The MMF Approach to Engineering Object-Oriented Design Languages

Tony Clark[1],
Andy Evans[2],
Stuart Kent[3],
Paul Sammut[4]

1: King's College London
anclark@dcs.kcl.ac.uk
2: University of York
andye@cs.york.ac.uk
3: University of Kent at Canterbury
stuart@mclellankent.com
4: University of York
pauls@cs.york.ac.uk

Abstract

The Unified Modeling Language (UML) is a family of languages used to express features of software systems. MMF is a framework designed to allow UML-like notations to be constructed from modular language definitions each of which expresses models for concrete syntax, abstract syntax and semantic domains. MMF consists of a meta-modelling language (MML) and a tool (MMT) that implements MML as a meta-programming language. MMF can be used to design application specific UML profiles that facilitate model analysis or automatic implementation. This paper provides an introduction to MMF through the development of a series of simple language definitions.

1. Introduction

The Unified Modeling Language [12] is a standardized notation for expressing object-oriented software systems. It is essentially a family of extensible modelling notations. The current UML definition lacks a number of desirable features that are currently being addressed through a co-ordinated effort to define a new version (UML 2.0 [18]). These features include enhancing the modularity and extensibility of UML and addressing the notion of UML semantics.

Modularity is important for UML because it is a very large notation, currently organized as a loose confederation of modelling languages. Each language addresses different aspects (some overlapping) of a given system. Module composition and extension facilities are essential due to the large number of modules involved. UML 1.3 provides modularity via packages which are name-spaces and containers. The current definition of packages does not provide facilities for composing and extending packages.

Semantics is a key weakness of the current UML 1.3 definition. Semantics is necessary to clearly answer questions relating to model ambiguity and to facilitate tool compatibility. The current semantics definition is weak in that it uses a semi-formal notation (The Object Constraint Language [16]) to define syntactic well-formedness conditions and uses natural language for the rest.

If we accept that UML is a family of languages then we can learn from the Computer Science community that have been engineering languages for the last 40 years. Among the many possible

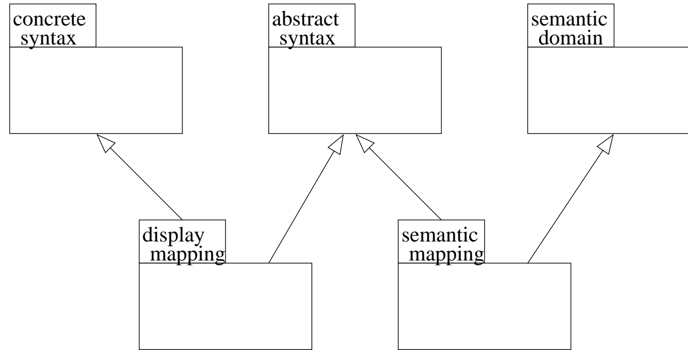

Figure 1: The MMF Method

approaches to language engineering there is a common feature: define the symbols used for denoting things, define the things to be denoted and define a mapping between them. We will refer to this style of semantics as *denotational* semantics.

A denotational semantics for UML will provide the key to unambiguous analysis of UML models. Such analysis is essential when verifying that implementations or refinements are correct and when verifying that tools implement UML correctly. Such a semantics will allow the inter-relationships between the different UML sub-languages to be analyzed; since the approach separates syntax and semantics, it will be possible to replace or tailor the concrete syntax of a UML sub-language without changing the essential meaning.

This paper describes the Meta-Modelling Facility (MMF) that consists of a denotational method for defining OO modelling notations, a language (MML) for expressing modelling notations and a tool (MMT) for implementing modelling notations. The development of MMF by the pUML group ([19]) is ongoing and has been supported by IBM.

The paper is structured as follows. Section 2 gives an overview of MMF. Section 3 gives an example of how a simple OO design notation can be modelled and subsequently extended using MMF. Section 4 describes the current state of MMF and the directions for future work.

2. The Meta-Modelling Facility (MMF)

2.1. The Method

MMF aims to provide a modular and extensible method for developing object-oriented design languages. Modularity is provided by *packages* that act as name-spaces and can be used to define reusable patterns. Each component of a language definition is given as a separate package; the packages are then combined to produce a complete definition.

The common language components are shown in figure 1. Each component is defined as a package. The syntax domain is defined in terms of concrete syntax (a human-centric representation) and in terms of abstract syntax (a computer-centric representation). The semantic domain is defined as a separate model. The display mapping links the abstract syntax to the concrete syntax; this mapping could be refined to produce a parsing algorithm or to produce well-formedness rules. The semantic mapping links the abstract syntax to the semantic domain. The semantic mapping gives the modelling language a meaning. Other mappings are possible, for example a mapping that provides a concrete syntax for the semantic domain; whilst MMF supports these mappings we limit ourselves to those shown for the purposes of this paper.

Extensibility is provided by the usual notion of class-based inheritance and a new notation for extending packages based on that of Catalysis [7]. Package extension allows new language components to be based on existing components, thereby allowing languages to be built incrementally. The package extension mechanism also supports the development of meta-patterns that guide the language developer in terms of common design issues and *variation points* whereby generic designs can be tailored to suit particular modelling domains. Examples of package extension and patterns are given in the next section.

2.2. The Language

MML is a language that supports the MMF method. 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 [3] [6] for an overview of the MMF approach, to [2] for the meta-circular definition of MML and to [4] and [5] for its formal definition.

2.2.1. Patterns

Figure 2 provides three examples of MML patterns that we will subsequently use in an example of MMF language engineering. A pattern is a package whose contents can be specialized and renamed when defining new packages. Since a pattern is just a package, patterns can be specialized or combined to produce new patterns.

