Engineering Modelling Languages: A Precise Meta-Modelling Approach

Tony Clark1 [1], Andy Evans [2] , and Stuart Kent [3]

[1] Department of Computer Science, King's College London, UK
anclark@dcs.kcl.ac.uk

[2] Department of Computer Science, University of York, UK
andye@cs.york.ac.uk

[3] Computing Laboratory, University of Kent at Canterbury, UK
s.j.h.kent@ukc.ac.uk

Abstract. MMF uses meta-modelling techniques to precisely define modelling languages. The approach employs novel technology based on package specialisation and templates. MMF is being applied to the UML 2.0 revision initiative and is supported by a tool.

1 Introduction

This paper describes a Meta-Modelling Framework (MMF) that addresses many of the deficiencies in the current definition of The Unified Modeling Language (UML) [18]. The facility comprises a language (MML) for defining modelling notations, a tool (MMT) that checks and executes those definitions, and a method (MMM) consisting of a model based approach to language definition and a set of patterns embodying good practice in language definition. The development of MMF by the pUML group ([14]) is ongoing and has been supported by IBM and Rational Inc. The work reported in this paper is a simplified version of the work described in out initial submission to the UML 2.0 revision initiative [13] [4] which is expected to be completed in 2002. This paper describes the components of MMF and uses them to develop a simple modelling language.

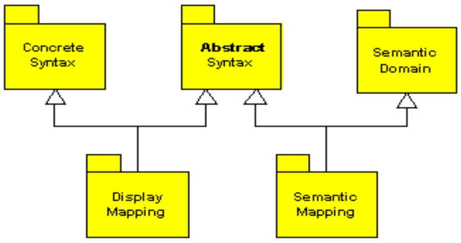
1.1 A Method for Meta-Modelling (MMM)

The UML is a collection of notations, some visual some textual. These notations currently have a loose mapping to an abstract syntax (which is imprecisely defined), which in turn is given an informal semantics written in natural language. The UML needs to become a precisely defined *family* of modelling languages, where a modelling language comprises a notation (concrete syntax), abstract syntax and semantics.

Software Engineers define languages as a collection of models with mappings between them. Typically a language consists of models for concrete syntax, abstract syntax and for the semantic domain. The MMF approach applies OO modelling to the definition of OO modelling languages. Each language component is defined as a package containing a class diagram. Package specialization is employed to support

© Springer-Verlag Berlin Heidelberg 2002

reusable, modular, incremental language design. OCL [20] [15] is used to define well-formedness constraints on the language components. Mappings between language components are defined in terms of OCL constraints on associations between model elements.



The MMF approach uses two key features of OO modelling technology: *package specialization* and *templates*. Package specialization permits (possible partial) definitions of model elements in a super-package to be consistently specialized in a sub-package. Templates are parametric model elements; supplying model elements as parameter values *stamps out* the template to produce a fresh model element. Templates provide a means of representing reusable modelling patterns; the MMF approach uses templates to capture patterns that occur repeatedly in OO modelling languages thereby providing a framework for defining language families.

This technology is not specific to MMF, UML has package specialization and parametric model elements and in particular the Catalysis approach [7] advocates the use of these features as part of an OO method. Algebraic specification languages such as Clear and OBJ and abstract programming languages such as ML and Haskell provide a means of constructing libraries of parametric components and organising systems by combining these components in different ways. However, MMF has provided the most precise definition of these concepts within the scope of OO modelling to date.

1.2 A Language for Meta-Modelling (MML)

MML is a static OO modelling language that aims to be small, meta-circular and as consistent as possible with UML 1.3. MML achieves parsimony by providing a small number of highly expressive orthogonal modelling features. The complete definition of MML is beyond the scope of this paper; the reader is directed to [2], [3] and [9] for an overview of the MMF approach, to [3] for the meta-circular definition of MML and to [5] and [6] for its formal definition. The rest of this section gives an overview of the main features of MML which are an OCL-like expression language; class definitions; package definitions and templates.

