Patterns for Renaming and Stamping Out Object-Oriented Models

Tony Clark
King's College London, anclark@dcs.kcl.ac.uk

Andy Evans
University of York, andye@cs.york.ac.uk

Stuart Kent
University of Kent, stuart@mclellankent.com

Abstract

Modern system engineering is supported by a families of modelling languages; each member of a specific family addresses a different aspect of the application domain. Constructing families of modelling languages is facilitated by the use of packages and templates. Packages are containers of modelling elements. Packages may be specialised and merged. Renamings may be applied to packages. Templates are parameteric packages. New packages are constructed by supplying arguments to a template. Packages and templates rely on the following underlying technology: specialisation; merging; renaming and stamping. Specialisation and merging have been described elsewhere; this paper defines mechanisms for renaming and stamping.

Keywords
UML, object-oriented, modelling, templates.

I. INTRODUCTION

Modern system engineering is supported by a families of modelling languages; each member of a specific family addresses a different aspect of the application domain. The initial stages of system specification define a modelling architecture suitable for the application domain and the development method. The architecture identifies the different aspects of the system that must be developed and the relationships between aspects. Modelling languages that support each aspect, including the inter-relationships, are developed to allow modular system development. The overall architecture and the inter-relationships act as a specification for the composition of the modular components thereby ensuring that the system and its development method will compose to give the correct behaviour.

Constructing families of modelling languages is facilitated by the use of packages and templates. Packages are containers of modelling elements. Packages may be specialised and merged. Renamings may be applied to packages. Templates are parameteric packages. New packages are constructed by supplying arguments to a template. Packages and templates rely on the following underlying technology: specialisation; merging; renaming and stamping.

This paper addresses these issues by defining a model of the supporting technology for system development viewed as a collection of interrelated languages. The cornerstone technology is a model of relationships that is used to define renaming, template stamping and syntax resolution. In order to keep the exposition simple and self contained the features are applied to themselves; therefore all the key components are used to define a simple modelling language that supports templates, renaming and resolution. We do not address inheritance and specialisation since these have been covered elsewhere; however, the relational approach could be used to define these in a similar way to the examples given in the paper.

II. MML

The results described in this paper have been produced as part of the 2U submission to the UML 2.0 revision initiative. 2U is a consortium of acadmic researchers and industrial software practitioners that aim to take a precise language engineering approach to the revision of UML 1.x. The work has been supported by IBM [2] and Rational Inc. An initial feasibility study [2] shows that the approach can be applied to produce well structured modelling languages.

The approach is based on a number of novel technologies and a tool that allows models to be constructed and checked. The key technologies are package specialisation and templates. Package specialisation allows different aspects of the same components to be modelled in different packages; the packages are merged by constructing a single package that inherits all of the partial super-packages thereby merging all of the partial views. Templates allow language patterns to be constructed and then stamped out as described in this paper. Package specialisation and templates are based on the concepts described in Catalysis [7] related work on patterns and templates is described in [6].

The 2U technologies are implemented in a tool called MMT that runs a language for constructing models called MML. MML is a generalisation of the Object Constraint Language (OCL); it adds features for defining classes, packages, package specialisation and templates. The semantics of OCL is analysed in [9] and [10]. MMT is a meta-modelling environment that allows models to be analysed and checked. This paper expresses research results using MML; the examples have been implemented and checked in MMT. MML is described in more detail in [3], [4] and [5].

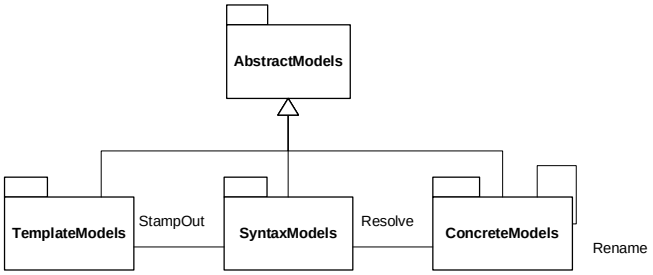
The UML 2.0 revision initiative is scheduled to be completed in August 2002. More details of all proposals can be

found at the OMG web site: www.omg.org and specific details of the 2U approach can be found at www.2uworks.org.

