

# Appendix A

## A Kernel Library to Support Profile Definitions

(c)December 1999  
Andy Evans, Stuart Kent

### 1 Introduction

The purpose of this short appendix is to illustrate some of the general features of the kernel library that is proposed in the main document. As already discussed, the kernel library aims to provide a collection of basic constructs for building UML profiles. These constructs can be reused and combined quickly and flexibly to build profiles in a well structured, layered fashion. By emphasising an architecture in which a clear distinction is made between syntax and semantic mappings, the kernel library also aims to encourage the development of very precise profile definitions.

This appendix primarily focuses on the static constructs of the library. It gives a description of their semantics based on the denotational approach to describing semantics. An example use of the kernel library is then illustrated by showing how a meta-modelling profile can be developed as an extension of the kernel.

### 2 Components of the Kernel Library

The components of the kernel library are divided up into the following five packages:

- **Fundamentals:** abstract meta-modelling concepts such as relationship, relatable, instantiable, instance, container, contained, generalisable and generalisation
- **Static basics:** generalised constructs for modelling the static properties of systems.
- **Constraint basics:** constructs relating to the expression of constraints.
- **Dynamic basics:** constructs for modelling the behaviour of systems.
- **Model management basics:** general mechanisms for extending and specialising the components of the language.

These are illustrated in Figure 1.

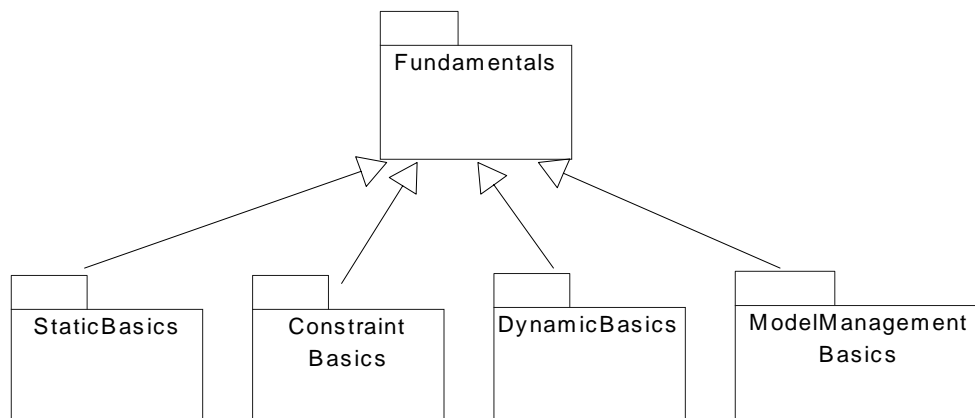


Figure 1: Components of the Kernel Library

### 3 Fundamentals

The fundamentals package provides abstract concepts and structures that are commonly used in software modelling language meta-models. These include concepts such as: relationship, relatable, instantiable, instance, container, contained, generalisable and generalisation. All basic modelling constructs can be viewed as extensions of these concepts.

Determining precisely what should be included in this package will ultimately require a survey of the meta-models of many different modelling languages. Thus, details of this package are omitted for now..

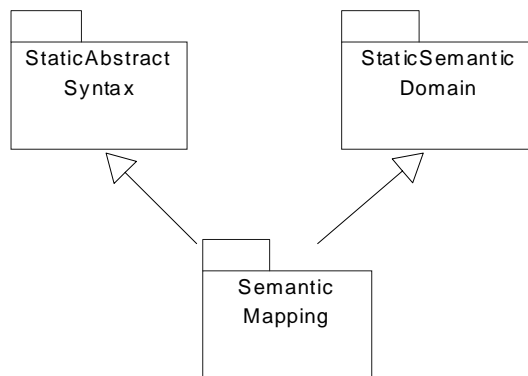
### 4 Static Basics

The static basics package includes the basic static modelling constructs required to build a UML semantics meta-model. However, what should be included in this package? At the most fundamental level, a static UML model (an instance of the UML meta-model) describes the set of valid configurations of objects and links (between objects) that can exist in the model. For example, a static model of a Network might describe the nodes (objects) and routes (links between nodes) that can exist in the Network. Other constructs, e.g. associations, generalization, textual descriptions, add nothing more than constraints to this fundamental model.

