Engineering Modelling Languages: A Precise Meta-Modelling Approach

Tony Clark[1], Andy Evans[2], Stuart Kent[3]

1.0 Introduction

This paper describes a Meta-Modeling Framework (MMF) that addresses many of the deficiencies in the definition of The unified Modeling Language (UML), and promises to support the OMG's newly announced strategy, Model Driven Architecture (MDA) [OMG01]. The facility comprises a language (MML) for defining modeling 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 ([pUML]) 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 [Cla01] which is expected to be completed in 2002.

The need for the definition of UML are recognized, in part, by the infrastructure RFP for UML 2.0 [OMG00]. In Section 3.0 on page 14 we identify some of the main requirements of the RFP, show how these relate to the needs described above, and outline how MMF addresses them.

This paper is structured as follows. The three main components (method, language and tool) to the MMF approach is described. The MMF approach is then applied to a simple description of a modelling language. The language, called SML, is a simplified version of a typical object-oriented static modelling language. MMF allows modelling language properties to be defined as a collection of templates. The templates for SML are defined and then used to construct models of syntax and semantics. Finally we place this work into context and outline future work.

1.1 A Method for Meta-Modelling (MMM)

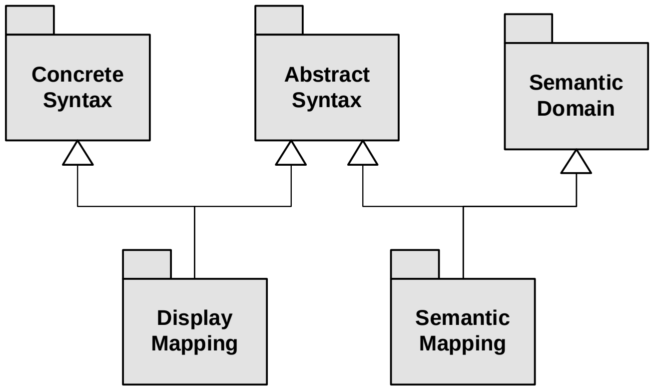
The UML is a collection of notations, some visual some textual. These notations currently have a loose mapping to an abstract syntax (which is imprecisely defined), which in turn is given an informal semantics written in natural language. The UML needs to become a precisely defined *family* of modeling languages, where a modeling 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. Mappings are defined by associating model elements. A language has mappings between concrete syntax and abstract syntax and between abstract syntax and the semantic domain. For example, a language for class diagrams can be defined in terms of a model for boxes and lines, a model for classes, attributes and associations, and a model for objects and slots. A mapping between abstract and concrete syntax relates classes, attributes and associations to boxes, text and lines. A mapping between abstract syntax and the semantic domain relates classes, attributes and associations to objects and slots.

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 reusable, modular, incremental language design. OCL [War99] [Ric99] is used to define well-

1. Department of Computer Science, King’s College London.
2. Department of Computer Science, University Of York.
3. Computing Laboratory, University of Kent.

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 [D'So98] 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 parameteric 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. MML is sufficiently expressive that it describes itself. This feature is not sufficient to guarantee that MML is unambiguous; however, it reduces the language to a handful of primitive semantic features that can be precisely captured by an external formal system.

The complete definition of MML is beyond the scope of this paper; the reader is directed to [Cla00a] , [Cla00b] and [Eva99] for an overview of the MMF approach, to [Clark01a] for the meta-circular definition of MML and to [Cla01b] and [Cla01c] 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. This language is not primitive to the approach which uses a much smaller MML calculus described in [Cla01c], however it encodes many of the abstractions which are both useful and familar to modellers.

The language provides a basic collection of data types including integers, booleans and strings together with standard operations over values of these types. Object slot reference and (synchronous) message passing is performed using the standard '.' operator. MML supports sets and sequences together with a small number of standard OCL interation constructs:

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

denotes 5. The full list of iteration constructs is defined in \cite{semantics}.