III. Relations Between Modelling Languages

A modelling language consists of a collection of related sub-languages. The sub-language relationships can be stamped out using relationship templates. This paper defines a model of relationships and some standard templates that can be used to construct useful relations.

In order to motivate the definitions we use a simple modelling language defined as a collection of related simple sub-languages. The overall architecture of the language is given as follows:



AbstractModels defines modelling features that are common to the sub-languages. Each sub-language extends AbstractModels. AbstractModels defined packages, classes and attributes. TemplateModels extends AbstractModels by defining string expressions for the names of modelling elements and the types of attributes. SyntaxModels extends AbstractModels by defining strings as the name of modelling elements and the types of attributes. ConcreteModels extends AbstractModels by adding strings for modelling element names and classes as the types of attributes.

The figure shows relationships between the sub-languages. TemplateModels are *stamped out* by supplying values for the variables in the string expressions. The structure of the template bodies is retained by StampOut and string expressions are replaced by strings. SyntaxModels are *resolved* by replacing the strings in attribute types with the appropriate class in the containing package. ConcreteModels can have elements *renamed* in which case the structure is preserved whilst the names of the modelling elements may change.

IV. A Simple Modelling Language

Object-oriented modelling languages, such as UML, provide notations that express static and dynamic system features. A common static feature of such languages provides modelling elements for classes and attributes. Classes that address a common aspect of a system are grouped together into packages.

Modelling languages differ with respect to the semantics of packages, classes and attributes; however, a number of common patterns recur. The containership pattern allows one type of modelling element to structurally contain another. The containership pattern is expressed as follows:

```
package Contains(X,Y)
  class <<X>>
    <<Y+"s">> : Set(<<Y>>);
  end
  class <<Y>>
  end
end
```

Another recurring pattern allows modelling elements to be named. The feature that names a modelling element is generally a string of characters; however, templates will use string expressions to name elements so the template allows the type of names to be varied:

```
package Named(X,Y)
  class <<X>>
    name : <<Y>>;
  end
end
```

Attributes in models are typed. The *type of* attribute types varies depending on whether the attribute occurs in a template (a string expression), a syntax model (a string) or a concrete model (a class). The following template is used to type an attribute:

```
package Typed(X,Y)
  class <<X>>
    type : <<Y>>
  end
end
```

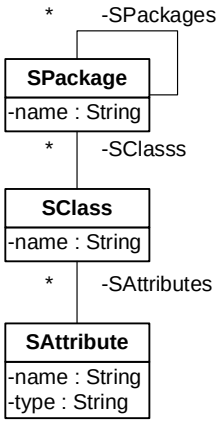
We develop different modelling languages for templates, syntax models and concrete models. These languages differ with respect to element naming and attribute typing; however they have a common intersection defined as a package of abstract models:

```
package AbstractModels
  extends
    Contains(Package,Package),
    Contains(Package,Class),
    Contains(Class,Attribute)
end
```

Syntax models use strings for the names of model elements and use strings for the types of attributes:

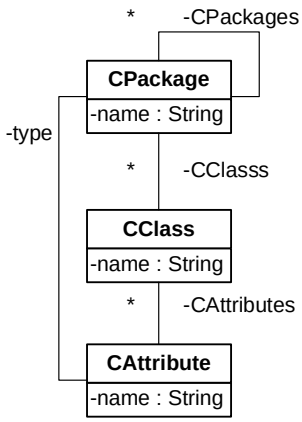
```
package SyntaxModels
  extends
    AbstractModels[
      SPackage/Package,
      SClass/Class,
      SAttribute/Attribute],
    Named(Package,String),
    Named(Class,String),
    Named(Attribute,String),
    Typed(Attribute,String)
```

```
end
```



Concrete models (or *resolved syntax* models) are defined by extending AbstractModels with string names for each type of modelling element and classes for attribute types. The abstract modelling elements are renamed:

```
package ConcreteModels
  extends
    AbstractModels[
      CPackage/Package,
      CClass/Class,
      CAttribute/Attribute],
    Named(CPackage,String),
    Named(CClass,String),
    Named(CAttribute,String),
    Named(CAttribute,CClass)
end
```



V. Simple Templates

Templates are parametric packages where the names of elements in a package are expressed using a simple string language. A string expression is either a string constant, a string valued variable or a concatenation of two strings. When the template is *stamped out* the package is copied and the string expressions are evaluated to produce string names for the new modelling elements.