The `container` pattern describes the essential features of modelling elements that contain things. A specialization of the `container` pattern may rename the classes `Container` and `Contained` and may rename the attribute `elements`. The renamings must be consistent with the pattern so that the source and target of the renamed attribute is the renamed container and contained classes respectively.

The `inheritance` pattern describes the essential features of modelling elements that can inherit features. Examples of inherited features are attributes and methods of classes. The `Inheriting` class has a number of parents and a number of features. The method `allParents` computes the transitive closure of the `parents` relation. The method `allFeatures` is used to calculate the complete set of inherited features for a class with inheritance. The attribute `features` and the method `allFeatures` may be renamed more than once for each inheritable feature of a modelling element when the pattern is specialized.

The `inheritance` pattern specializes the `container` pattern. This is shown by placing a specialization arrow between the patterns. The child pattern specializes all (and therefore contains all) of the contents of the parent pattern. A pattern specialization may be annotated with renamings. A class, attribute or method in the pattern may be renamed. The resulting child component is a specialization of the corresponding modelling element in the pattern. When an attribute is specialized, its multiplicity may change, for example from `*` (zero or many) to `1` (exactly one).

In addition to graphical language features for packages, classes and attributes, MML provides the Object Constraint Language (OCL). OCL is used to express constraints and queries on models. A query is an operation that has no side effect, for example generating the set of all parents or the inherited features of an inheriting modelling element:

```
inheritance.Inheriting::allParents():Set(Inheriting)
  self.parents->iterate(p ps = Set{} |
    p.allParents()->including(p))
```

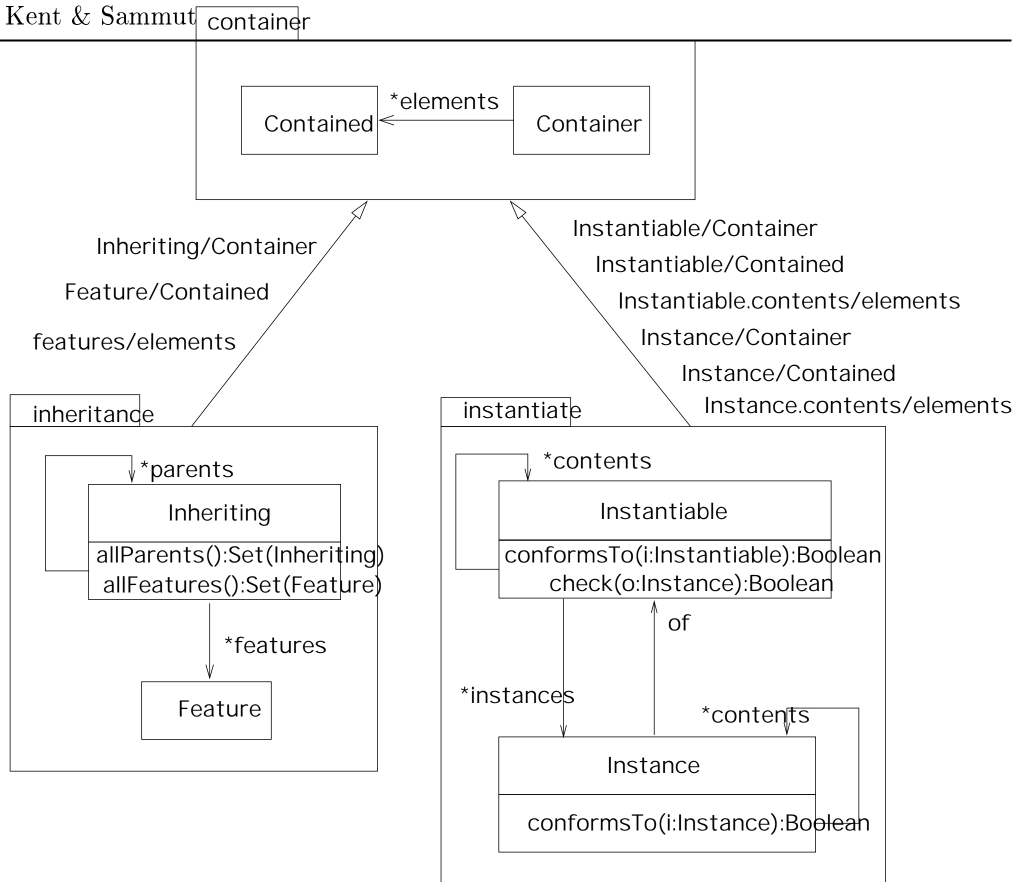

Figure 2: MML Patterns

```
inheritance.Inheriting::allFeatures():Set(Feature)
  self.allParents()->including(self)->iterate(i fs = Set{} |
    fs->union(i.features))
```

The `instantiate` pattern describes the essential features of a modelling element that has instances. This pattern is used to define the semantic mapping for MMF languages. Each `Instantiable` modelling element has a number of `instances`.

Definition 1 *Each instance of an instantiable element must satisfy the* `check` *method of the instantiable element:*

```
instantiate.Instantiable inv:
  self.instances->forAll(i | self.check(i))
```

Instantiable elements contain sub-instantiable elements; this pattern is repeated for instances.

Definition 2 *The instance relationship is monotonic with respect to the containment relationship:*

```
instantiate.Instantiable inv:
  self.instances->forAll(i |
```

```
    i.contents->forAll(i' |
      self.allContents()->exists(c |
        c.instances->includes(i')))) and
  self.allContents()->forAll(c |
    c.instances->forAll(i |
      self.instances->exists(i' |
        i'.contents->contains(i))))
```

One instance is said to *conform to* another instance if it upholds the principle of substitutability. The notion of substitutability depends on the type of instance and is defined on a case-by-case basis.