1.2.1 A Basic Expression Language

MML consists of a basic expression language which is based on OCL. The language provides a basic collection of data types including integers, booleans and strings together with standard operations over values of these types. MML supports sets and sequences together with a small number of standard OCL iteration constructs; the following denotes 5 (the full list of iteration constructs is defined in [6]):

```
    Set{1,2,3}->select(x | x > 1)->iterate(y n = 0 | n + y)
```

1.2.2 Class Definitions

MML classes define the structure, behaviour and invariants of their instances. The following defines a class of people.

```
class Person
  name : String; age : Integer; married : Boolean;
  children : Set(Person); parents : Set(Person);
  init(s:Seq(Instance)):Person
    self.name := s->at(0) []
    self.age := s->at(1) []
    self;
  averageChildAge():Integer
    self.children->iterate(c a=0 | a+c.age)/self.children->size;
  inv
    IfMarriedThenOver15 self.married implies self.age >= 16;
    OnlyTwoParents self.parents->size = 2

end
```

The definition of the class Person shows a number of MML features. In general, an MML definition consists of a name and an expression. A class definition introduces a new name whose scope is the class definition and relative to the package in which the class is defined using the '::' operator, for example SomePackage::Person.

A class has a number of attributes each of which is a definition consisting of a name and a type. A class definition has a number of method definitions each of which have typed parameters, a return type and a body. The body of a method is an expression which provides the return value when the method is called by sending an instance of the class a message. The init method of a class is automatically invoked when a new instance of the class is created. A class definition has a number of invariant constraint definitions following the keyword inv. Each constraint consists of a name and a boolean expression. The constraints express well formedness properties of the instances of the class. For example, in order to be married a person must be aged 16 or over.

1.2.3 Association Definitions

Classes may be associated to show logical dependency between instances of the classes. Currently MML supports only binary associations. A binary association consists of the two classes being associated, the name of the association and two association ends (one for each class). An association end is a definition consisting of a name and a multiplicity. The multiplicity constraint the number of instances of the attached class that can be associated with an instance of the class attached to the other end. For example, suppose that the children and parents attributes of the Person class were defined via an association (at most 2 parents, but any number of children):

```
association Family
  parents : Person mult: 2
  children : Person mult: *
end
```

1.2.4 Package Definitions

Packages are used in MML to group definitions of model elements. MML provides a powerful package specialization mechanism that allows packages to inherit from parent packages and to consistently specialize all of the inherited contents. For example:

```
package People
  class Person
     // as given above
  end;
  association Family
    // as given above
  end
end
```

Note that the association Family refers to the class Person as defined in the package People. Now, suppose that we want to extend the notion of being a person with an employer:

```
package Employment extends People
  class Person yearsInService : Integer end;
  class Company name : String end;
  association Works
    company : Company mult: 1
    employees : Person mult: *
  end
end
```

The package Employment extends the package People and therefore includes all of the definitions from People. A package is a *name space* and we may refer to two different classes called Person: People::Person and Employment::Person. Employment::Person contains all the definitions from People::Person extended with a new attribute named yearsInService. A package may only contain one definition with any given name. Therefore the association named Family in the package Employment must refer to the extended definition of Person. All definitions given by People have been *consistently extended* in Employment. The notion of consistent extension for model elements defined in a package is similar to the idea of *virtual methods* in C++. Package specialization supports multiple inheritance. Packages may be nested in which case the for package specialization outlined above hold for the nested packages.

1.2.5 Templates

A *template* is a parametric model element. When parameters are supplied to the template the result is a new model element. The supplied parameter values are model elements that are used by the template to construct, or *stamp out*, the new model element. Templates are used to capture patterns of recurring structure, behaviour and constraints that occur in models. Templates differ from specialization, which also captures patterns, in that there is no dependency between the template and the result of stamping it out. Specialization captures patterns in terms of (abstract) model elements that are specialized rather than stamped out. The process of specialization can lead to dependencies both between a super-model element and its sub-model elements and can also lead to sibling dependencies between different sub-model elements. Templates are not a replacement for specialization; they offer a new tool to the modeller that should be used where appropriate.