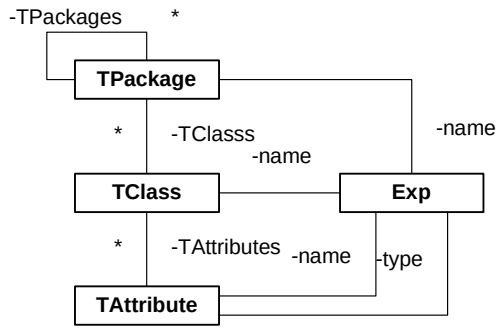
String expressions are defined by the following package:

```
package Expressions
  class Exp end
  class Const extends Exp
    value : String;
    init(s:Seq(Instance)):Object
      self.value := (s->at(0)) []
      self
    end
    eval(env:Env):String
      self.value
   end
  end
  class Var extends Exp
    name : String;
    eval(env:Env):String
      env.lookup(self.name)
   end
  end
  class Concat extends Exp
    left : Exp;
    right : Exp;
    eval(env:Env):String
      self.left.eval(env) + self.right.eval(env)
    end
  end
end
```

Each concrete sub-class of Exp defines a method named ‘eval’ that is used to evaluate the expression in the context of some variable values. The context is supplied as a value of type Env (not defined here) that associated strings (variable names) with strings (their values).

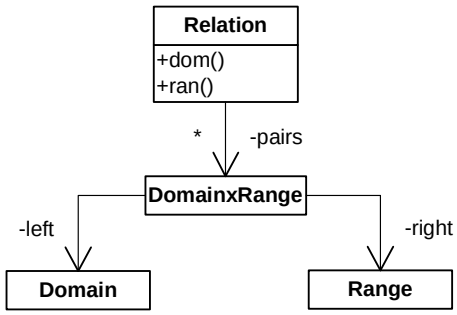
The bodies of templates are expressed in the language defined by the package TemplateModels:

```
package TemplateModels
  extends
    Expressions,
    AbstractModels[
      TPackage/Package,
      TClass/Class,
      TAttribute/Attribute],
    Named(TPackage,Exp),
    Named(TClass,Exp),
    Named(TAttribute,Exp),
    Type(TAttribute,Exp)
end
```



VI. Relations

A relation consists of a domain and a range and a set of pairs. For the purposes of this paper the domain and range are classes. Each pair has a *left* and a *right* component. The left component is an instance of the domain and the right component is an instance of the range:



A relation is a pattern that can be captured as a template:

```
package Relation(N,D,R)
   class <<N>>
      pairs : Set(<<D + "x" + R>>);
      dom():Set(<<D>>)
         self.pairs->collect(p | p.left)
      end
      ran():Set(<<R>>)
         self.pairs->collect(p | p.right)
      end
   end
   class <<D + "x" + R>>
      left : <<D>>;
      right : <<R>>;
   end
   class <<D>> end
   class <<R>> end
end
```

Relations can be expressed between any pair of modelling element types by extending an appropriate stamping of the Relation template:

```
   package Free
      extends
         ConcreteModels,
         Relation(P<->P,CPackage,CPackage),
         Relation(C<->C,CClass,CClass),
         Relation(A<->A,CAttribute,CAttribute)
   end
```

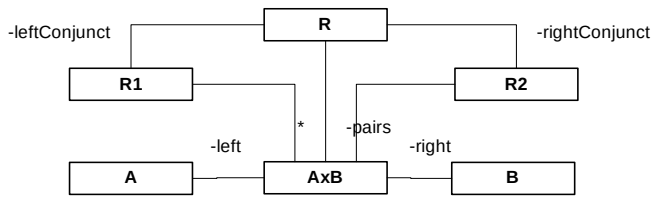
The package Free defines three relations on packages, classes and attributes respectively. Since no further constraints are expressed, these relations are *free* in the sense that they may pair up any element of the domain with any element of the range. In particular, the names of the domain elements may be different from the names of the range elements. Furthermore, there is no requirement for the relations to be interdependent, even though packages contain classes and classes contain attributes.