Thus, the static basics package must be able to represent the following concepts and properties:

- Objects, which are instances of classes.
- Links, which are instances of attributes.

To achieve this, the static basics package is decomposed into three sub-packages. These describe the abstract syntax, semantic domain and mapping rules from abstract syntax to semantics domain for each construct in the static basics package. The generalization arrows indicate that the static mapping package imports and extends the constructs in the static abstract syntax and static semantic domain packages<sup>1</sup>.



**Figure 2: Packages in Static Basics**

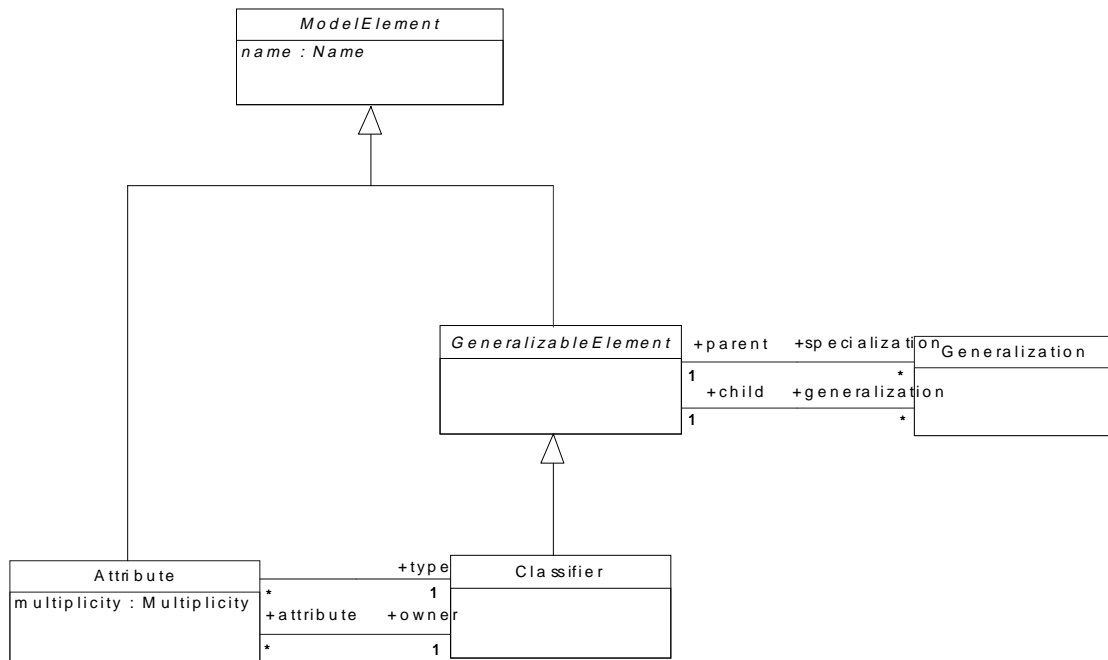
---

1. The semantics of sub-packaging and package extension are based on those proposed by the Catalysis method. Briefly, a package may include sub-packages. When a package is imported, all its model elements, including any packages are copied into the importing package. Imported elements can then be extended and constrained, but no elements may be removed.

## 4.1 Abstract Syntax

The abstract syntax package for the static basics package is shown in detail in Figure 3. The main constructs provided by the package are classifiers and attributes, which are themselves subclasses of model elements. In addition, the generalization/ specialization relationship is also introduced because of its primary importance in describing the taxonomic properties of static models.

The constructs in this model are a subset of the constructs supported by the core package in the UML semantics document. This will allow greater freedom in the ways in which the package can be extended.