Since MML is a meta-circular language, types are values with state and operations. Objects are created by sending a ‘new’ message to a class together with sequence of iinitial slot values (the ‘init’ method of the class will deal with processing the initial values):

```
Person.new(Seq{“Fred”,28,true})
```

produces a new object of type Person named Fred, age 28 who is married.

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. Once the definition is evaluated the name is associated with the value of the expression for a given scope. The name is a definition is usually a literal (i.e. not an evaluated 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. The scope of an attribute name is relative to an instance of the class using the ‘.’ operator (slot reference) and relative to the class or one of its sub-classes using the ‘::’ operator. The scope of a class definition includes the body of the class (hence, children can be declared of type Person).

A class definition may include method definitions each of which have typed parameters, a return type and a body. The scope of a method name is relative to an instance of the class using the ‘.’ operator (message passing) and relative to a class or one of its sub-classes using the ‘::’ operator. 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[1] when a new instance of the class is created. All classes inherit from Object by default; Object provides a method called init which simply returns the receiver.

1. Since MML is a meta-language, meta-classes are reponsible for defining their own object creation protocol. The default protocol is provided by the class named Class in terms of a method called new that creates a new instance of the receiver and then sends the new instance an init message with the initialisation parameters.

A class definition may include invariant constraint definitions following the keyword inv. The scope fo the constraint name is relative to the class using the '::' operator. 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:

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

The multiplicity 2 requires all people to have two parents. The multiplicity permits all people to have any number of children (including 0).

An association introduces a number of implicit definitions. Each association end introduces an attribute definition to the attached class. Each multiplicity introduces an invariant constraint to the appropriate class. Both classes have a *round trip* invariant constraint that requires consistency when using the association to navigate from one class to the other and then back again.

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. For example, a basic model of companies could be factored out from Employment into a package called Companies . The package Employment would then be redefined to have two super-packages: People and Companies.

Packages may be nested in which case the for package specialization outlined above hold for the nested packages. For example, we may define a language as a package called L1 that contains nested packages named ConcreteSyntax, AbstractSyntax, SemanticDomain in addition to packages that define appropriate mappings. To define an extension of L1 we may define a package L2 that extends L1 and contains the apaporpriate extensions. The MML package specialization mechanism will ensure that all the models in L2 are correctly linked together.

1.2.5 Templates

A *template* is a parametric model element[1]. When parameters are supplied to the template the result is a new model element[2]. The supplied parameter values are model elements[3] 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
```

1. Currently our approach has made extensive use of template packages. There is no technical reason why this approach should not be applied to any model element.
2. Currently the results are *ground* model elements, i.e. not new templates. There is no technical reason why this approach should not allow a curried form of template instantiation.
3. This paper uses names as the parameter values. Our approach has used names up to this point, but we feel that more structured values will be required in general.

The package Contains is defined to have six parameters. Container is the name of the container class, Contained is the name of the contained elemnt 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 encluding them in << and >>. The names are supplied as strings and therefore the string concatenation operator + is used to construct new names.

Suppose that we wished 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
```

The Employment package has been completely defined by reusing models and template defined patterns.

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 simlple 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. MMT support a Swing-like graphics library in terms of machine primitives. A collection of libraries have been constructed that implement model editors, diagram viewers and a mechanism for performing all well-formedness checks on a given model.

2.0 The Definition of a Simple Modelling Language

MMF advocates a particular method for defining modelling languages. This approach is currently being applied to the definition of the modelling language UML and part of the UML 2.0 revision initiative. The approach leads to a collection of template libraries that capture reusable language properties. These libraries are then used to express UML as a family of languages.

The libraries of templates and the MMM approach are not limited to the definition of UML. This section shows how the approach can be used to construct a simple modelling language (SML). A small library of templates is constructed that capture the essential properties of the modelling language. SML is then defined by stamping out the templates. Due to space considerations we limit the definition of SML to the abstract syntax, the semantic domain and the mapping between the two.