Suppose that we wish to capture the notion of containment. This involves two classes: a *container* and a *contained element*. Suppose also that all containers provide access to their contained elements via a method with the same name as the contained element class. Finally, suppose that we know all contained elements are named and that the container cannot contain two different elements with the same name. This can be expressed as a template in MML:

```
package Contains(Container,n1,m1,Contained,n2,m2)
  class <<Container>>
    <<n2>>():Set(<<Contained>>) self.<<n2>>
    inv
      <<"Every" + Contained + "HasADifferentName">>
        self.<<n2>>->forAll(c1 c2 | c1.name = c2.name implies c1 = c2)
  end;
  association <<Container + Contains>>
    <<n1>> : <<Container>> mult: <<m1>>
    <<n2>> : <<Contained>> mult: <<m2>>
  end
end
```

The package template Contains is defined to have six parameters. Container is the name of the container class, Contained is the name of the contained element class, n1 is the name used by an instance of the contained class to refer to its container and n2 is the name used by an instance of the container class to refer to its contents. The parameters m1 and m2 are the appropriate multiplicities for the containment. Throughout the body of the template definition literal names may be turned into expressions that are evaluated by enclosing them in << and >>. The names are supplied as strings and therefore the string concatenation operator + is used to construct new names. Suppose that we wish to express the containment relationship between a person and their children:

```
package People
 extends Container("Person","children",*,"Person","parents",2)
 class Person ...atribute and method definitions... end
end
```

Stamping out the container template produces a new package that can be used as the parent package of People. Defining the parents and children attributes this way has not saved much effort, however the template can be reused when defining the Employment package:

```
package Employment
 extends Companies, People,
  Container("Company","employees",*,"Person","employer",1)
end
```

1.3 A Tool for Meta-Modelling (MMT)

MMT is a prototype tool written in Java that supports the MMF approach. MMT consists of a virtual machine that runs the MML calculus which is a simple object-based calculus that supports higher order functions. All the MML examples contained in this paper are derived from MML code running on MMT (some slight simplifications have been applied). MMT defines MML by loading a collection of meta-circular boot files written in MML. The definitions in this paper have been loaded and checked in MMT which provides a flexible environment for inspecting and flattening definitions of packages and classes. A full description of MMT is outside the scope of this paper.

2 The Definition of a Simple Modelling Language

SML is a static modelling language that consists of packages and classes with attributes. Packages can contain both packages and classes. Classes contain attributes. An attribute has a name and a type. SML supports inheritance: packages may have super-packages, classes may have super-classes and attributes may have super-attributes. The meaning of SML package models is given by snapshots that contain objects. Each object is a container of slots which are named values. A package is a *classifier* for snapshots that contain sub-snapshots and objects corresponding to the packages and classes in the package. The structure of the syntax, semantic domain and semantic mapping for SML follows standard patterns that occur in modelling languages. The following sections show how these patterns can be captured as templates and then how SML can be defined by stamping out the templates.

2.1 Templates for SML Definition

2.1.1 Named Model Elements

Most modelling elements in SML are named. Like Java, MMT makes use of a toString method when displaying objects:

```
package Named(Model)
   class <<Model>>
     name : String;
     toString():String
       "<" + self.of.name + self.name + ">"
   end
end
```

2.1.2 Cloning Model Elements

Packages may have parents. A child package is defined to contain all the model elements defined by the parent package. A given model element is defined in a single name space; a package provides the name space for all of its elements. Therefore, when a model element is inherited from a parent package, the element must be copied and the containing name space must be updated to be the child package. The process of inheriting a copy of a model element and updating its containing name space is referred to as *cloning.* The cloning pattern occurs in two distinct stages: (1) a model element is shallow copied (no copying of slots) and the containing name space is updated; (2) the slots are copied.

```
package Clonable(Container,Contained)
   class <<Contained>>
     clone(nameSpace:<<Container>>):<<Container>>
       let o = self.copy()
           ms = self.of.allMethods()
           cs = ms->select(m | m.name = "cloneAux")
       in o.<<Container>> := nameSpace []
          cs->collect(m | (m.body)(o,nameSpace)) [] o
       end
   end
end
```

The Clonable template is defined above and is used to declare a clonable model element. The definition uses knowledge about the MML meta-level in order to copy an instance of the container class. Every object has a method named ‘copy’ that produces a shallow copy of the receiver. The template updates the value of the container to be the name space supplied to ‘clone’ and then invokes all of the methods defined by the container class named ‘cloneAux’. Each method will deal with copying the slots of the new object ‘o’.