**Figure 3: Abstract Syntax for the Static Basics**

### *Well-formedness rules*

The following well-formedness rules apply to the static abstract syntax package:

[1] No two attributes belonging to the same classifier can have the same name.

```

context Classifier
self.allAttributes-> forAll ( p, q | p.name = q.name implies p = q )
  
```

Where the operation *allAttributes* returns all the *Attributes* of a *Classifier* (including those of its parents).

[2] Circular inheritance is not allowed.

```

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

Where the operation *allParents* returns a *Set* containing all the *GeneralizableElements* inherited by this *GeneralizableElement*, excluding the *GeneralizableElement* itself.

## 4.2 Semantic Domain

The static semantic domain package describes the semantic domain of the static abstract syntax. As discussed above, a UML model is described by a collection of objects (instances) and links, as shown in Figure 4.

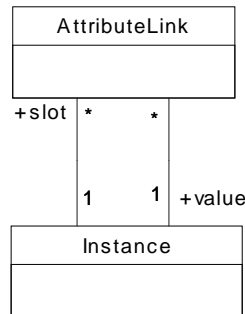


Figure 4: Static Semantics Domain Package

## 4.3 Semantic Mapping

The semantic mapping package imports all the constructs of Figure 3 and Figure 4. It associates constructs in the abstract syntax package with their semantic domains, and is shown in Figure 5.

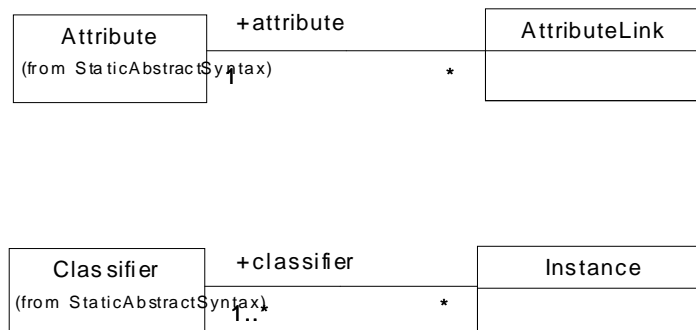


Figure 5: Static Basics Semantic Mapping Package

A classifier is denoted the set of instances that can be instantiated from it. Conversely, an instance may be a direct instance of many classifiers. This is permitted because certain profiles in UML may allow multiple classification, i.e. instances that can be directly instantiated from many classifiers. This property can, however, be easily ruled out in a profile if required.

An attribute is denoted by a set of attribute links. An attribute link represents a link between an instance (the owner of the attribute) and the value of the attribute (another instance).

In addition, the following OCL expressions precisely describe the constraints that apply to mapping between abstract syntax to semantic domain:

[1] The type of the Instance must match the type of the Attribute.

```

context AttributeLink
self.value.classifier ->includes (self.attribute.type -> union(self.allChildren))
  
```

Where the operation allChildren results in a set containing all the children of a given Classifier.

[2] The AttributeLinks match the declarations in the Classifiers.

```
context Instance
self.slot->forall ( a | self.classifier->exists ( c |
    c.allAttributes->includes ( a.attribute ) ) )
```

[3] The number of associated AttributeLinks must match the multiplicity of the Attribute.