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

2.1.1 Named Model Elements

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

TABLE 1. A template for naming model elements.

Most modelling elements in SML are named. Like Java, MMT makes use of a toString method when displaying objects. Note that there is no reason why nameable model elements could not inherit from an abstract class, however Named is an example of a simple template.

2.1.2 Cloning Model Elements

```
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
```

TABLE 2. A template for clonable 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.

Figure 2 on page 7 shows the definition of a template Clonable that can be 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 shallpw 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

```
package NameSpace(Container,Contained)
  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
end
```

TABLE 3. A template for name spaces.

A *name space* is a container of named model elements that provides a protocol for accessing the elements by name. There are a variety of increasingly sophisticated approaches to organising name spaces; options include access modes such as private, public and protected definitions in Java and include the decision relating to name ownership (should the contained model element own its name or should the container own all the names of contained elements).

Figure 3 on page 8 shows a template that defines 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
```

TABLE 4. A template for container model elements.

Many model elements in SML contain other model elements. Figure 4 on page 8 defines a simple containment template. 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 SelfContains 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

Specialization in modelling languages is a relationship between model elements. Specialization occurs in many different variations, for example *inheritance* is usually used to mean the ability to extend existing definitions with new features; *conformance* requires instances of one class to be substitutable for instances of another class in some or all contexts. We claim that variations of specialization can be captured as a collection of patterns that are combined in a number of different ways. It is beyond the scope of this paper to deal with the issue of specialization in depth, however we give a collection of templates for a simple version of inheritance in SML.

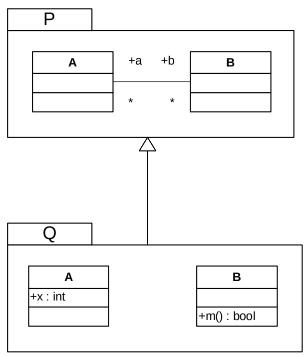
In SML packages may be extended to contain new definitions; classes can be extended to contain new attributes, methods and constraints. Specialization may occur explicitly when the modeller defines a package to extend a super-package or defines a class to extend a super-class. Specialization may occur

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

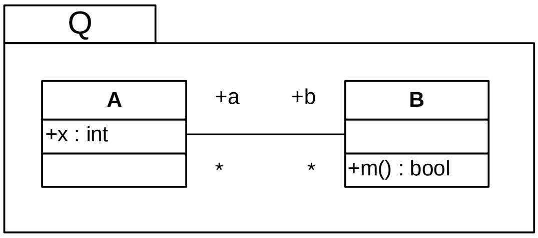
TABLE 5. A template for specializable model elements.

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. Figure 5 on page 9 defines a template that captures the basic notion of a specializable model element. 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 parents of a container are the local parents, as defined above, any any parents which are inherited from its own container. The following diagram shows how this works:



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 specializa-

```
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
```

TABLE 6. A template for specializable containers.

ble. 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 defined in Figure 6 on page 10.

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.

The container class invariant requires all the contents of a container to have different names. This cinstraint forces multiple definitions with the same name to be merged. The rules for merging can differ quite widely between modelling languages and between different modelling elements in the same language. Merging is therefore left open in SML.

2.1.6 Relations

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

TABLE 7. A template for general purpose relations between model elements.

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. Figure 7 on page 11 defines a template for general purpose relations.

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 sentances 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 Figure 8 on page 11 which shows a typical instantation 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).

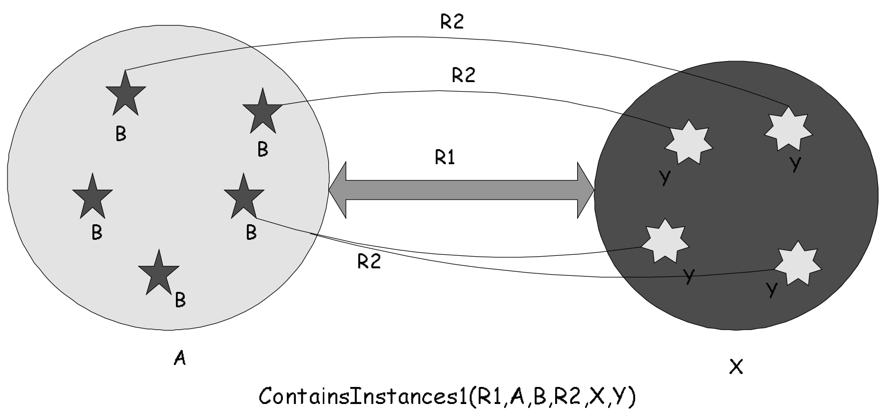


TABLE 8. Instantiable Containment

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.