2.1.3 Name Spaces

```
class <<Container>>
  <<"locallyDefines"+Contained>>(name:String):Boolean
    self.<<Contained+"s">>()->exists(m | m.name = name);
  <<"localLookup"+Contained>>(name:String):Set(<<Contained>>)
    self.<<Contained+"s">>()->select(m | m.name = name);
  <<"defines"+Contained>>(name:String):Boolean
    self.<<"all"+Contained+"s">>()->exists(m | m.name = name);
  <<"lookup"+Contained>>(name:String):<<Contained>>
    if self.<<"locallyDefines"+Contained>>(name)
    then self.<<"localLookup"+Contained>>(name).selectElement()
    else if self.<<"defines"+Contained>>(name)
         then self.<<"all"+Contained+"s">>()->select(m |
           m.name = name).selectElement()
         else state.error("NameSpace::lookup")
         endif
    endif
end
```

A *name space* is a container of named model elements that provides a protocol for accessing the elements by name. The template defined above is a simple notion of name space in which contained elements are assumed to own their own names. The template defines a name space lookup protocol involving local lookup and inherited lookup. The template therefore represents a mixin that requires the container to define a pair of methods for the contained elements that returns the local contains and the inherited contents.

2.1.4 Containers

```
package Contains(Container,Contained)
   class <<Container>>
     <<Contained + "s">>():Set(<<Contained>>)
       self.<<Contained + "s">>
     cloneAux(me:<<Container>>,nameSpace:<<Container>>)
       me.<<Contained + "s">> :=
         (me.<<Contained + "s">>()->collect(x |
           x.clone(nameSpace.<<"lookup" + Container>>(me.name))))
       end;
   association <<Container + Contained>>
     <<Container>> : Contains::<<Container>> mult: 1
     <<Contained + "s">> : Contains::<<Contained>> mult: *
   end
end
```

Many model elements in SML contain other model elements. The contains template defines a method for accessing the contained elements; providing method access allows the contained elements to be encapsulated. A variation of Contains is Self-Contains which has a single parameter. SelfContains is used to express model elements that can contain other model elements of the same type. A root self container contains itself; the method providing access to the contained elements of a self container removes the ‘self’ from the elements it returns (thereby satisfying the round trip constraint and also preventing cycles occurring when processing the contained elements).

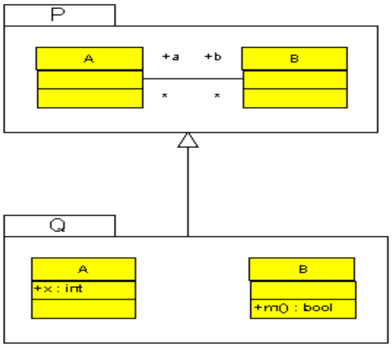
The template defines a method for cloning the contained elements when a container instance is cloned. The cloneAux method is supplied with the model element to clone (me) and the current name space (nameSpace) containing the model element. Each contained element is passed its name space by looking up the appropriate model element in nameSpace. In the absence of package specialization, the nameSpaces passed to model elements when they are cloned will be the appropriate copy of the original nameSpace container for the element. However, if a package is specialized, nameSpaces may be extended in which case the cloning mechanism will guarantee that the most specific definition is supplied to clone as the containing name space.