```
context Instance
self.classifier.allAttributes->forall ( a | a.multiplicity.multiplicityRange->exists ( mr |
    self.selectedAttributeLinks ( a )->size >= mr.lower.oclAsType(Integer) and
    (mr.upper = 'unlimited' or
    (mr.upper <> 'unlimited' and
    self.selectedLinkEnds ( a )->size <=
    mr.upper.oclAsType ( Integer ) ) ) ) )
```

Where the operation selectedAttributeLinks results in a set containing all AttributeLinks corresponding to a given Attribute.

The following meta-model fragment taken from the UML semantics document describes the relationship that exists between Multiplicity and MultiplicityRange:

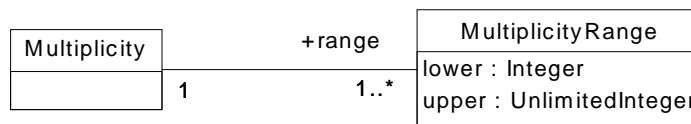


Figure 6: Relationship between Multiplicity and Multiplicity Range

## 5 Model Management

The model management package contains constructs that are used to manage the complexity of building large UML models. The primary model management construct is the package. A package is a container of UML model constructs. Unlike the UML 1.3 semantics, it is proposed that the package is quite sufficient for describing all types of containers, including models and sub-systems. Therefore, these constructs are redundant in the kernel library.

As before, the model management package is divided into the following packages: abstract syntax, semantic domain and semantic mapping (not shown here).

### 5.1 Abstract Syntax

The model management abstract syntax package is shown in Figure 7. A package is generalizable element and contains a collection of model elements.

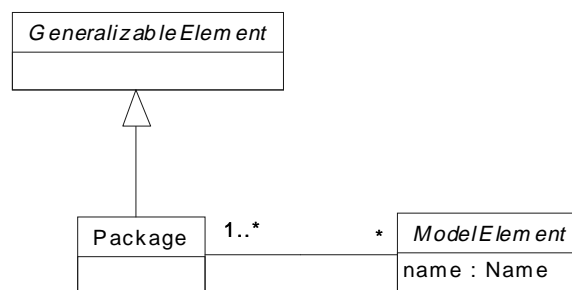


Figure 7: Model Management Abstract Syntax Package

## Well-formedness rules

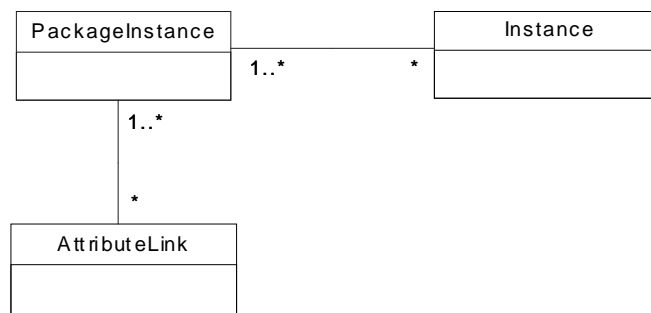
[1] No two model elements in a package may have the same name.

context Package