Definition 3 *The definition of instance conformance is monotonic with respect to containment:*

```
instantiate.Instance::conformsTo(i:Instance):Boolean
  self.contents->forAll(i' |
    i.contents->exists(i'' |
      i'.conformsTo(i'')))
```

One instantiable modelling element conforms to another when their respective instance sets are in the conformance relationship:

```
instantiate.Instantiable::conformsTo(i:Instantiable):Boolean
  self.instances->forAll(instance |
    i.instances->exists(instance' |
      instance.conformsTo(instance'))
```

2.2.2. Classifiers

A fundamental feature of MML models is the idea of a *classifier*. A classifier is a modelling element that is associated with instances that it classifies using OCL constraints. For example, a class is a classifier and is associated with its objects. In general a classifier contains a collection of features that are used to determine its set of legal instances. A classifier has parent classifiers whose features it inherits. A classifier is therefore both a container (of features) and a generalizable component (inheritance).

Figure 3 shows a simplified definition of MML classifiers as an example of how packages and package specialization support reusable patterns. The package `classifier` defines two classes `ModelElement` and `Classifier`. A model element is simply a named component in a model.

2.3. The Tool

The MMF method and language is supported by a prototype tool called MMT. The tool provides programmatic support for OO modelling language engineering. MMT is a small object-based virtual machine implemented in Java. MML is translated to a very small kernel language that runs on the machine. In addition to OCL, the kernel language can support object side-effects and can create new objects. A new object is created when an expression of the following form is evaluated:

```
@ClassName
  slotName = slotValue;
  slotName = slotValue;
  ...
end
```

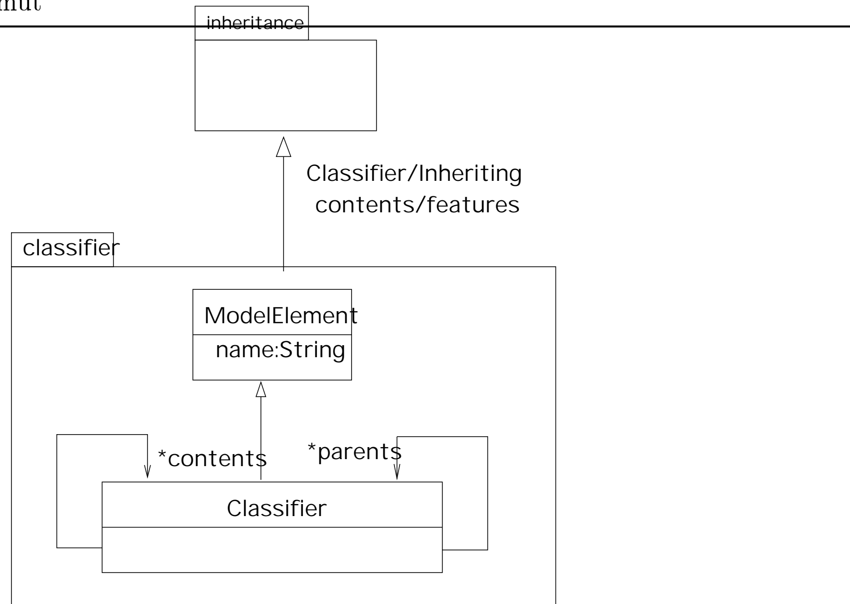

Figure 3: Simple Classifiers

MMT supports packages, classes, attributes and methods. Currently, MMT supports package, class and method specialization. Method specializations must have the same name and may refer to the specialized definition using the special OCL extension `super.run()`.

3. Engineering a Design Notation

This section gives an example of how MML can be used to precise define and subsequently extend an OO design notation. A basic notation for expressing static properties of a system (a small sub-set of UML class diagrams) is defined in section 3.1 using the MMF method. Section 3.2 shows how the MMF approach can be used to extend the basic modelling language to support states. Finally, the language with states is extended with state transition machines in section 3.3 that place constraints on the dynamic behaviour of objects in a system.

3.1. A Simple Modelling Language

This section defines a basic notation for modelling the static structure of a system. MMF requires us to model the abstract syntax, the semantic domain and the concrete syntax for each modelling language. These are defined in sections 3.1.1, 3.1.2 and 3.1.4 respectively. The mappings between abstract syntax and the semantic domain and the concrete syntax and abstract syntax are defined in sections 3.1.3 and 3.1.5 respectively.

3.1.1. Abstract Syntax

The essential features of a static modelling notation are defined by the meta-model in figure 4. A package is a classifier that contains classes. A class is a classifier that contains attributes. An attribute is a classifier that contains a single type.