2.1.5 Specialization

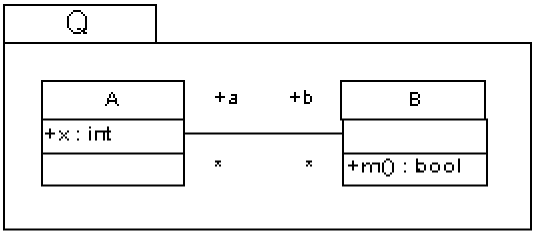
```
package Specializable(Model)
   class <<Model>>
     parents : Set(<<Model>>);
     allLocalParents() : Set(<<Model>>)
       self.parents->iterate(parent P = self.parents |
         P->union(parent.allLocalParents()))
   end
 end
```

In SML packages may be extended to contain new definitions; classes can be extended to contain new attributes, methods and constraints. Specialization may occur explicitly when the modeller defines a package to extend a super-package or defines a class to extend a super-class. Specialization may occur implicitly when the container of a model element m specializes another container that defines a model element m' such that m and m' have the same name. Every specializable model element must have a set of parents of the same type. The method allLocalParents is the transitive closure of the parents relation.

The contents of a container are defined by its parents: the local parents, as defined above, and any parents which are inherited from its own container:



Package P defines classes A and B and a binary association between them. The binary association has ends named a and b causing two attributes to be added to the classes at opposite ends of the association. Package Q defines two classes A and B with an attribute and an operation respectively. Package P is the parent of package Q. In order to compute the attributes of Q::A we must first compute its parents. A has no parents in Q but since the container of Q::A has parents we must inspect P in order to check whether it defines a class named A. We find it does and that P::A has an attributes named b. Therefore Q::A defines an attribute named b. The type of Q::A::b is a class called B which must be referenced with respect to the container of Q::A, namely Q. We find that Q defines Q::B and therefore the type of Q::A::b is Q::B. If we repeat this process for Q::B we find that Q::B defines Q::B::a whose type is Q::A. If we *flatten* the package inheritance the result is as follows:



A specializable container requires both the container and the contained model elements to be specializable. The complete set of parents for the contained model elements are defined by computing both the local parents (the transitive closure of the parents relation) and the inherited parents via the container. The contents of a container are computed with respect to *all* parents of the container. The template for specializable containers is:

```
package SpecializableContainer(Container,Contained)
   extends Specializable(Container),Specializable(Contained)
   class <<Container>>
     <<"all" + Contained + "s">>() : Set(<<Contained>>)
     self.allParents()->iterate(parent S = self.<<Contained+"s">>() |
       S->union(parent.<<"all"+Contained+"s">>()->reject(c |
         self.<<"locallyDefines + Contained>>(c.name))->collect(c |
           c.clone(self))))
      inv
       <<Contained + "sHaveDifferentNames">>
       self.<<"all" + Contained + "s">>()->forAll(c1 c2 |
         c1.name = c2.name implies c1 = c2)
   end;
   class <<Contained>>
     allParents() : Set(<<Contained>>)
       self.allLocalParents()->union(self.allInheritedParents());
     allInheritedParents() : Set(<<Contained>>)
       if self.<<Container>> = self
       then Set{}
       else self.<<Container>>.allParents()->iterate(parent S = Set{} |
         S->union(parent.<<"all"+Contained+"s">>()->select(m |
           m.name = self.name)))
       endif
   end
 end
```

All the contained elements of a specializable container are constructed as follows. Firstly all the parents of the container are constructed (recall that the parents of a model element will include both the locally defined parents and the parents inherited from the container's container). The locally defined contents are merged with the contents of all the parents after removing any parent contents that are shadowed locally. Finally, all inherited contents must be cloned in order that they are correctly contained.

2.1.6 Relations

```
package Relation(Name,Domain,Range)
   class <<Name>>
     left : <<Domain>>;
     right : <<Range>>
   end
end
```

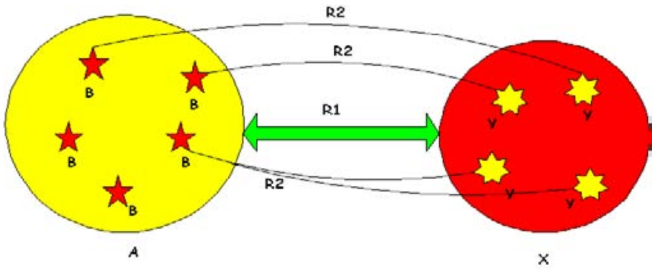
A relation has a name and holds between a class of domain elements and a class of range elements. A relation is essentially an *association class* that defines a constraint on pairs of domain and range instances.

