents->select(oclIsKindOf(Link)).association
    and s.package.accessible->select(oclIsKindOf(Classifier))
        ->includesAll(s.allContents->select(oclIsKindOf(Instance)).classifier
```

If a link is of an association accessible to the package then that guarantees that link ends are of association ends accessible to the package. Similarly,

instances at the ends of link ends are guaranteed to be of classifiers in the package. This is because we ensured earlier that associations only connect classifiers accessible to the package.

5.1 Constraints

According to the meta-model, all modeling elements may be subject to constraints [Gro99] (page 2-14). With the separation of design and instance elements, this can be refined by associating constraints with design elements only. A constraint may take on many forms: for example, there can be constraints on classes such as the OCL invariants used to constrain the meta-model in this paper, or constraints on actions which are contracts comprising pre and post conditions written in OCL. This suggests that the decision in the current version of the meta-model to pin down a constraint so that it has a body which is a simple boolean expression may be premature – only invariant constraints have this structure. The class diagram in Fig 6 captures these revisions.

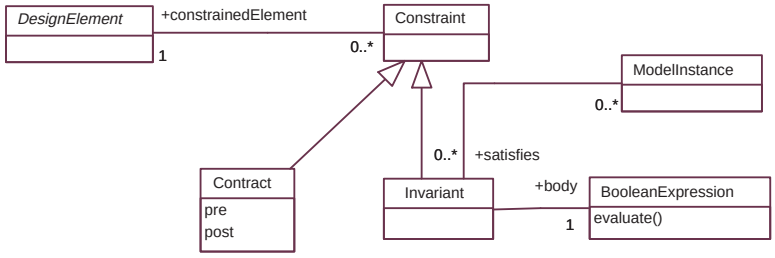

Fig. 6. Meta-model fragment for constraints

The diagram focuses on a single kind of constraint, namely invariants on packages.

6 A Model-Instance Viewpoint Architecture

In terms of future work, it is important to understand how the pUML approach should fit within the overall architecture of the UML semantics. Currently, the UML semantics adopts a rather ad-hoc approach, in which its various elements are distributed throughout a number of different packages, for example, there are a state-machine, model-management and use-case packages. Each of these packages has some overlap with each other, but because this is not made explicit in the architecture, understanding and maintaining the model is difficult.

An alternative architecture is one that makes a clear distinction between the core semantics, which describe the essential concepts and meaning of a language,

and viewpoints, which describe the different ways in which the core concepts can be viewed by the modeller. For example a static modelling view encompasses classes and objects. Furthermore, viewpoints can be directly related to the diagrammatical notations that are used to visually represent models, for example class and association icons, etc.

The viewpoint architecture has been successfully applied in the development of the RM-ODP standard [ISO96], where it is used to describe multiple views of open distributed systems. The advantage of adopting a viewpoint-oriented architecture is that it places clear boundaries on the roles that different parts of the semantics plays. While the core semantics makes clear the meaning of the language, the views and diagram elements specify the syntactical features of the language (which is essential to tool designers).

Figure 7 gives an overview of an architecture that both supports a viewpoint-oriented model of the UML semantics, and which also places emphasis on precisely documenting the relationships between model elements and instances (denotations).

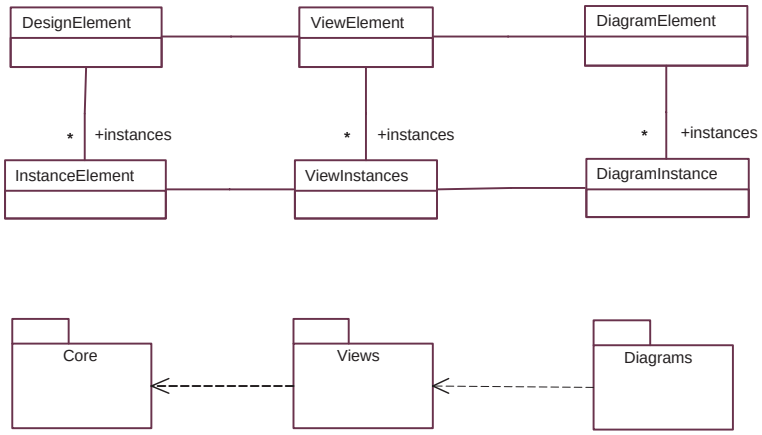

Fig. 7. Model-Instance View Architecture (MVA)

Here, the semantics is to divided into three main packages: the core, views and diagrams-packages. Within these packages, the relationship between model elements and their denotations (instances) are given by the 'instances' associations. This makes explicit the fact that most denotations in UML consist of mappings from generic concepts to the instances which they represent. In the core-package, elements might include modelling concepts such as associations and classes, while their instances are objects and links. In the view-package, views map view elements and instances to appropriate core elements and in-

stances. A behavioural view, for example, would provide a mapping to only behavioural modelling elements in the core, such as operations and actions. Finally, the diagrams-package provides a link between diagram meta-models and their instances (class icons, etc.) to elements in the core model (through possibly many viewpoints).