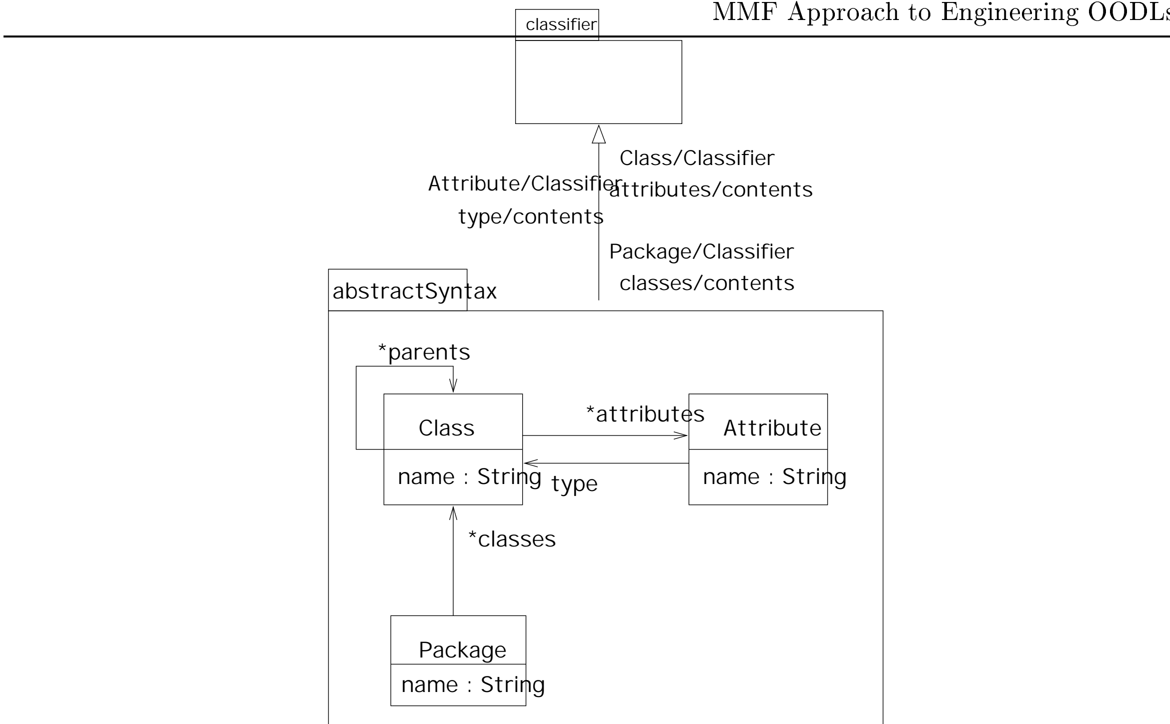

Figure 4: Abstract Syntax

3.1.2. Semantic Domain

Figure 5 defines the semantic model used to give a meaning to the basic modelling notation in the previous section. Objects have slots; each slot has a name and a value. Objects are grouped together into package instances. The value of a slot can be found using the method `slotValue`:

```
semanticDomain.Object::slotValue(name:String):Object
  self.slots->select(s | s.name = name)->asSequence->first.value
```

3.1.3. Semantic Mapping

The meaning of the basic modelling language is given as a family of relations defined in figure 6. The semantic mapping package is a specialization of the `instantiate` pattern which requires each type of modelling component is associated with a semantic component that represents its *instances*. In addition the `instantiate` pattern requires that the semantic mapping package provides appropriate definitions for the `check` and `conformsTo` methods.

Definition 4 *Every instance of an attribute is a slot whose name is the same as that of the attribute and whose value is an instance of the type of the attribute.*

```
semanticMapping.Attribute::check(s:Slot):Boolean
  self.name = s.name and
  self.type.check(s.value)
```

Every instance of a class must have slots that correspond to the class attributes.

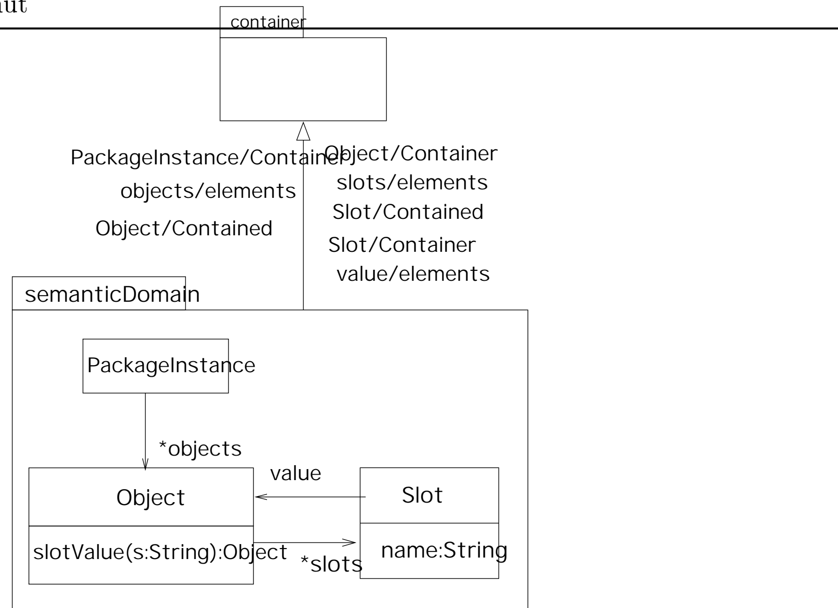

Figure 5: Semantic Domain

```
semanticMapping.Class::check(o:Object):Boolean
  self.allAttributes()->forAll(a |
    o.slots->exists(s | a.check(o)))
```

Every instance of a package is a package instance that contains instances of the appropriate classes:

```
semanticMapping.Package::check(p:PackageInstance):Boolean
  self.allClasses()->exists(c |
    c.check(o))
```

In addition to defining how to check instances, `instantiate` requires a definition of semantic conformance for each specialization of `Instance`. A slot is specialized by specializing its value:

```
semanticMapping.Slot::conformsTo(s:Slot):Boolean
  super.run() and
  self.name = s.name and
  self.value.conformsTo(s.value)
```

Theorem 1 *Every instance of a class has slots corresponding to the attributes that the class and its super-classes.*

```
semanticMapping.Class inv:
  self.instances->forAll(o |
    o.slots->forAll(s |
      self.allAttributes()->exists(a |
        a.name = s.name and
        a.type.check(s.value))))
```