2.1.7 Instantiation

A key feature of the MMF approach is the definition of modelling languages in terms of their abstract syntax and *semantic domain.* The abstract syntax is a model of the legal sentences of the language. The semantic domain is a model of the legal meanings that sentences can take. A language definition is completed by a model of the mapping between the abstract syntax and the semantic domain.

The relation between abstract syntax and semantic domain is referred to as *instantiation.* In general the instantiation relation between a model element and its instances may be aribitrary (expressions denote values, classes denote objects, state machines denote filmstrips, etc). However, if we know the structure of the abstract syntax and semantic domain then this places structure on the instantiation relationship. This structure can be expressed as templates.

Consider the following diagram:



ContainsInstances1(R1,A,B,R2,X,Y)

The diagram shows a typical instantiation relationship between two containers called ContainsInstances1. The instantiable model elements are shown on the left of the diagram and the instances are shown on the right. Elements of type A contain elements of type B and elements of type X contain elements of type Y. Elements of type A have instances of type X and elements of type B have instances of type Y. We wish to express the instantiation constraint that in order for an X to be classified as an instance of an A (R1) every Y that the X contains must be an instance of some B that the A contains (R2).

This form of instantiation relationship occurs between packages and snapshots where every object in the snapshot must be an instance of some class in the package, however not all classes need to be instantiated in the snapshot. This relationship is defined as a template:

```
package ContainsInstances1(
    R1,ModelContainer,ModelContained,
    R2,InstanceContainer,InstanceContained)
  extends
    Relation(R1,ModelContainer,InstanceContainer),
    Relation(R2,ModelContained,InstanceContained)
  class <<R1>>
```

```
   left : <<ModelContainer>>;
   right : <<InstanceContainer>>
   inv
     <<"InstancesOf"+ModelContainer+
             "ContainsInstancesOf"+ModelContained>>
     self.right.<<InstanceContained + "s">>()->forAll(i |
       self.left.<<"all" + ModelContained + "s">>()->exists(m |
         <<R2>>.new(Seq{m,i}).check() = Set{}))
  end
end
```

Other instantiation relationships are possible. For example, if we view slots as the instances of attributes and objects as the instances of classes then classes contain attributes and objects contain slots. An object is a well formed instance of a class when all the attributes have instances. This relationship can be defined as a template which we will call ContainsInstances2. Finally, there is an instantiation relationship which is defined as follows:

```
package ContainsInstances(R1,A,B,R2,X,Y)
  extends
    ContainsInstances1(R1,A,B,R2,X,Y),
    ContainsInstances2(R1,A,B,R2,X,Y)
end
```

2.1.8 Relationships between Attributes

```
package RelateAtt(R,Domain,Range,DomainAtt,RangeAtt,Pred)
   extends Relation(R,Domain,Range)
   class <<R>>
    inv
      <<"Relate"+Domain+"::"+DomainAtt+"To"+Range+"::"+RangeAtt>>
      Pred(self.left.<<DomainAtt>>,self.right.<<RangeAtt>>)
   end
end;
package SameName(R,Domain,Range)
   extends RelateAtt(R,Domain,Range,"name","name",=)
end;
package TypeCorrect(R,Domain,Range)
  extends RelateAtt(R,Domain,Range,"type","value",check)
end
```

An attribute relation involves a domain class and a range class. The relation specifies the domain and range attributes that are to be associated and also specified the predicate that will be used to check the values of the attributes. The invariant constraint in RelateAtt simply applies the predicate to the values of the slots in domain and range objects. SameName associates a domain and range object by requiring that they have the same values for the slot ‘name’. In SML this constraint is required when associating the attributes of a class with the slots of an instance of the class. TypeCorrect associates a domain class with an attribute named ‘type’ and a range class with an attribute named ‘value’. The predicate is satisfied when all the invariant constraints of the type return true for the value.