```
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
```

TABLE 9. A template for instantiable containers.

Figure 9 on page 12 shows this relationship defined as a template. 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
```

TABLE 10. A template for relating attributes and a specialization for names.

Figure 10 on page 12 shows the definition of a template for relating attributes between models and two specializations of the template. 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 requireing 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","Package"),
  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
```

TABLE 11. The SML Abstract Syntax Model

Figure 11 on page 13 shows the definition of the abstract syntax model for SML. 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
```

TABLE 12. The SML Semantic Domain Model

Figure 12 on page 14 shows the semantic domain for SML. This 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
```

TABLE 13. The SML Semantic Domain Model

Figure 13 on page 14 shows the semantic mapping for SML. 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.

3.0 MMF, UML 2.0 and MDA

This section considers in more detail how elements of MMF address issues of the current UML 2.0 revision and the newly-announced OMG MDA strategy. The main impact of MMF ideas is in the area of UML infrastructure, as distilled in the UML 2.0 infrastructure RFP. The RFP requires that the UML meta-model be restructured to separate kernel constructs from the standard elements that depend on them, that the meta-model be organised as a collection of packages, that the meta-model enfore a separation of concerns between notation and semantics with mappings between them, and that the kernel should support profiles. We believe that the MMF approach addresses all of these requirements.

In a wider context, MMF tackles a number of issues essential to realise the Model Driven Development (MDD) dream, which has at least one incarnation in the new OMG strategy of Model Driven Architecture (MDA) [OMG01]. An interesting vision of MDD/MDA has been presented by Desmond D'Souza [D’So01].

4.0 Conclusion and Future Work

This paper has described the MMF approach to engineering ModelingLanguages. The approach separates the issues of how to modelsyntax and semantics domains and allows languages to be developedfrom modular units. The approach also supports reusable patternsfor language engineering. The paper has illustrated the approachwith a small modeling language which is then extended in twodifferent ways: a static extension and a dynamic extension. The potential application of the MMF approach to the revision of UMLand to realizing Model Driven Architecture (MDA) has been discussed.

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 techology has its roots in Catalysis [D'So98] and has been developed further in [Cl01d] and [D'So99]. 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 toexpress 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 [Bot00] and [Ric00b]. The MMTtool is still under development and has its roots in OO meta-programming theory and systems such as Smalltalk, CLOS and the ObjVLisp model; the consequence of this is that the tool is very flexible. Other tools exist, such as Argo and USE [Ric00a] [Hus00] that can be used to model languages; however these tools tend to have a fixed meta-model.

Further work is focussing on a number of areas. We are applying the approach to the definition of rich and expressive visual modeling languages, such as [Ken97] and [How99]. 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, and using MML ideas to help redefine aspects of UML with one of the main submission teams. But perhaps our most ambitious plans are in applying the MMF approach to realise MDA. For example, we are looking at the use of MMF to define mappings from precise OO specifications of e-Business systems down onto their realisation on particular combinations of implementation technologies. In a similar context, we are looking at Model Driven Testing. We are exploring the use of packages to encode modeling patterns in specific application areas, as well as encoding different implementation strategies as patterns at the meta-modelling level.