Proof

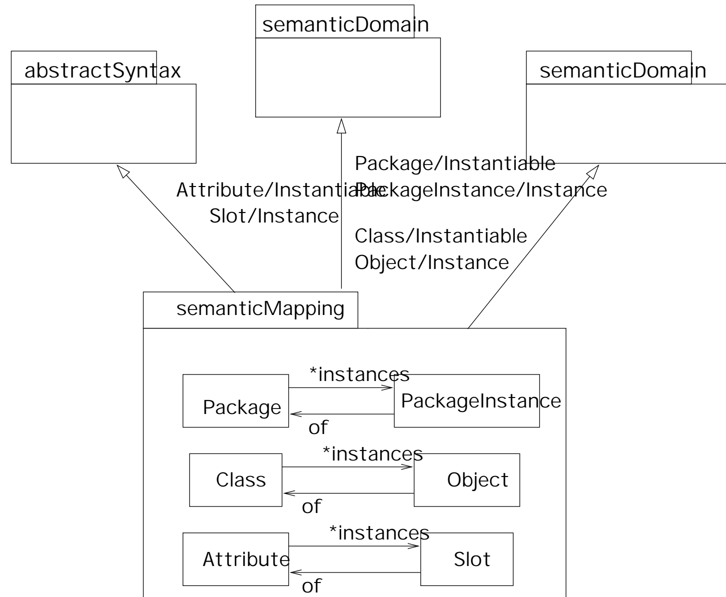

Figure 6: The Semantics of the Basic Modelling Language

1. We start with the constraint given in definition 2:

```
instance.Instantiable inv:
  self.instances->forAll(i |
    i.contents->forAll(i' |
      self.allContents()->exists(c |
        c.instances->includes(i'))))
```

2. The renamings given in figure 6 gives the following:

```
semanticMapping.Class inv:
  self.instances->forAll(i |
    i.slots->forAll(s |
      self.allAttributes()->exists(a |
        a.instances->includes(s))))
```

3. From definition 1 we get:

```
semanticMapping.Class inv:
  self.instances->forAll(i |
    i.slots->forAll(s |
      self.allAttributes()->exists(a |
        a.check(s)
```

4. Finally, from definition 4 we get:

```
semanticMapping.Class inv:
  self.instances->forAll(i |
```

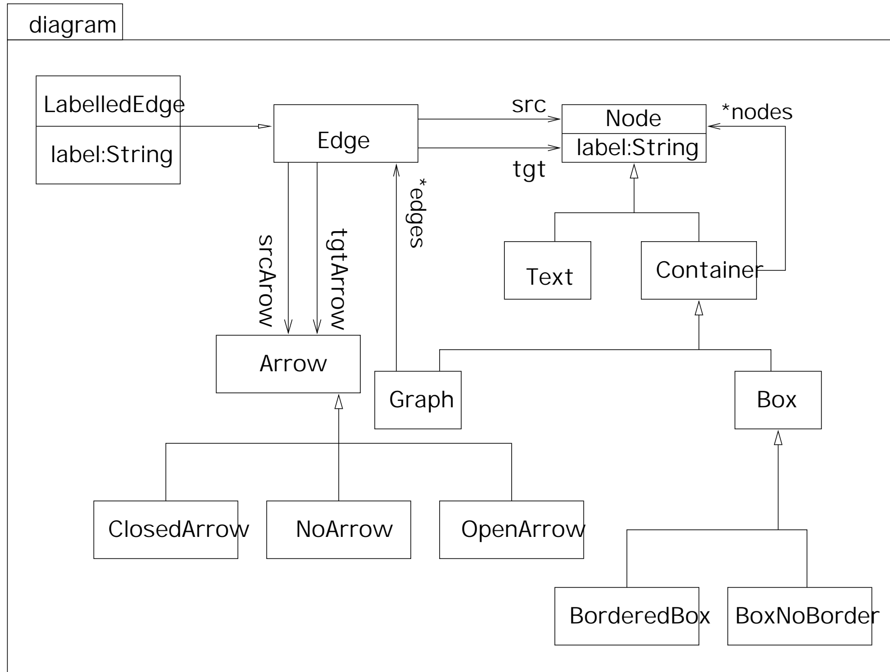

Figure 7: Concrete Syntax

```
i.slots->forAll(s |
  self.allAttributes()->exists(a |
    a.name = s.name and
    a.type.check(s.value)
```

QED

3.1.4. Concrete Syntax

The concrete syntax of a static model is a graph where the nodes are nested boxes containing text and where edges are either generalizations (unlabelled with a closed arrow at the target end) or attributes (labelled and with an open arrow at the target end). Figure 7 defines a model for the concrete syntax of our basic modelling notation.

3.1.5. Syntax Mapping

A package can generate its concrete syntax using the method `draw` defined by extending `Package` and `Class` as shown in the figure 8. The result is a graph where the nodes are produced by requesting each class to generate its concrete syntax and the edges are produced by generating the appropriate generalization and attribute arrows.

A class draws itself as a box containing text fields. The uppermost text field contains the name of the class. The attributes of the class whose types are basic (Integer, String, Boolean, *etc.*) are listed in text fields below the name:

```
display.Class.draw():BorderedBox
```