2.2 Definition of SML

We have described the MMF approach to language definition which is to model all components of the languages and to employ object-oriented techniques to achieve modularity and reuse. The previous section has used the novel technology of package specialization and templates to define a library of modelling language patterns. This section shows how the patterns can be used to construct a simple modelling language called SML.

2.2.1 Abstract Syntax

```
package AbstractSyntax
  extends
   SelfContains("Package"), SpecializableContainer("Package","Pack-
age"),
   SpecializableContainer("Package","Class"),
   SpecializableContainer("Class","Attribute"),
   Specializable("Attribute"),
   Contains("Package","Class"), Contains("Class","Attribute"),
   Clonable("Package","Class"), Clonable("Package","Package"),
   Clonable("Class","Attribute"),
   Named("Package"), Named("Class"), Named("Attribute"),
   NameSpace("Package","Package"), NameSpace("Package","Class"),
   NameSpace("Class","Attribute")
  class Attribute
   type : Class
   cloneAux(me:Attribute_,nameSpace:Class)
     me.type := (nameSpace.Package.lookupClass(me.type.name))
  end
end
```

The definition of the abstract syntax model for SML is given above. It is interesting to note that the MMF approach achieves a declarative specification of the model in terms of its properties explicitly listed in the ‘extends’ clause for the package. For example, we know that a package has the properties of a specializable container, that a package contains both packages and classes, and so on. If we were to define the abstract syntax as the result of flattening this definition, many of these properties would be implicit and therefore difficult to extract.

2.2.2 Semantic Domain

```
package SemanticDomain
  extends
    SelfContains("Snapshot"),
    Contains("Snapshot","Object"),Contains("Object","Slot"),
    Named("Snapshot"), Named("Slot")
  class Slot value : Object end
end
```

The domain is much simpler than the abstract syntax model. In our work using templates to define a UML 2.0 infrastructure we have a much richer semantic domain (for example, snapshots, objects and slots have parents). One of the benefits of the

MMF approach is that we can easily refactor the structure of a model in terms of its properties by adding new templates to the 'extends' clause of the package.

2.2.3 Semantic Mapping

```
package SemanticMapping
  extends
    AbstractSyntax, SemanticDomain,
    ContainsInstances1(
      "PackXSnap","Package","Class",
      "ClassXObj","Snapshot","Object"),
    ContainsInstances(
      "ClassXObj","Class","Attribute",
      "AttXSlot","Object","Slot"),
    SameName("AttXSlot","Attribute","Slot")
    TypeCorrect("AttXSlot","Attribute","Slot")
 end
```

The semantic mapping includes all of the elements from the abstract syntax and semantic domain and then constructs relations between them. For example, the relation PackXSnap is defined to check that every object contained in a snapshot is an instance of some class in the corresponding package.

3 Conclusion and Future Work

This paper has described the MMF approach to engineering Modelling Languages. The approach separates the issues of how to model syntax and semantics domains and allows languages to be developed from modular units. The approach also supports reusable patterns for language engineering. The paper has illustrated the approach with a small modelling language. MMF aims to provide coherent methods, technology and tools for engineering modelling languages. The core technology is not new, the methods for defining languages are well developed, the technology has its roots in Catalysis [7] and has been developed further in [5] and [8]. The novelty in MMF arises from bringing these otherwise disparate technologies together within a single consistent object-oriented framework. The MMF approach does not use a formal mathematical language to express the semantics of the languages; however, it is sufficiently expressive to support the infrastructure of these approaches and therefore can benefit from many of the results such as [1] and [17]. The MMT tool is still under development. Other tools exist, such as Argo and USE [16] [11] that can be used to model languages; however, unlike MMT, these tools tend to have a fixed meta-model.