VII. Relational Conjuncts and Disjuncts

Relational combinators are templates that are used to construct composite relations. The templates take relations as arguments and combine them in different ways to produce a new relation. The following binary relational combinator builds a new relation that satisfies all the constraints on both the argument relations:

```
package And(N,N1,N2,D,R)
   extends
      Relation(N1,R,D),
      Relation(N2,R,D),
      Relation(N,R,D)
   class N
      leftConjunct : <<R1>>;
      rightConjunct : <<R2>>;
      inv
         <<"Pairs satisfy both " + R1 + " and " + R2>>
         self.pairs->forAll(p |
            self.leftConjuct.pairs->includes(p) and
            self.rightConjuct.pairs->includes(p))
         fail: "Illegal conjunct."
      end
   end
end
```

The relation And(R,R1,R2,A,B) is the following model:



Similarly, relation disjunction is defined by the following template:

```
package Or(N,N1,N2,D,R)
   extends
      Relation(N1,R,D),
      Relation(N2,R,D),
```

```
    Relation(N,R,D)
  class N
    leftDisjunct : <<R1>>;
    rightDisjunct : <<R2>>;
    inv
      <<"Pairs satisfy either" + R1 + " or" + R2>>
      self.pairs->forAll(p |
        self.leftDisjunct.pairs->includes(p) or
        self.rightDisjunct.pairs->includes(p))
      fail: "Illegal disjunct."
    end
  end
end
```

VIII. Preserving Names

Concrete model elements are named. A renaming operator is applied to a package and will change the names of some elements in the package whilst preserving others. The preservation of names can be expressed as a constraint on a relation. The constraint may be placed on any relation in which the domain and range elements are typed, therefore this pattern can be expressed as a template:

```
package SameName(N,D,R,T)
  extends
    Named(D,T),
    Named(R,T)
  class <<N>>
    inv
      <<D + " and " + R + " have same name">>
        self.pairs->forAll(p | p.left.name = p.right.name)
      fail: "Elements must have the same name."
    end
  end
end
```

The template SameName can be used to force the names of packages, classes and attributes to remain the same:

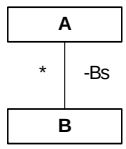
```
package PreserveCNames
  extends
    Free,
    SameName(P<->P,CPackage,CPackage,String),
    SameName(C<->C,CClass,CClass,String),
    SameName(A<->A,CAttribute,CAttribute,String)
end
```

The package PreserveCNames now forces each relation to contain pairs where the name of the domain element is the same string as the name of the range element. However, the relations do not force containment to be preserved, for example if package P contains class C in the domain of P<->P then there is no requirement for P to contain C in the range of P<->P.

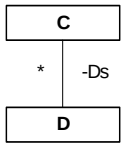
IX. Preserving Containment

Relations over domains that are constructed using the Contains template will typically (but not always, consider nested packages that are flattened into a single top-level container) preserve containment in the range. For example, when stamping out or renaming the containment pattern between packages and classes is preserved. Consider the case of two languages expressed using the containment pattern:

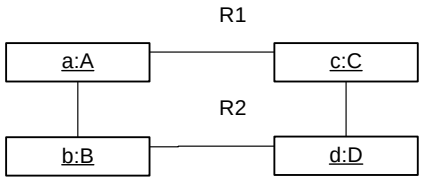
```
package Domain
  extends
    Contains(A,B)
end
```




```
package Range
  extends
    Contains(C,D)
end
```



Suppose that we wish to construct a relation from Domain to Range that preserves the containment structure. To do this we set of two sub-ordinate relations R1 from A to C and R2 from B to D and then express a constraint requiring that the following diagram commutes:

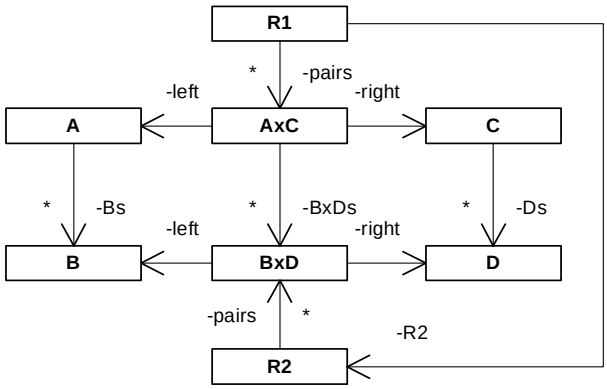


This can be expressed by setting up a dependency between the relations R1 and R2. Each R1 pair contains an R2 pair such that the diagram above commutes. Since this does not depend on the particular classes A, B, C and D this can be expressed as a template:

```
package PreserveContainment(R1,A,C,R2,B,D)
  class <<A + "x" + C>>
    <<B + "x" + D+ "s">> : Set(<<B + "x" + D>>);
  end
  class <<R1>>
    <<R2>> : <<R2>>
    inv
      <<"Every " + "x" + D + " satisfies " + R2>>
        self.pairs->foraAll(p |
          p.<<B + "x" + D + "s">>->forAll(p' |
            self.<<R2>>.pairs->includes(p')))
      <<B + " maps to " + D>>
        self.pairs->forAll(p |
          p.left.<<B + "s">>->forAll(c |
            p.<<B + "x" + D+ "s">>->exists(p' |
              p'.left = c and
              p.right.<<D + "s">>->exists(c' |
                p'.right = c')))) and
        self.pairs->forAll(p |
```

```
            p.right.<<D + "s">>->forAll(c |
              p.<<B + "x" + D+ "s">>->exists(p' |
                p'.right = c and
                p.left.<<B + "s">>->exists(c' |
                  p'.left = c')))
          fail: "Container fails to commute."
      end
    end
end
```

The package PreservesContainment(R1,A,C,R2,D,D) produces the following model:



Each AxC pair contains a BxD pair that preserves the containment. Consider an instance p:AxC. Navigating left produces a set of B instances(Bs), navigating right produces a set of D instances (Ds). The pair p also contains a set of BxD pairs that must be a sub-relation of R2. The every element of the set Ds must be related to a corresponding element of Ds.

This structure exhibits a pattern that occurs in many different variations on models. The model given above is perhaps one the of the simplest examples where the structure is preserved directly. A simple variation occurs where there is a modelling element in between an A and a B which may or may not be preserved by the relations. Another variation might require a particular number of Bs to be related to a different number of Ds.

The same type of structure arises when types are preserved:

```
package PreserveTypes(R1,A,C,R2,B,D)
  class <<A + "x" + "C">>
    type : <<B + "x" + D>>;
  end
  class <<R1>>
    <<R2>> : <<R2>>;
    inv
      "Preserve types"
      self.pairs->forAll(p |
            p.<<B + "x" + D>>.left = p.left.type and
            p.<<B + "x" + D>>.right = p.right.type)
    end
  end
end
```

The following package defines a relation that preserves the containment structure of concrete models:

```
package PreserveCContainment
  extends
    Free,
    PreserveContainment(
      P<->P,CPackage,CPackage,
      P<->P,CPackage,CPackage),
    PreserveContainment(
      P<->P,CPackage,CPackage,
      C<->C,CClass,CClass),
    PreserveContainment(
      C<->C,CClass,CClass,
      A<->A,CAttribute,CAttribute),
    PreserveTypes(
      A<->A,CAttribute,CAttribute,
      C<->C,CClass,CClass)
end
```

The relation defined by PreserveCContainment does not state anything about the names of the elements.