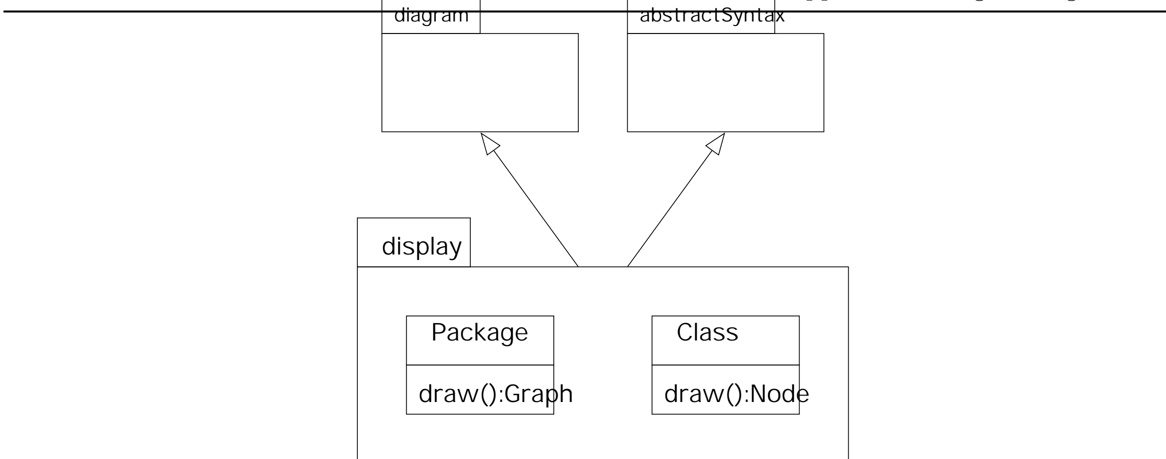

Figure 8: Basic Drawing

```
  @BorderedBox
    label(_) = self.name;
    nodes(_) = Seq{
      @BorderedBox
        label = "name";
        nodes(_) = Seq{@Text label(_) = self.name end}
      end,
      @BorderedBox
        label = "attributes";
        nodes(_) =
          self.attributes->select(a |
            self.basicType(a.type))->collect(att |
              @Text label = att.name+":"+att.type.name end)->asSequence
      end}
  end
```

A package is drawn as a complete graph:

```
display.Package::draw():Graph
  @Graph
    nodes(_) = self.classes->collect(c | c.draw());
    edges(g) = self.gens(g.nodes)->union(self.atts(g.nodes));
  end;
```

Generalization arrows are produced by creating an edge with a closed arrow for each super-class of each class. The method `findNode` is used to index the appropriate node given a class name:

```
display.Package::gens(nodes:Set(Node)):Set(Edge)
  self.classes->iterate(c edges = Set{} |
    edges->union(c.supers->collect(c' |
      @Edge
        src(_) = self.findNode(nodes,c.name);
        tgt(_) = self.findNode(nodes,c'.name);
        srcArrow = @NoArrow end;
```

```
        tgtArrow = @ClosedArrow end
      end)));
```

Attribute arrows are produced by finding all the attributes of each class whose target type is non-basic (i.e. not Integer, String, Boolean etc.) and creating a labelled edge with an open arrow at the target end:

```
display.Package::atts(nodes:Set(Node)):Set(Edge)
  self.classes->iterate(c edges = Set{} |
    edges->union(c.attributes->reject(a |
      self.basicType(a.type))->collect(a |
        @LabelledEdge
          label = a.name;
          src(_) = self.findNode(nodes,c.name);
          tgt(_) = self.findNode(nodes,a.type.name);
          srcArrow = @NoArrow end;
          tgtArrow = @OpenArrow end
        end)));
```

3.2. A Language with States

MMF allows new modelling languages to be defined as extensions of existing languages. The new language must define the appropriate components: abstract syntax; concrete syntax; semantic domain; and mappings. The components of the new language may extend corresponding components of existing language definitions.

This section extends the basic static modelling language define in section 3.1 with *states*. Each class defines a number of states. An instance of a *class with states* must be in one of the states. Section 3.2.1 defines the abstract syntax; the semantic domain for the new language is the same as that defined in section 3.1.2; the extended semantic relation is defined in section 3.2.2; the concrete syntax domain is the same as defined in section 3.1.4; section 3.2.3 defines the relationship between the extended abstract syntax and the concrete syntax.

3.2.1. Abstract Syntax

The states of a class are listed in the class definition as state names. States are inherited. The extended abstract syntax model is shown in figure 9. States are classifiers and will therefore have instances; however, states do not have any internal structure:

```
stateModel.State inv:
  contents = Set{}
```

3.2.2. Semantic Mapping

The meaning of the extended modelling language is defined by a mapping to the semantic domain defined in figure 3.1.2 *i.e.* no new semantic features are required. The semantic mapping is shown in figure 10. The states defined by a class are the names of boolean values slots in instances of the class. Since an instance must be in exactly one state at any given time, one of the state slots must be `true` and all the other state slots must be `false`:

```
stateMapping.State::check(s:Slot):Boolean
  s.name = self.name and
  Boolean.check(s.value)
```

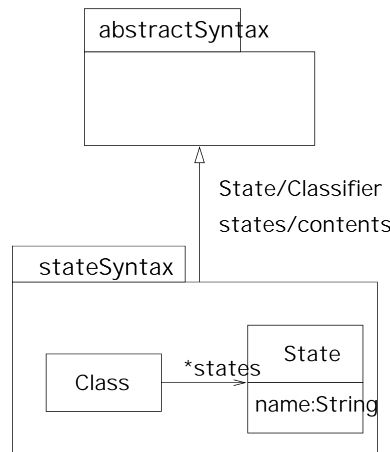

Figure 9: Abstract Syntax Extended for States

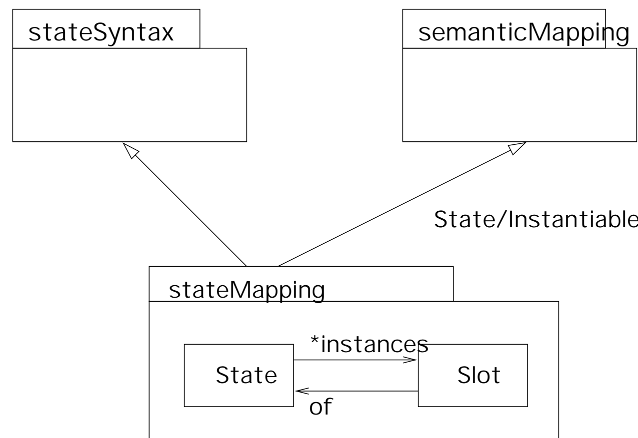

Figure 10: Extended Semantic Domain

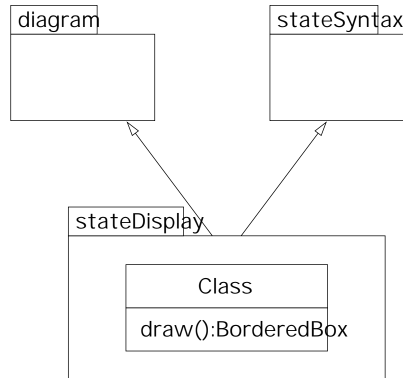

Figure 11: Concrete Syntax for the Language with States

It follows from the pattern `classifier` that classes with states have instances with state slots. This constraint is specialized further in orfer to require exactly one of the slots to be true:

```
stateMapping.Class::check(o:Object):Boolean
  super.run() and
  self.allStates()->iterate(s b = false |
    o.slotValue(state.name) xor b)
```