self.modelElement -> forall(p,q | p.name = q.name implies p = q)

## 5.2 Semantics Domain

The model management semantics domain package is shown in

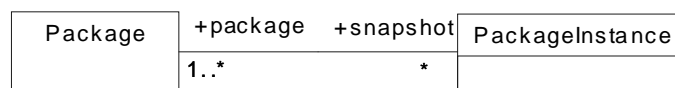


**Figure 8: Model Management Semantics Domain Package**

A package instance is a collection of instances and attribute links.

## 5.3 Semantic Mapping

Figure shows the model management semantic mapping package. It imports all the constructs of Figure 7 and Figure 8.



**Figure 9: Model Management Semantic Mapping Package**

A package is denoted by a set of package instances. Each package instance represents a *snapshot* of attribute links and instances that will conform to the classifiers and attributes belonging to the package. The complete mapping package, including imported constructs is shown in Figure 10.

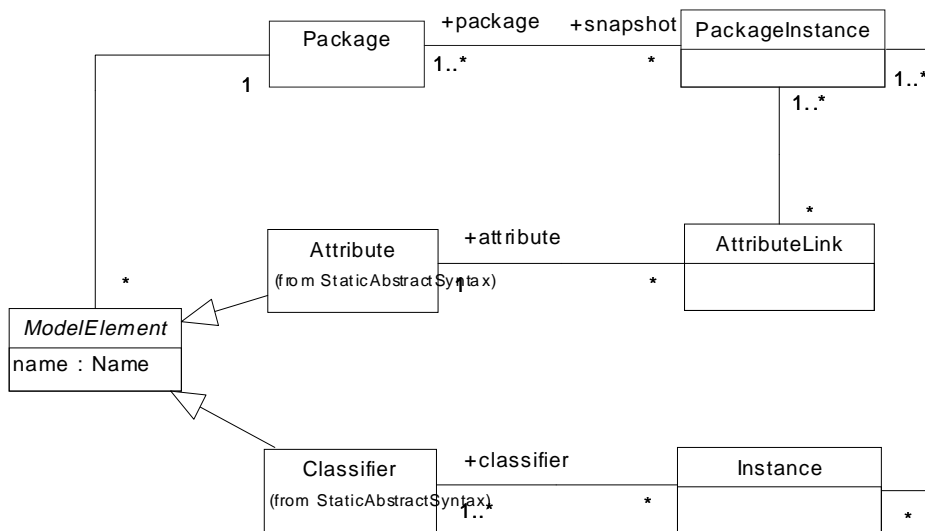


Figure 10: Complete Mapping Package

## 6 Other Packages

The definition of the dynamic basics and constraint basics packages would be structured in the exactly the same way as the above examples. For the sake of brevity, they are omitted here.

## 7 Extending the Model

Once the kernel library has been defined, it can be used to build profiles to support different uses of UML. This is where the power of the library mechanism becomes apparent. The library provides the basic constructs from which new constructs can be readily defined, thus easing the work of the profile designer.

In order to build a new profile, constructs from the library are extended. This involves adding new classes, attributes, associations and constraints. Note, that any profile must satisfy the rules for extending packages (details of which can be found in the main document). As an example, a profile for meta-modelling is now presented. It provides the key constructs that are used