X. Copying

A copy of a model requires that the structure of the modelling element is preserved and no names are changed:

```
package CopyConcreteModels
  extends
    PreserveCNames,
    PreserveCContainment
end
```

The package CopyConcreteModels defines relations P<->P, C<->C and A<->A. Consider an instance r:P<->P and a package p such that p is in the domain of the relation; this is achieved by requiring that there exists a pair x in r.pairs such that x.left = p. The corresponding element of the domain is *read off* the relation as the package x.right. The properties of the package p' = x.right are as follows: (1) p.name = p'.name since SameName(P<->P,CPackage,CPackage,String); (2) the classes of p' must be the image of the C<->C related corresponding classes of p.CClasss since PreservesContainment(P<->P,CPackage,CPackage,C<->C,CClass,CClass); (3) for each class c' in p'.CClasss corresponding to a class c in p.CClasss, c'.name = c.name since SameName(C<->,CClass,CClass,String); (4) the attributes of c' must be the image of the A<->A related corresponding attributes of c; (5) for each attribute a' of c' corresponding to an attribute a in c, a.name = a'.name.

XI. Renaming

A renaming is a relation that holds between model elements. The elements in the domain and range of the relation differ only with respect to the names of certain elements, in all other respects they are copies.

Renaming is based on a simple template that relates elements in the domain and range:

```
package Rename(new,old,N,D,R,T)
   extends
      Relation(N,D,R),
      Named(D,T),
      Named(R,T),
   class <<N>>
      inv
         <<old + " becomes " + new>>
            self.pairs->forAll(p |
               p.left.name = old implies
                  p.right.name = new and
               p.left.name <> old implies
                  p.right.name = p.left.name)
      end
   end
end
```

In general renaming operators are applied to containers of model elements. For example the renaming operator [Y/X] can be applied to a package to renaming the class X to become Y. Therefore in order to construct a renaming relation we must know the domain and range of the relation and construct an an appropriate structure and name preserving/changing relation. For example, suppose that:

[Y/X] : CPackage -> CPackage

we must construct a relation on packages that preserves the name of packages and attributes, preserves the containment structure of packages, classes and attributes, changes the name of classes called Y to X otherwise preserving the name of classes:

```
package RenameClassXToY
   extends
      PreserveContainment(
         P<->P,CPackage,CPackage,
         P<->P,CPackage,CPackage),
      PreserveContainment(
         P<->P,CPackage,CPackage,
         C<->C,CClass,CClass),
      PreserveContainment(
         C<->C,CClass,CClass,
         A<->A,CAttribute,CAttribute),
      SameName(P<->P,CPackage,CPackage,String),
      Rename(Y,X,C<->C,CClass,CClass,String),
      SameName(A<->A,CAttribute,CAttribute,String),
      PreserveType(
         A<->A,CAttribute,CAttribute,
         C<->C,CClass,CClass)
   end
end
```

Given a particular language it is possible to define templates for defining renamings on various model elements. Following from the package RenameClassXToY we can construct a template for renaming classes:

```
package RenameClass(N,new,old)
      extends
         PreserveContainment(
            N,CPackage,CPackage,
            N,CPackage,CPackage),
         PreserveContainment(
            N,CPackage,CPackage,
            C<->C,CClass,CClass),
         PreserveContainment(
            C<->C,CClass,CClass,
            A<->A,CAttribute,CAttribute),
         SameName(N,CPackage,CPackage),
         Rename(new,old,C<->C,CClass,CClass,String),
         SameName(A<->A,CAttribute,CAttribute,String),
         PreserveType(
            A<->A,CAttribute,CAttribute,
            C<->C,CClass,CClass)
      end
   end
```

Renamings can be applied to model elements contained by an element specified by name. For example the following renaming:

[A.Y/A.X]:CPackage -> CPackage

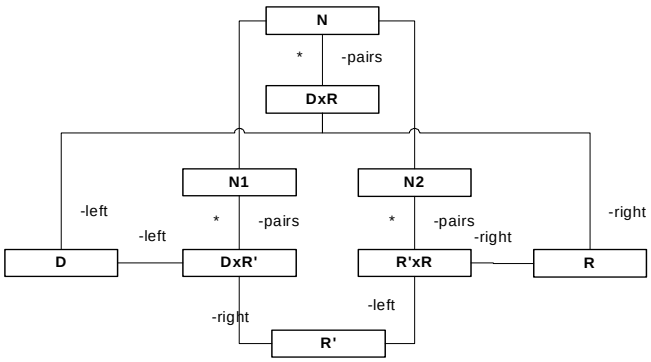
changes the name of the attribute named X in the class named A in the range to be an attribute named Y in the class named A in the range.

```
package RenameAYAX
   extends
      PreserveContainment(
         P<->P,CPackage,CPackage,
         P<->P,CPackage,CPackage),
      PreserveContainment(
         P<->P,CPackage,CPackage,
         C<->C,CClass,CClass),
      PreserveContainment(
         R1,CClass,CClass,
         R2,CAttribute,CAttribute),
      PreserveContainment(
         R3,CClass,CClass,
         R4,CAttribute,CAttribute),
      SameName(P<->P,CPackage,CPackage,String),
      SameName(R1,CClass,CClass,String),
      SameName(R2,CAttribute,CAttribute.,String),
      ConstantName(A,R3,CClass,CClass),
      Rename(Y,X,R4,CAttribute,CAttribute),
      Or(C<->C,R1,R3,CClass,CClass),
      PreserveType(
         R2,CAttribute,CAttribute,
         C<->C,CClass,CClass),
      PreserveType(
         R4,CAttribute,CAttribute,
         C<->C,CClass,CClass)
   end
```

XII. Relational Joins

Relations that are defined in isolation can be joined together to produce a new relation providing that the domain of one relation is the same as the range of the other. Relational joins provide a way of building relations in a modular way.

Consider two relations N1 with domain D and range R', and N2 with domain R' and range R. The join of N1 and N2 produces a new relation N with domain D and range R such that the domain elements of N are the same as N1 and the range elements of N are the same as N2. For every pair of N elements there must be an N1 pair p1 and a N2 pair p2 such that p1.right = p2.left. The structure of a join is shown on the following diagram:



The join of two relations is a general pattern that can be defined as a template:

```
package Join(N,D,R1,R2,R)
   extends Relation(N,D,R)
   class <<N>>
      <<R1>> : <<R1>>;
      <<R2>> : <<R2>>;
      inv
         <<"Join covers domain of " + R1>>
            self.dom() = self.<<R1>>.dom()
         fail: "Domain of join is incorrect."
         <<"Join covers range of " + R2>>
            self.ran() = self.<<R2>>.ran()
         fail: "Range of join is incorrect."
         <<"Join pairs from " + R1 + " and " + R2>>
            self.pairs->forAll(p |
               self.<<R1>>.pairs->exists(pairR1 |
                  pairR1.left = p.left and
                  self.<<R2>>.pairs->exists(pairR2 |
                     pairR2.left = pairR1.right and
                     pairR2.right = p.right)))
         fail: "Incorrect join."
      end
   end
end
```

Renaming provides an example of relational joins. A renaming can be defined as a structure preserving relation between models. Given two renamings, the composite renaming is constructed by joining the two sub-renamings. The following renaming changes the name of classes from X to Y and from A to B:

```
package RenameClassXtoYAndAToB
   extends
      RenameClass(R1,Y,X),
      RenameClass(R2,B,A),
      Join(R,CPackage,R1,R2,CPackage)
end
```

XIII. Stamping Out Templates

Templates are models in which the names of elements are string expressions. Templates are *stamped out* in a context in which the string expression variables are associated with strings. Stamping out copies the structure of the template body and replaces the string expressions with their values.

Stamping out relies on a pattern of sting expression evaluation that can be expressed as a template:

```
package Eval(env,N,field,Domain,Range)
   extends
      Relation(N,Domain,Range),
   class <<N>>
      inv
        <<"Eval "+Domain+" to "+Range+" " + field>>
            self.pairs->forAll(p |
               p.left.<<field>>.eval(env) = p.right.<<field>>)
        fail: "Range is not value of domain expression."
      end
   end
end
```

The relation between TemplateModels and SyntaxModels can now be expressed in terms of structure preservation and name evaluation:

```
package StampOut(env)
   extends
      TemplateModels,
      SyntaxModels,
      PreserveContainment(
         P<->P,TPackage,SPackage,
         P<->P,TPackage,SPackage),
      PreserveContainment(
         P<->P,TPackage,SPackage,
         C<->C,TClass,SClass),
      PreserveContainment(
         C<->C,TClass,SClass,
         A<->A,TAttribute,SAttribute),
      Eval(env,P<->P,name,TPackage,SPackage),
      Eval(env,C<->C,name,TClass,SClass),
      Eval(env,A<->A,name,TAttribute,SAttribute),
      Eval(env,A<->A,type,TAttribute,SAttribute)
end
```

XIV. Resolving Names in Abstract Syntax

Syntax models contain attributes whose types are names of classes in the containing package. These names must be *resolved* by replacing the names with the appropriate classes.