3.2.3. Syntax Mapping

The concrete syntax domain for class diagrams containing state information is the same as that defined in figure 7. The display mapping is defined in figure 11. The state information is displayed as a sequence of state names in a *state compartment* of a class:

```
stateDisplay.Class::draw():BorderedBox
  @BorderedBox
    label(_) = self.name;
    nodes(_) = Seq{
      super.run(),
      @Text
        label(_) = self.states->iterate(state s = "< " |
          s + state + " ") + ">"
      end}
  end
```

3.3. A Language with Dynamic Features

This section extends the static language with states defined in section 3.2 with dynamic features. A state transition machine is often used in modelling notations to place restrictions on the legal sequence of states occurring in object life-cycles. Section 3.3.1 defines dynamic extensions to the abstract syntax; section 3.3.2 defines dynamic extensions to the semantic domain; section 3.3.3 defines dynamic extensions to the semantic mapping; finally, section 3.3.4 defines dynamic extensions to the concrete mapping.

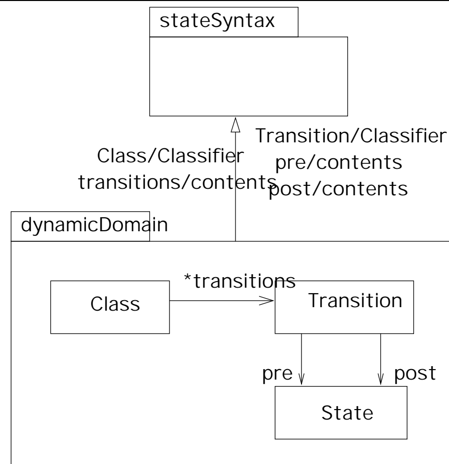

Figure 12: A Simple Dynamic Modelling Language

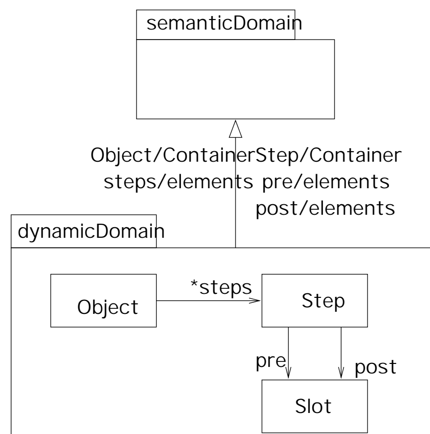

Figure 13: A Dynamic Semantic Domain

3.3.1. Abstract Syntax

Figure 12 defines the modelling components necessary to express simple constraints on the sequence of states occurring in an object life-cycle. A class contains a collection of state transitions; each transition corresponds to a legal state change that can occur in the life-cycle of an instance of the class.

3.3.2. Semantic Domain

Figure 13 defines the dynamic semantic domain. Each object contains its life-cycle represented as a collection of steps. Each step records a state change of the object. In order to be well formed, an object's life-cycle must start at the object's current state and form a chain. We will assume that we are dealing with finite life-cycles. This leads to the following invariant:

```
dynamicDomain.Object inv:
```

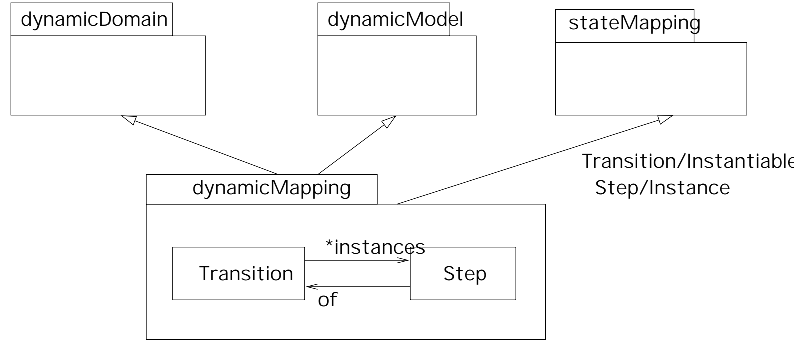

Figure 14: A Dynamic Semantics

```
self.steps->exists(first |
  self.slotValue(first.pre.name) = true and
  self.steps->forAll(s |
    s = first or
    self.steps->exists(s' |
      s'.post = s.pre and s' <> first))) and
self.steps->exists(last |
  self.steps->forAll(s |
    s = last or
    self.steps->exists(s' |
      s.post = s'.pre and s' <> last)))
```

3.3.3. Semantic Mapping

Figure 14 shows the semantic mapping for the dynamic modelling language.