7 Further Work

This paper has described an approach to the semantics of UML which builds upon the meta-model defined in the UML standard semantics document. Fragments of the semantics have been shown here, specifically the semantics of associations and generalisation, and the introduction of snapshots which will play a pivotal role in the semantics of behavioural constraints. A model-instance viewpoint architecture has been proposed as a way of integrating the core semantics into the complete UML meta-model.

Our immediate goal is to complete this semantics for a core notation set, seeking compliance with the current UML standard. The core is likely to include a static and a dynamic aspect. We can also distinguish between intra- and extra-package. The static part, intra-package, will include associations, classes and OCL invariants. The dynamic part, intra-package, will include actions, pre/post conditions for actions and action compositions (e.g. sequences). The extra-package part focuses on relationships between packages. In the best tradition of "bootstrapping" the meta-model itself will be written in the smallest subset possible of the static part of the core-classes, simple associations and OCL invariants should be sufficient.

In the longer term, our intention is to give a semantics to the complete notation set, by mapping into the core, extending the core only when there is not already a concept which suffices. Of course one role of semantics is to clarify and remove ambiguities from the notation. Therefore we will not be surprised if we find that the notation needs to be adjusted or the informal semantics rewritten. However, we will be able to provide a tightly argued, semantically-based recommendation for any change deemed necessary.

Some consideration also needs to be given to quality assurance. There are at least three approaches we have identified:

1. peer review and inspection
2. acceptance tests
3. tool-based testing environment

So far the only feedback has come from 1. Since a meta-model is itself a model, acceptance tests could be devised as they would be for any model. Perhaps "testing" a model is a novel concept: it at least comprises devising object diagrams, snapshots, that the model must/must-not accept. Better than a list of acceptance tests on paper would be a tool embodying the meta-model, that allowed arbitrary snapshots to be checked against it.

References

[ARKB99] A.S.Evans, R.B.France, K.C.Lano, and B.Rumpe, *Towards a core meta modelling semantics of UML*, Behavoral Specifications for Businesses and Systems (Haim Kilov, ed.), Kluwer Press, 1999, To appear.

[BR98] J-M. Bruel and R.B.France, *Transforming UML models to formal specifications*, UML'98 - Beyond the notation, LNCS 1618, Springer-Verlag, 1998.

[BRJ98] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language user guide*, Addison-Wesley, 1998.

[DA98] D. D'Souza and A.C.Wills, *Objects, components and frameworks with UML*, Object Technology Series, Addison-Wesley, 1998.

[EFLR98] Andy Evans, Robert France, Kevin Lano, and Bernhard Rumpe, *Developing the UML as a formal modelling notation*, UML'98 Proceedings (Jean Bezivin and Pierre-Allain Muller, eds.), Springer-Verlag, LNCS 1618, 1998.

[FELR98] R. France, A. Evans, K. Lano, and B. Rumpe, *The UML as a formal modeling notation*, Computer Standards & Interfaces **19** (1998).

[Gro99] Object Management Group, *OMG Unified Modeling Language Specification, version 1.3beta. found at: http://www.omg.org*, 1999.

[ISO96] ISO/IEC, *Reference model of open distributed processing - part 1-5, ISO/IEC DIS 10746-1*, Tech. report, 1996.

[pG99] The pUML Group, *The precise UML web site: http://www.cs.york.ac.uk/puml*, 1999.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language reference manual*, Addison-Wesley, 1999.

[Sch86] D. A. Schmidt, *Denotational semantics: A methodology for language development*, Allyn and Bacon, 1986.

[Spi92] J.M. Spivey, *The Z reference manual, 2nd edition*, Prentice Hall, 1992.