Syntax resolution can be expressed as a relation from SyntaxModels to ConcreteModels. The relation preserves the structure of the domain and the types of the model elements. An extra constraint is added to the component relation that associates packages with packages in order to require that the names of attributes are resolved:

```
package Resolve
  extends
    SyntaxModels,
    ConcreteModels,
    PreserveContainment(
      P<->P,SPackage,CPackage,
      P<->P,SPackage,CPackage),
    PreserveContainment(
      P<->P,SPackage,CPackage,
      C<->C,SClass,CClass),
    PreserveContainment(
      C<->C,SClass,CClass,
      A<->A,SAttribute,CAttribute),
    SameName(P<->P,SPackage,CPackage,String),
    SameName(C<->C,SClass,CClass,String),
    SameName(A<->A,SAttribute,CAttribute,String)
  class SPackagexCPackage
    inv
      "Resolve names"
      self.SClassxCClasss->forAll(p |
          p.SAttributexCAttributes->forAll(p' |
            p'.right.type =
              self.right.CClasss->select(c |
                c.name = p'.left.type).selectElement()))
      fail: "Attribute types not resolved."
    end
  end
end
```

XV. Flattening Nested Package Structure

A variation on the PreserveContainment template is FlattenContainment. This can be used to construct a relation from a language supporting models containing nested packages to a language that supports just top-level packages. The relation preserves all the structure except package nested is flattened. When flattening nested packages the names of the contained classes must be modified in order to make then unique when they are lifted to the outer package. A suitable strategy is to pre-pend the name of the containing package to the names of classes that it contains when they are lifted out.

XVI. Conclusion

This paper has described an approach to developing languages for object-oriented modelling. The approach is based on packages, package specialisation, templates, renaming and relations between sub-languages. The approach has been exemplified in terms of a small static modelling language; the meta-models for the language are defined using the key technologies.

The approach must be applied to richer modelling languages in order to be useful in practice. Relations must be used to define package specialisation. The key feature of package specialisation is a relation between a sub-package and a collection of super-packages; the relation preserves the structure of the super-packages whilst defining how inherited modelling components are merged.

As seen in the example templates given in this paper, string expressions can be used in the bodies of methods and constraints. The mechanisms used to stamp out more complex template bodies are the same as those defined in this paper. For example, a language of templated expressions can be defined where literal names (slot names and method names) are supplied as string expressions. The StampOut relation must then map between templated expressions and expressions, replacing the string expressions with strings as described here for model element names.

In 2001 the Object Management Group launched a key initiative to define a framework for Model Driven Architecture (MDA) [1] [8]. MDA aims to model all aspects of the system development process including software artifacts, business processes and development processes. The relationship approach described in this document provides a framework within which all features of a development can be specified. Static modelling notations, such as UML, use associations as a basis for relationships; the relationships defined in this paper are more expressive than associations. The semantics of associations is described in [11].

The relations described in this paper exist at the meta-level of the modelling framework. In order to realise the goal of MDA relations must be available to the modeller at the application level; our next aim is to provide new modelling structures whose semantics is expressed using the constructs given in this paper.

XVII. References

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

[2] 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

[3] Clark A., Evans A., Kent S. (2000) The Specification of a Reference Implementation for UML. Special Issue of L'Objet on Object Modelling, 2001.

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

[5] Clark A., Evans A., Kent S. (2002) Engineering Modelling Languages: A Precise Meta-Modelling Approach. Presented at the ETAPS FASE Conference, Grenoble France, 2002

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

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

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

[9] 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.

[10] 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.

[11] Gogolla M., Richters M (2002) Expressing UML Class Diagram Properties with OCL. In Clark A, Warmer J. (eds) Advances in Object Modeling with the OCL. Springer Verlag, LNCS 2263.