Theorem 2 *The instances of transitions are steps where the pre- and post-states of the step are instances of the corresponding pre- and post-condition of the transition.*

```
dynamicMapping.Transition.check(s:Step):Boolean
  self.pre.check(s.pre) and
  self.post.check(s.post)
```

Proof
The proof is indentical for both the pre and post states. The following is a proof for the pre-state:

1. From definition 2 we get:

   ```
   instantiate.Instantiable inv:
     self.instances->forAll(i |
       i.contents->forAll(i' |
         self.allContents()->exists(c |
           c.instances->includes(i'))))
   ```

2. The renamings leading to figure 14 produces:

   ```
   dynamicMapping.Transition inv:
   ```

```
self.instances->forAll(step |
  step.pre->forAll(slot |
    self.allPre()->exists(state |
      state.instances->includes(slot))))
```

3. We know from the definition of `Transition` and `Step` that the multiplicity of pre-states and pre-conditions are both 1, therefore by definition 1:

   ```
   dynamicMapping.Transition inv:
     self.instances->forAll(step |
       step.pre.check(step.pre))
   ```

4. Finally, if the previous step holds for all instances then:

   ```
   dynamicMapping.Transition.check(step:Step)
     self.pre.check(step)
   ```

   QED

Theorem 3 *The life-cycles of class instances must be instances of the appropriate state transition machine.*

```
dynamicMapping.Class::check(o:Object):Boolean
  self.trans->forAll(t |
    o.steps->exists(s |
      t.check(s)))
```

Theorem 4 *Two steps conform when the corresponding pre and post state instances conform.*

```
dynamicMapping.Step::conformsTo(s:Step):Boolean
  self.pre.conformsTo(s.pre) and
  self.post.conformsTo(s.post)
```

Proof
We show just the `pre` side of the proof:

1. From definition 3 we get:

   ```
   instantiate.Instance::conformsTo(i:Instance):Boolean
     self.contents->forAll(i' |
       i.contents->exists(i'' |
         i'.conformsTo(i'')))
   ```

2. The renaming leading to figure 14 produces:

   ```
   dynamicMapping.Step::conformsTo(s:Step):Boolean
     self.pre->forAll(slot |
       s.pre->exists(slot' |
         slot.conformsTo(slot')))
   ```

3. Since the multiplicity of `pre` is specialized to 1:

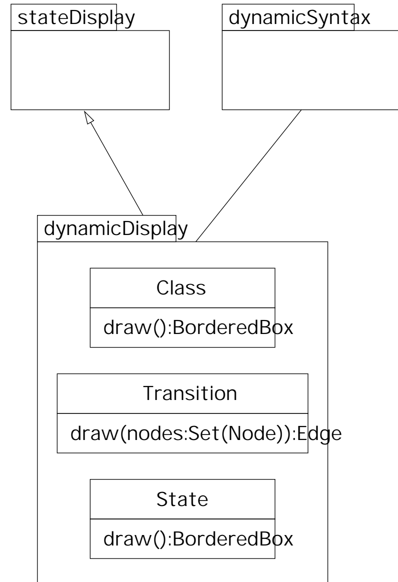

Figure 15: Concrete Syntax for a Dynamic Language

```
dynamicMapping.Step::conformsTo(s:Step):Boolean
  self.pre.conformsTo(s.pre)
```

QED

Theorem 5 *Two objects conform when their slots, states and steps conform.*

```
dynamicMapping.Object::conformsTo(o:Object):Boolean
  self.steps->forAll(s |
    o.steps->exists(s' |
      s.conformsTo(s')))
```

3.3.4. Syntax Mapping

Figure 15 shows a display mapping for the dynamic language. The state transition machine for a class is displayed as a graph inside a new class compartment. The nodes of the graph are the names of the states and the edges show the permissable state transitions.

```
dynamicDisplay.Class::draw():BorderedBox
  @BorderedBox
    label(_) = self.name;
    nodes(_) = Seq{
      super.run(),
      @BorderedBox
        label() = 'transition machine';
        nodes(_) =
          @Graph
            nodes(_) = self.states->forAll(s | s.draw());
            edges(_) = self.trans->forAll(t | t.draw(self.nodes))
```

```
        end
      end}
  end

dynamicDrawing.Transition::draw(nodes:Set(Node)):Edge
  @Edge
    src(_) = findNode(self.pre.name,nodes);
    tgt(_) = findNode(self.post.name,nodes);
    srcArrow(_) = @NoArrow end;
    tgtArrow(_) = @OpenArrow end
  end

dynamicDrawing.State::draw():BorderedBox
  @BorderedBox
    label(_) = self.name;
    nodes(_) = @Text label(_) = self.name end
  end
```

4. Conclusion

This paper has described the MMF approach to engineering Object-Oriented 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 very small modelling language which is then extended in two different ways: a static extension and a dynamic extension. The examples have shows how the approach supports precise analysis of the modelling languages by constructing proofs of language properties.

The MMF approach to engineering OO modelling languages is proposed as an approach that can support the UML 2.0 development process. As such it is consistent with the current definition of UML 1.3 and is a development of the approach reported in [8] and [13]. The approach will support extensions to both the syntax and semantics of UML, for example [9] and [11]. 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 [15]. The MMT tool [17] 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 [14] [10] that can be used to model languages; however these tools tend to have a fixed meta-model.

MML is currently being used as part of the UML 2.0 revision process. In addition it is being used to develop application specific profiles. We plan to include implementation languages as part of the profile library development, for example including profiles for Java. Since MMF can precisely support and integrate a wide variety of different languages, a very interesting area for deployment is eBusiness, where the choice of individual implementation mechanisms depends on key semantic features of the models.

Bibliography

[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 The Unified

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

[2] Brodsky S., Clark A., Cook S., Evans A., Kent 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.

[3] Clark A., Evans A., France R., Kent S., Rumpe B. (1999) Response to UML 2.0 Request for Information. Available at http://www.puml.org/papers/RFIReponse.PDF.

[4] Clark A., Evans A., Kent S. (2000) The Specification of a Reference Implementation for UML. Accepted for publication in a Special Issue of L'Objet in 2000.

[5] Clark A., Evans A., Kent S. (2000) The Meta-Modeling Language Calculus: Foundation Semantics for UML. To be presented at the ETAPS FASE Conference 2001.

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

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

[8] 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.

[9] 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.

[10] 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.

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

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

[13] 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.

[14] 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.

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

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

[17] The MMT Tool. Available at http://www.puml.org/mmt.zip.

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

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