5.0 Bibliography

OMG00 UML 2.0 Infrastructure Request for Proposals, available from http://www.omg.org/uml

OMG01 The OMG (2001) Executive Overview: Model Driven Architecture. Available from http://www.omg.org/mda/

Bot00 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 The Unified Modeling Language -- Advancing the Standard. Third International Conference. York, UK 2000. Proceedings volume 1939 LNCS, 278 -- 293 , Springer-Verlag.

Cla00a Clark A., Evans A., Kent S. (2000) Profiles for Language Definition. Presented at the ECOOP pUML Workshop, Nice.

Cla00b 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.

Cla01a 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

Cla01b Clark A., Evans A., Kent S. (2000) The Specification of a Reference Impl mentation for UML. Special Issue of L'Objet on Object Modelling, 2001.

Cla01c Clark A., Evans A., Kent S. (2000) The Meta-Modeling Language Calculus: Foundation Semantics for UML. ETAPS FASE Conference 2001, Genoa.

Cla01d Clarke S., Walker R. J. (2001) Composition Patterns: An Approach to Designing Reusable Aspects, in Proceedings of ICSE'2001, May 2001.

D'So98 D'Souza D., Wills A. C. (1998) Object Components and Frameworks with UML -- The Catalysis Approach. Addison-Wesley.

D'So99 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 The Unified Modeling Language -- Beyond the Standard. Second International Conference. Fort Collins CO, USA. 1999. Proceedings volume 1723 LNCS, 265 -- 277, Springer-Verlag.

D'So01 D'Souza D. (2001) Model Driven Architecture and Integration. Available from http://www.catalysis.org/omg/

Eva99 Evans A., Kent S. (1999) Core meta-modelling semantics of UML -- The pUML approach. In France R. \& Rumpe B. (eds) UML '99 The Unified Modeling Language -- Beyond the Standard. Second International Conference. Fort Collins CO, USA. 1999. Proceedings volume 1723 LNCS, 140 -- 155, Springer-Verlag.

How99 Howse J., Molina F., Kent S., Taylor J. (1999) Reasoning with Spider Di grams. Proceedings of the IEEE Symposium on Visual Languages '99, 138 -- 145. IEEE CS Press.

Hus00 Hussmann H., Demuth B., Finger F. (2000) Modular Architecture for a Toolset Supporting OCL In Evans A., Kent S., Selic B. (eds) UML 2000 The Unified Modeling Language -- Advancing the Standard. Third International Conference. York, UK 2000. Proceedings volume 1939 LNCS, 278 -- 293 , Springer-Verlag.

Ken97 Kent S. (1997) Constraint Diagrams: Visualizing Invariants in Object-Oriented Models. In Proceedings of OOPSLA '97, 327 -- 341.

UML1.3 Object Management Group (1999) OMG Unified Modeling Language Specification, version 1.3. Available at http://www.omg.org/uml.

Ric99 Richters M., Gogolla M. (1999) A metamodel for OCL. In France R. \& Rumpe B. (eds) UML '99 The Unified Modeling Language -- Beyond the Standard. Second International Conference. Fort Collins CO, USA. 1999. Proceedings volume 1723 LNCS, 156 -- 171, Springer-Verlag.

Ric00a Richters M., Gogolla M. (2000) Validating UML Models and OCL Constraints. In Evans A., Kent S., Selic B. (eds) UML 2000 The Unified Modeling Language -- Advancing the Standard. Third International Conference. York, UK 2000. Proceedings volume 1939 LNCS, 265 -- 277, Springer-Verlag.

Ric00b Richters M., Gogolla M. (2000) A Semantics for OCL pre and post conditions. Presented at the OCL Workshop, UML 2000.

War99 Warmer J., Kleppe A. (1999) The Object Constraint Language: Precise Modeling with UML. Addison-Wesley.

UML20 The UML 2.0 Working Group Home Page http://www.celigent.com/omg/adptf/wgs/uml2wg.html.

pUML The pUML Home Page http://www.puml.org.