to build meta-models in UML. To achieve this, it extends the static, constraint and extension constructs of the kernel library, whilst placing restrictions on the way in which some of the constructs can be used. For example, multiple classification and n-ary associations are not desirable in a simple meta-modelling language, and are therefore ruled out by constraints. Figure 11 illustrates the necessary package extensions.

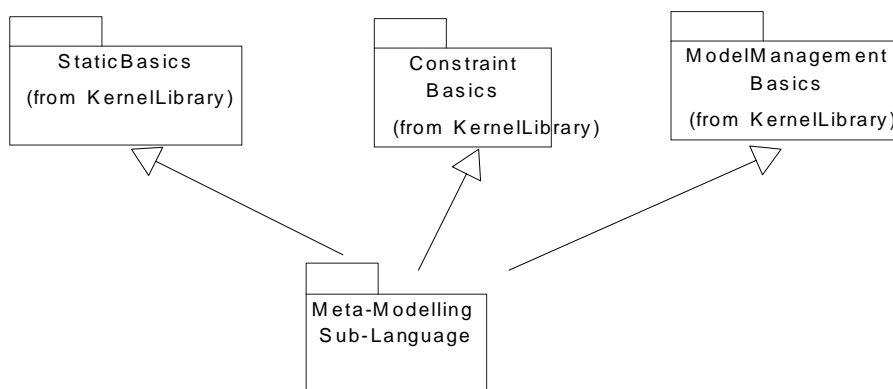


Figure 11: Meta-modelling Sub-language Profile

## 7.1 Extensions to Static Basics Package

An example extension to the static basics package is shown in Figure 12. Note, that each component of the sublanguage extends the abstract syntax, semantic domain and semantic mapping packages of the static basics package.

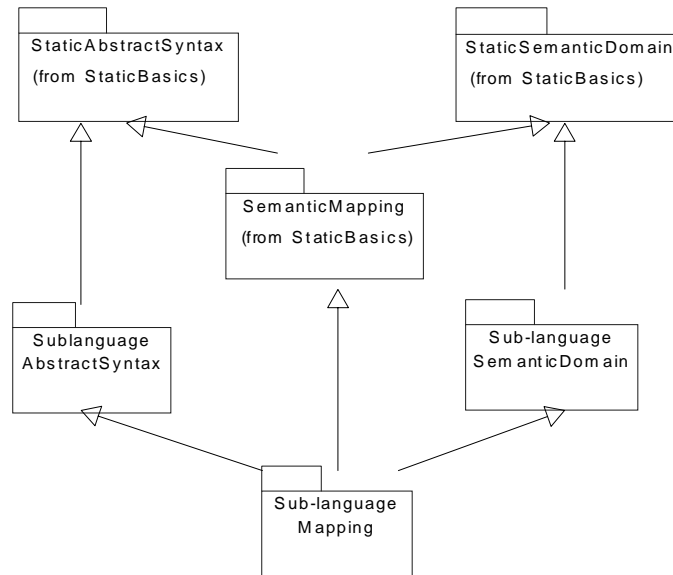


Figure 12: Meta-modelling Sublanguage Extensions

## 7.2 Abstract Syntax

Figure 13 shows the details of the sublanguage abstract syntax package. It extends the kernel language with two extra constructs: binary associations and abstract classifiers.

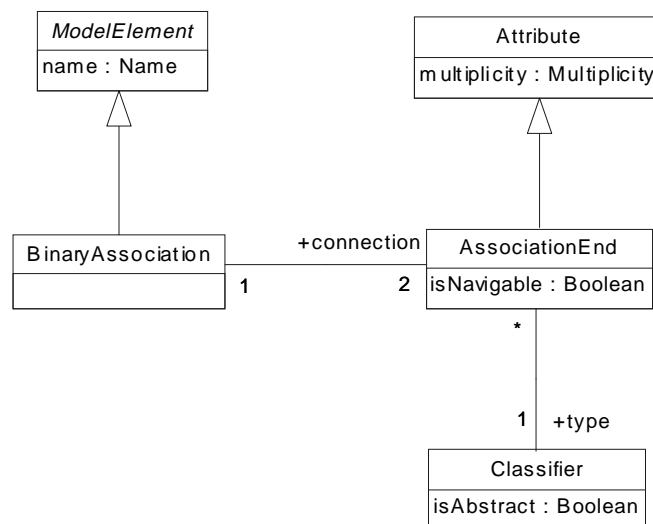


Figure 13: Extensions to Static Abstract Syntax Package

Binary associations associate exactly two classifiers (which may be the same). Every association has two association ends. An association end is a subclass of an attribute. Its type is the classifier to which the association end is connected. The `isNavigable` attribute is used to indicate whether it is possible to navigate along an association end.

An abstract classifier is a classifier that cannot be instantiated. The attribute, `isAbstract`, is used to indicate whether a classifier is abstract or not.

## *Well-formedness Constraints*

[1] The AssociationEnds must have a unique name within the Association.