We are applying the approach to the definition of rich and expressive visual modelling languages, such as [12] and [10]. In particular, the syntax employed in these diagrams is more sophisticated than that typically employed in UML. We are engaged in the UML 2.0 revision process [4] [19], and using MML ideas to help redefine aspects of UML with one of the main submission teams. The interested reader is directed to [4] which contains many examples of templates in a diagram format. But perhaps our most ambitious plans are in applying the MMF approach to realise the OMG MOdel Driven Architecture (MDA) initiative.

References

[1] Bottoni P., Koch M., Parisi-Presicce F., Taentzer G. (2000) Consistency Checking and Visualization of OCL Constraints. In Evans A., Kent S., Selic B. (eds) UML 2000 proceedings volume 1939 LNCS, 278 -- 293 , Springer-Verlag.

[2] Clark A., Evans A., Kent S. (2000) Profiles for Language Definition. Presented at the ECOOP pUML Workshop, Nice.

[3] Clark A., Evans A., Kent S, Cook S., Brodsky S., (2000) A feasibility Study in Rearchitecting UML as a Family of Languages Using a Precise OO Meta-Modeling Approach. Available at `http://www.puml.org/mmt.zip`.

[4] Clark A., Evans A., Kent S. (2001) Initial submission to the UML 2.0 Infrastructure RFP. Available at `http://www.cs.york.ac.uk/puml/papers/uml2submission.pdf`

[5] Clark A., Evans A., Kent S. (2001) The Specification of a Reference Implementation for UML. Special Issue of L'Objet on Object Modelling, 2001.

[6] Clark A., Evans A., Kent S. (2001) The Meta-Modeling Language Calculus: Foundation Semantics for UML. ETAPS FASE Conference 2001, Genoa.

[7] D'Souza D., Wills A. C. (1998) Object Components and Frameworks with UML -- The Catalysis Approach. Addison-Wesley.

[8] D'Souza D., Sane A., Birchenough A. (1999) First-Class Extensibility for UML - Packaging of Profiles, Stereotypes, Patterns. In France R. & Rumpe B. (eds) UML '99 proceedings volume 1723 LNCS, 265 -- 277, Springer-Verlag.

[9] Evans A., Kent S. (1999) Core meta-modelling semantics of UML -- The pUML approach. In France R. & Rumpe B. (eds) UML '99 proceedings volume 1723 LNCS, 140 -- 155, Springer-Verlag.

[10] Howse J., Molina F., Kent S., Taylor J. (1999) Reasoning with Spider Diagrams. Proceedings of the IEEE Symposium on Visual Languages '99, 138 -- 145. IEEE CS Press.

[11] Hussmann H., Demuth B., Finger F. (2000) Modular Architecture for a Toolset Supporting OCL In Evans A., Kent S., Selic B. (eds) UML 2000 proceedings volume 1939 LNCS, 278 -- 293 , Springer-Verlag.

[12] Kent S. (1997) Constraint Diagrams: Visualizing Invariants in Object-Oriented Models. In Proceedings of OOPSLA '97, 327 -- 341.

[13] UML 2.0 Infrastructure Request for Proposals, available from http://www.omg.org/uml

[14] The pUML Home Page `http://www.puml.org`.

[15] Richters M., Gogolla M. (1999) A metamodel for OCL. In France R. & Rumpe B. (eds) UML '99 proceedings volume 1723 LNCS, 156 -- 171, Springer-Verlag.

[16] Richters M., Gogolla M. (2000) Validating UML Models and OCL Constraints. In Evans A., Kent S., Selic B. (eds) UML 2000 proceedings volume 1939 LNCS, 265 -- 277, Springer-Verlag.

[17] Richters M., Gogolla M. (2000) A Semantics for OCL pre and post conditions. Presented at the OCL Workshop, UML 2000.

[18] Object Management Group (1999) OMG Unified Modeling Language Specification, version 1.3. Available at `http://www.omg.org/uml`.

[19] The UML 2.0 Working Group Home Page `http://www.celigent.com/omg/adptf/wgs/uml2wg.html`.

[20] Warmer J., Kleppe A. (1999) The Object Constraint Language: Precise Modeling with UML. Addison-Wesley.