```
self.connection->forAll( r1, r2 | r1.name = r2.name implies r1 = r2)
```

## **7.3 Semantics Domain**

No new constructs are required in the meta-model sublanguage semantics domain. This is typical of building profiles using the kernel library: because the basic semantic foundation has already been defined, further extensions tend only to require the addition of new constraints and/or associations in the semantic mapping package.

## **7.4 Semantic Mapping**

In this package, additional constraints are required to capture the specific semantic properties of the new constructs that have been added. For example, a binary association is defined in terms of two attributes. Each attribute represent a uni-directional link between instances that are related by the association. Thus, an additional constraint must be added to ensure that every uni-directional link has a corresponding inverse link. Other constraints are introduced to precisely define the meaning of abstract classifiers and to rule out multiple instantiation, which is too rich a concept for the sublanguage.

[1] Attribute links belonging to association ends always have a reverse link.

```
context Association
self.associationEnd -> forall(e1, e2 | e1.name <> e2.name implies
  e1.attributeLink -> forall(a1 | e2.attributeLink ->
    exists(a2 | a2.instance = a1.value and a2.value = a1.instance)))
```

[2] Abstract classifiers cannot be instantiated.

```
context Classifier
self.isAbstract implies self.instance -> isEmpty
```

[3] Multiple instantiation is not permitted.

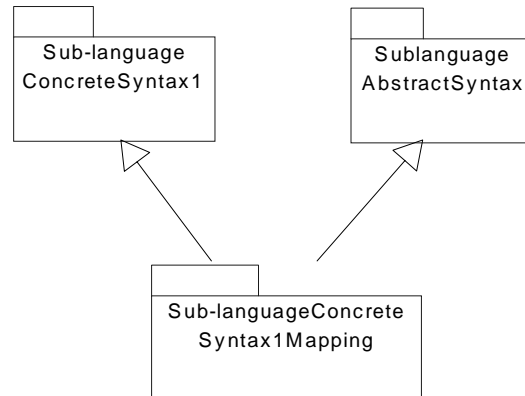
```
context Instance
self.classifier -> size = 1
```

## **7.5 Further Extensions**

In order to provide a complete definition of the meta-modelling sublanguage profile, extensions will need to made to the other model management basics and constraint basics packages. Specifically, these will include constraints on the importing of packages and the introduction of constraints of generalization relationships such as disjoint and overlapping constraints.

## 8 Concrete Syntax

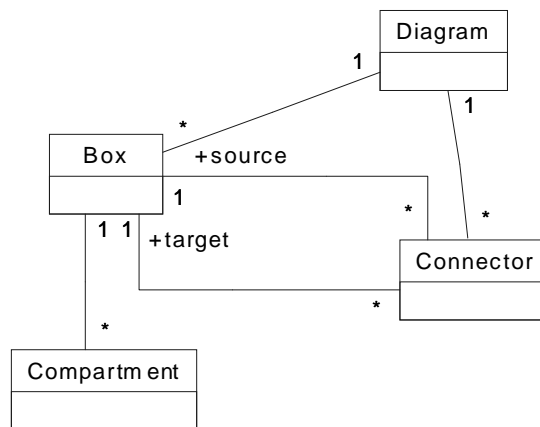
Once the abstract syntax and semantics have been defined for the profile, a concrete syntax can be built. Figure 14 shows the general architecture of this part.



**Figure 14: Concrete Syntax Architecture for the Meta-modelling Sub-language Profile**

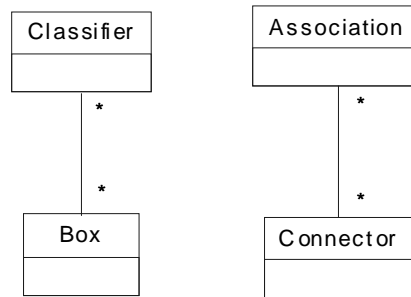
The Meta-modelling Sub-language can quite validly support a number of concrete syntaxes, so this architecture can be extended with further concrete syntax packages if required.

The concrete syntax package is shown in Figure 15. A diagram consists of connectors and boxes. A connector connects a box to another box and box may contain a number of compartments.



**Figure 15: Concrete Syntax Package**

Figure 16 shows the mapping between concrete and abstract syntax for the above package. A class is realised by a box with two compartments, and an association is realised by a connector whose source and target boxes are the realisations of the source and target classes of the association.



**Figure 16: Mapping from Concrete to Abstract Syntax**

The following OCL invariants record the fact that a class/association is realised by a single box/connector in any one diagram, a class must be realised by a box with one or two compartments, and that the connector realising the association must connect boxes that realise the classes at the source and target of the association:

```

context c:Classifier
  c.boxes->size=c.boxes.diagram->size

context a:Association
  a.source.boxes->forAll(b |
    a.connectors->select(c|c.diagram=b.diagram).source=b)
  and a.target.boxes->forAll(b |
    a.connectors->select(c|c.diagram=b.diagram).target=b)

context c:Classifier
  c.boxes->forAll(b | let n=b.compartments->size in n>=1 and n<=2)
  
```

## 9 Conclusions

This note has illustrated some key constructs and a typical architecture of the proposed kernel library. It has been shown that the library provides a useful foundation for building UML profiles. Furthermore, the layered approach to building the kernel library and meta-modelling sublanguage profile has enabled construction to be carried out in an incremental fashion, thus reducing the risk of introducing ambiguities and mis-conceptions into the meta-model.